University of Paderborn
Fürstenallee 11
33102 Paderborn

Biologically Inspired Methods for Organizing Distributed Services on Sensor Networks

Dissertation

A thesis submitted to the
Faculty of Computer Science, Electrical Engineering and Mathematics
of the
University of Paderborn
in partial fulfillment of the requirements for the
degree of *Dr. rer. nat.*

Tales Heimfarth

Paderborn, Germany
November 27, 2007

Supervisors:

1. Prof. Dr. rer. nat. Franz J. Rammig, University of Paderborn
2. Prof. Dr.-Ing. Ulrich Rückert, University of Paderborn

Date of public examination: Dezember 21, 2007

Acknowledgements

I am grateful to several persons that contribute to this work. I am deeply in indebted to Prof. Dr. rer. nat. Franz J. Rammig, who oriented and encouraged me during the development of my research. I am thankful for the opportunity to work and share experiences in his research group.

Also, I am grateful to Prof. Dr.-Ing. Ulrich Rückert, for the support and for the fruitful time in the "Graduiertenkolleg".

A special thank to Prof. Dr. rer. net. Flávio R. Wagner, who always contributes to my research and kindly helps me during my academic journey.

Furthermore, I would like to express my thanks to my colleagues Peter Janacik, Johannes Lessmann and Dalimir Orfanus for contributing to this work. Without the many discussions with my colleague Peter, this work would not be possible. In addition, the development of our network simulator would not be possible without the valuable contributions of Peter and Johannes. The cooperation with Dalimir will bring fruitful results for our future research.

It is also important to thank our working group, for the opportunity to research in a single environment where the cooperation is very strong.

Very important to the development of this thesis was my wife Carolina A. Heimfarth. She gave me incentive and support in all moments during the development of this work.

Last but not least, I would like to thank my parents, Celso D. Heimfarth and Carmen T. L. Heimfarth, who supported my education and are always present in my life.

Contents

List of Figures

List of Tables

Abstract

Wireless sensor networks (WSN) enable a myriad of new applications, e.g. human-embedded sensing, habitat exploration and ocean data monitoring. Nevertheless, they have different requirements from conventional systems. Self-configuration, energy-efficient operation, collaboration and in-network processing are examples of important requirements. In order to achieve these requirements, the system software of a sensor node plays a fundamental role: it should provide useful abstractions to enable the development of the applications and at the same time comply with the constrained resources of the sensor nodes.

The range of possible applications in a sensor node covers distinct tasks like clock synchronization, data acquisition, signal processing and data fusion. The traditional approach in this area is to provide operating system concepts with dramatically reduced functionality. In this work, we present an alternative approach. Our OS provides potentially arbitrary functionality that dynamically adapts to the actual profile of requirements. The basic idea is to offer services that are distributed over a cluster of nodes instead of having the entire system on each node.

Cooperation is the keyword to achieve complex tasks using the restricted sensor nodes. Our operating system (OS) supports this cooperation among neighboring nodes using the concept of distributed services. We are combining the typical OS functionality with the middleware one. Our system is responsible to coordinate the migration and placement of those services. For that, we develop a biologically inspired heuristic responsible to drive the placement of the services in the WSN. We develop two version of the heuristic, with different complexity and performance. Both are completely distributed and based on local information and local rules. The communication necessary for organizing the services is done by means of stigmergy.

Further, we present two clustering heuristics responsible to decompose the network graph into connected sub-graphs (called clusters). Each cluster will hold a complete instance of the OS and application. With the network divided in clusters, the organization overhead is reduced, since protocols that rely on some global information are restricted to a single cluster. This enhances the scalability of the system.

The clustering problem is called minimum intracommunication-cost clustering. The idea is that a minimum amount of resources must be present in each cluster and the clusters should be well connected. The first heuristic is able to handle networks with low topology changes, whereas the second can deal with moderate changes. Both heuristics rely on the principle of division of labor in social insects.

We evaluate our heuristics using the Shox wireless network simulator. The service distribution heuristics were able to produce very good assignments, near to the optimal, for most experiments. Our clustering heuristic for systems with low topology changes outperforms an existing heuristic (expanding ring) in term of cost for most cases. It was able to produce clusters that were, at most, in average 1.43 times the optimal for all simulated scenarios. Moreover, the results have low standard deviation.

Several enhancements can be done in our heuristics. In order to better distribute the burden imposed on the clusterhead, clusterhead rotation may be included in the emergent clustering. Moreover, an additional negotiation phase at the end of the heuristic may be included to improve the performance of the heuristic.

Moreover, we aim to combine the concept of our OS with the control script mechanism present in virtual machines for WSN. This will enable a straightforward development of data-centric scripts that use the extensive functionality of our distributed services to achieve complex goals.

Chapter 1

Introduction

Wireless sensor networks (WSN) belong to a new class of networks composed by lightweight wireless nodes deployed in a physical environment. Normally, they have the tasks of sensing it and reacting to the sensed values. Each node is equipped with sensors, a processor, some memory and a wireless interface. A myriad of applications can be realized using such networks, e.g. human-embedded sensing, habitat monitoring and ocean data exploration. Vantages of such systems are a very high spacial resolution when increasing the number of nodes used, robustness due to the inherent redundancy, easy deployment and reduced energy consumption.

Due to their specific nature, sensor networks have different requirements from standard systems. Among others, self-configuration, energy-efficient operation, collaboration and in-network processing are very important requirements. All layers in a sensor are designed to cope with those requirements and cross-layer optimizations are very common in those systems. The system software of a sensor node has a fundamental role in the sensor: at one side, it should provide useful abstractions and a programming model to the applications' programmer. At the other side, it must comply with the requirements of the sensor network.

A very important requirement, which limits the system software functionality, is that a sensor OS must have the ability to manage a very constrained hardware. More precisely, the constrained memory poses a big challenge. The amount of functionality that may be present in each node is limited. Therefore, cooperation between nodes is needed in order to accomplish complex tasks. Spacial correlation among neighboring sensors represents also a motivation for local cooperation. In our opinion, such cooperation among nodes must be well-supported by the system software of a sensor.

Due to the high cost of the communication in WSNs, it is a better choice, for several scenarios, to process the data locally in order to reduce its amount before forwarding it to the access point (or other sink). Moreover, if a higher autonomy of the network is desired, where the nodes may react by themselves to changes in the environment, this local processing is inevitable. Nevertheless, as already said, each node has a constrained hardware and limited functionality. Hence, the work must be distributed over multiple nodes, which cooperate towards the desired result. The system software must provide abstractions that enable this cooperative work.

In this work, we propose the NanoOS, an operating system (OS) for sensor nodes that enables the automatic distribution of services among the nodes of a sensor network. The idea of NanoOS is to unify in one system typical OS functionality with middleware one. This means that the NanoOS is responsible to manage the local resources of the nodes and to present appropriate abstractions for the local processing as well as to distributed processing among the sensor nodes.

Our main abstraction is the processing thread, that describes the execution of code associated with a state. The processing thread can be tasks, that are started in a particular node and aren't mobile, and services, whose functionality is shared among other services and tasks. Mobile services are services that may migrate among the nodes in the system. The mobile services can be made available by our OS or by the applications. Such services can be used, for example, to process the data generated in the sensors, locally transforming it in a more high level indication, that will be transported to the access point.

With our service architecture, we are aiming to support not just WSN traditional applications like data fusion, but also provide means to an efficient development of distributed algorithms on top of the sensor network.

As a requirement for our system, the distribution of the services among the sensor nodes should be done in a transparent way. It is expected that the NanoOS coordinates the migration and placement of the services in the system. In this work, we develop a biologically inspired heuristic that is responsible for controlling the migration and optimizes the initial service placement. Both types of services, OS and application, may be migrated automatically by our heuristic. We develop two versions of the heuristic, with different complexity and performance. Both are completely distributed and based on local information and local rules. Furthermore, the necessary communication used by our heuristics is done by means of stigmergy, i.e., cues are left in the environment instead of direct messages exchange. The main objective of the two heuristics is to minimize the amount of communication of the system, i.e., modules that have intense interaction among themselves should be placed at nearby positions.

The presented migration results in a natural grouping of the communicating modules. Instead of leaving this weak kind of clustering, we decide to make a hard separation of the nodes of the system into clusters. Each cluster will hold a complete instance of the OS and application, i.e., all services needed by the processing threads inside a given cluster must be instantiated in the same cluster. The decomposition of the network in clusters brings several advantages. First, the organization overhead is reduced, since protocols that rely on some global information are restricted to a single cluster. For example, the service discovery process and the routing tables can be restricted to contain mainly information about modules and nodes residing in the same cluster. Moreover, the clustering may be implemented as a layer in the protocol stack and used by other layers. For example, the medium access layer may use the cluster in order to increase the communication system capacity through the promotion of the spatial reuse of the wireless channel. Another example of benefit of clustering is that a topology control may be constructed upon it.

The creation of a hierarchy in the network was also proposed as a means of achieving scalability. Centralized algorithms may be applied locally to one cluster, not compromising the scalability of the system as a whole.

We call our clustering problem the minimum intracommunication-cost clustering. The idea is that a minimum amount of resources must be present in each cluster. In addition, it is assumed that the nodes inside a cluster will heavily communicate, therefore, we desire well-connected clusters, i.e., where nodes belonging to the same clusters have very good links among them. Two biologically inspired heuristics are proposed in this work to decompose the network in clusters. The first was developed for clustering networks with low topology changes and the second one can cope with moderate changes.

The heuristics are based on the election of a subset of nodes that represent the clusters. Those nodes are called clusterheads. When a clusterhead emerges, it starts to look for members. The clusterhead election is based on the division of labor and task allocation in social insects. In social insects, different tasks are performed by specialized individuals. In the same way, our clusterhead election procedure allocate the task of coordinating a cluster to the node that is more suitable.

The members in both heuristics are selected based on their fitness to be included in the cluster.

This fitness is composed of different parameters, like connectivity to the cluster and distance to the clusterhead. Each clusterhead tries locally to select the most connected nodes to be member of the cluster.

In order to estimate the quality of the links in the network, we are also proposing a combined link metric, that is responsible to summarize the goodness of a link. All the heuristics presented in the thesis are based on this metric, i.e., we measure the quality of the service assignment and cluster construction based on this metric.

The performance of the proposed approaches are evaluated using the Shox wireless network simulator. Shox is a discrete-event simulator developed in our working group targeting the simulation of ad hoc wireless networks. Results show that the proposed service distribution is able to produce very good assignments, near to the optimal, for most of our experiments. This incurs a much lower energy consumption than the initial assignment for the service/task communications.

We have also tested our clustering heuristic for static topologies. For comparison, a modification of the expanding ring clustering algorithm was used. Our heuristic outperforms the expanding ring in every analysed scenario. The clustering cost produced by our approach was near the optimal for several cases, and the average cost was at most 1.44 the optimal for all tested scenarios. Moreover, our heuristic produces more predictable clusters than the expanding ring.

Document Outline

Chapter 2 provides the state-of-art of system software for wireless sensor networks. Three types of systems are analyzed in this chapter: operating systems, middlewares and virtual machines. The advantages and shortcomings of each type of system software are analyzed, and the differences to our NanoOS are highlighted.

In Chapter 3, the architecture of our OS is described. Moreover, we present some applications that would profit from our service distribution approach. In addition, our link metric is introduced in this chapter. As already said, it is the basis for our service distribution and clustering heuristics.

Chapter 4 presents a basic and extended heuristic to control the migration of the mobile services. First, the related work about migration of components in distributed systems and more specifically, in WSN is presented. Further, the formal problem description is introduced and the heuristics developed to solve the problem are shown in details.

The related work concerning cluster construction in ad hoc networks is presented at the beginning of Chapter 5. Moreover, we describe formally our optimization problem and prove that it is NP-complete. Further, we present our two heuristics that are able to decompose the networks to clusters. At the same time, concepts from self-organizing and emergent systems that are used in the heuristics are also described.

The experiments realized with the corresponding results are presented in the Chapter 6. Moreover, we present in this chapter how we evaluate the results. This is made with reference approaches, and the results are normalized against this reference. For small cases, we use as reference the optimal solution calculated with integer linear programming (for the clustering problems) and branch-and-bound (for service distribution). A genetic algorithm is responsible for delivering the reference cost in large problems. At the end of the chapter, the results of several realized experiments are analysed.

Finally, Chapter 6 presents the conclusions of this thesis.

Chapter 2

System Software for Wireless Sensor Networks

In this chapter, the state-of-art of system software for wireless sensor network is presented. By system software, we mean software components providing application-independent services and managing node resources [79]. We will survey the different kinds of system software for sensor networks, as operating systems, middleware, and virtual machines.

2.1 Embedded System OS

There are several operating systems for embedded platforms such as, for example: *Vxworks*, *WinCE*, *PalmOS*, *QNX*, *Apertos*, and *μLinux*. Regarding their development aims, these systems split into two groups: *general purpose embedded OS* and *specific application(-oriented) OS*. The upper part of Figure 2.2 depicts these groups and their members.

Many embedded system OS do not comply with the required properties for sensor nodes. In most cases, their memory and performance requirements can only be satisfied by platforms larger than sensor nodes by one order of magnitude. This fact is depicted in Figure 2.1. The figure shows the design space that sensor OS are targeting. It further demonstrates that the footprint of sensor OS is smaller than the footprint of PDA-class operating systems by one order of magnitude.

2.1.1 Configurable Operating Systems

Some embedded commercial operating systems like *VxWorks* [125] have a rather fine-grained service architecture and permit the *configuration* of the services tailored to the hardware and the application. This modularity and configurability is also a desired characteristic in a sensor OS because it reduces the resource requirements. Nonetheless, most OS of this class have a too large footprint for sensor network use. Another point, also stressed here, is that the existing embedded system's OS cannot appropriately cope with the dynamic behavior of the nodes (and the topology) of a sensor network. They don't provide the support for the type of applications running on the sensors (e.g. data fusion, database-like queries, data dissemination/gathering, etc).

Another group of OS goes a step further and permits *dynamic reconfiguration* of the set of running services in order to adapt themselves to changed situations (e.g. change of the set of running tasks). Such operating systems, like the academic *Apertos* [139], are known as *reflective operating systems*. A *reflective operating system* has the ability to *reflect about its actual state* and, based on

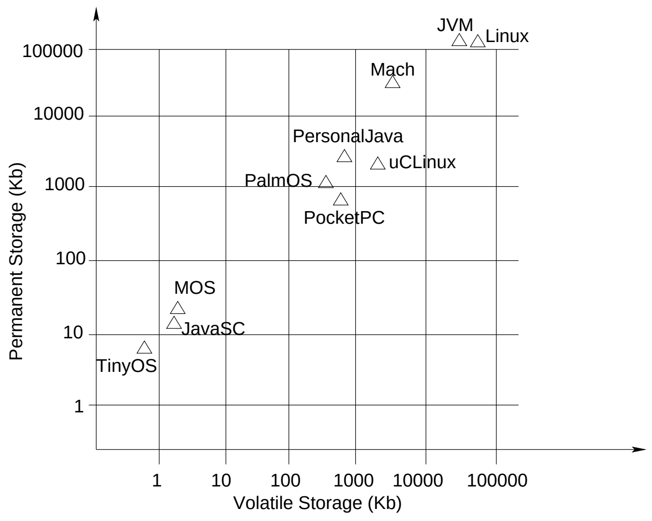


Figure 2.1: Design Space of Sensor OS [123].

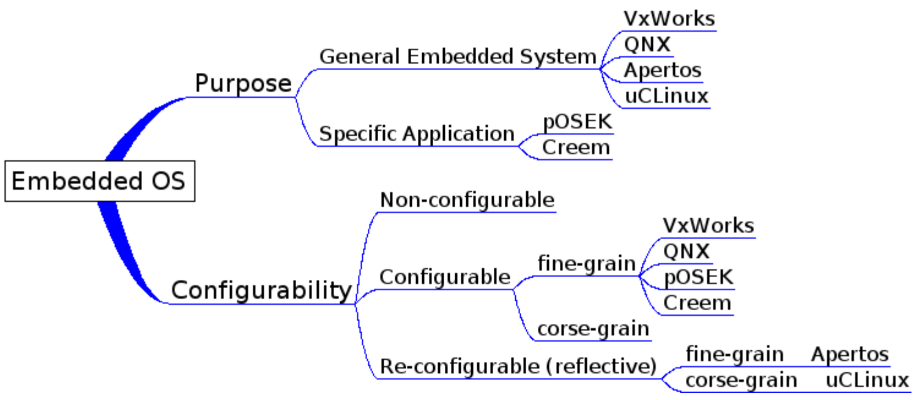


Figure 2.2: Classification of Embedded OS

this reflection, to change the current structure in order to self-adapt to the new requirements of the environment or the applications *by using reconfiguration*. The services in such operating systems are implemented as objects with a correspondent *meta-object*, which *analyzes* the behavior of the object and the requirements of the applications during run-time and re-configures the operating system in order to better serve the application's requirements. Although this is a highly desirable characteristic in a dynamic sensor network environment, the existing reflective operating systems have the following drawbacks (as presented above): they are not designed to take in account the diverse dynamics of sensor networks and they have a too large footprint for this network class in most cases. Moreover, they are not designed to support typical WSN applications.

2.2 Sensor Network OS

A sensor operating system must have a very small footprint and, at the same time, it must provide a limited number of common services for application developers. These services comprise hardware management of sensors, radios, and I/O buses, and devices such as external flash memory. Moreover, task coordination, power management, adaptation to resource constraints, and networking are also required services [123].

The OS should support the specific needs of the WSN. For example, they must support energy management. An appropriate programming model and a clearly way to structure a protocol stack are also necessary.

For example, in order to cope with the demands of sensor nodes, a series of OS were proposed/developed. Bertha can deal with some demands of WSN, whereas TinyOS, PeerOS and MANTIS OS are specifically designed for sensor networks. A brief description of a selection of OS for sensor nodes will be provided further below.

We will discuss in the next section some specific aspects relevant to a sensor network OS. We divided the aspects in two groups: single node and node group concerns.

2.2.1 Single Node Concerns

2.2.1.1 Hardware Management

A main task of an OS is the management of the hardware resources of the node [116]. The OS should provide abstract services (e.g. sensing and data delivery to neighbors). As there is no memory management unit (MMU) in a typical sensor processor, this hardware management can be implemented by means of a library of functions.

The lack of an MMU leads to the consequences that there is no protection from erroneous hardware (and memory) access.

2.2.1.2 Task Coordination

Another major problem of the sensor network OS is the task coordination of multiple tasks [123]. The OS should allocate the processor to a certain task and also control the synchronization among the tasks (mutual exclusion).

There are two basic approaches: to leave the coordination to the tasks or to handle this inside the OS. Doing that inside the OS has two drawbacks: CPU bandwidth and memory utilization. On the other hand, it frees the application developer from the complexity of task coordination and usually supports the development of more elaborate and complex applications with less effort.

Usually these two task coordination approaches are implemented in a sensor network OS using event-based and preemptive thread multitasking paradigms.

Event-based Kernels Tasks are implemented as event handlers that run until completion. This provides concurrency without the need of mechanisms like per-thread stacks or mutual exclusion. The main advantage of this approach is the small memory requirement: because processes cannot block, just a global stack is necessary, saving the scarcest resource of a sensor node. A main problem occurring in event-driven systems is the difficulty to implement applications using state-driven programming: the event-driven model is hard to manage by the programmers and not all problems are easily described as state machines. Moreover, most existing applications are written for preemptive multithread environments.

Another problem is the description of concurrency, since, when a handler is running, all others are blocked.

Preemptive Thread Multitasking Kernels Preemption leads to the necessity of saving the current state of the registers in the stack. This means that one stack per thread is necessary, leading to a relative high memory requirement. Moreover, the context switch operation is rather time-consuming, that means, for a task set composed mainly of IO bound tasks or small tasks, the overhead caused by the context switch is very high. Since sensor networks have a resource constrained hardware, this is a strong argument against this OS paradigm.

Nonetheless, preemptive multitasking supports the development of more complex, elaborated distributed applications. Moreover, existing embedded applications can be ported more easily to such an environment. In the NanoOS, we have a preemptive thread multitasking OS. However, we use a special service organization in order to reduce the memory consumption of our OS.

2.2.1.3 OS Architecture

Classical operating systems running on CPU with MMU (memory management unit) have either monolithic kernel, microkernel, or exokernel architecture. The amount of functionalities implemented in the kernel space (which runs in supervisor mode) is a criteria used to classify the architectures.

Monolithic Kernel: Implements all abstractions in the kernel space, including file system, virtual memory, device drivers, networking, etc.

Microkernel: The low level facilities are implemented in the kernel space, whereas the higher-level abstractions are processes in the user space. Moreover, a microkernel uses to be modular and some of them support exchanging of modules by means of changing servers (processes) in the user space.

Exokernel: In an exokernel, nothing is implemented in the kernel space. The kernel just multiplexes the hardware resource used by the user space processes. Hardware events activate stub handlers that pass the events to user-level processes. The user-level processes implement the policies.

A library-based OS is a set of functions that implement abstractions to facilitate the hardware management. Nevertheless, it does not provide memory protection. Library-based OS are mainly used in systems where the processor does not possess a memory management unit. This is usual for sensor nodes, which have small processors without MMU.

A variation of a library-based OS is a component-based one. Each component realizes some abstraction and comprises code and state. They are composable. The OS and the protocol stack are written as a set of components connected to each other. Moreover, in such operating systems the application may be written just as another component in the system or a service interface can be used (Figure 2.3). With a service interface, it is possible to rise the level of abstraction. Nevertheless, the performance and the cross-layer optimization possibilities are penalized [69].

Another important point about the architecture of an WSN operating system is that often the border between communication stack protocols, OS services, and application programs turns to be subtle. Moreover, the standard layering method (in special used in the communication stack inside the OS) is relaxed due to the use of cross-layer optimization methods, where the strict confinement of the layered approaches is loosened.

2.2.1.4 Power Management

The battery inside sensor nodes after their deployment cannot be easily exchanged. In addition, for several applications, a long life time of the network is desired. Moreover, Moore's law does not apply

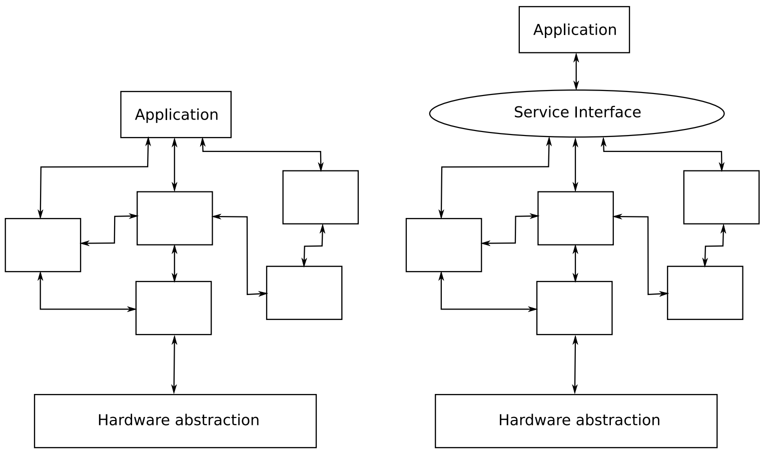


Figure 2.3: Two possible options of interface between the application and a protocol stack: applications as ordinary components or a deliberately designed service interface. Source: [69].

to battery capacity. Due to these facts, mechanisms to assure a good energy utilization using power saving techniques are highly desirable.

There aren't abstractions that remain consistent over the diversity of power management techniques. Nonetheless, several techniques for power management exist [45]:

- duty-cycling - reduces the average power utilization by cycling the power of a given subsystem
- batching - amortizes the high cost of start-up by bundling several operations together and executing them in a burst
- hierarchy techniques - order the operations by their energy consumption and invoke the low-energy ones before the high-energy ones in a fashion similar to the short-circuit techniques used by several compilers for the evaluation of boolean expression in various languages
- redundancy reduction (or even elimination) - uses compression, aggregation, or message suppression

The low-power operation mode in the sensor network can be addressed in various levels. In [45], the following levels have been recognized:

Sensing Sensing is a very important task in a sensor network. The most commonly used technique to lower the energy consumption spent in this operation is duty-cycling, i.e., cycling the power on and off. In a data collection type of application, a sleep-wakeup-sample-compute-communicate cycle can be used where the node sleeps most of the time [46]. For exceptional event detection applications, rare events may pose problems if the sensing subsystem has to be powered continuously.

Communication Communication is very important in a sensor network, but it is also power-consuming. Several techniques has been developed to reduce the radio consumption [45]:

- Radio Management: in order to reduce the energy consumption, several techniques may be used. Polled operation works by sampling the channel periodically (and power down during the remaining time). Scheduled radio works by coordinating in advance when radio may transmit and receive. Triggered operation uses a low-power secondary radio to signal a more capable, but also more power-demanding radio to wake up.

- Middleware/OS level - energy-aware services. For example, the time synchronization, routing, and dissemination services may be implemented in a way that saves energy. Because many applications do not need constant time synchronization, reactive synchronization may be used to save energy. In the routing level, nodes with lack of energy can be preserver as other nodes are used to route the packets [83].

  Services for data dissemination may use effective and energy-efficient algorithms instead of naively flooding the network. Epidemic/gossip techniques as well as meta-data based techniques can be used to reduce redundant transmissions.

- Miscellaneous optimizations include conserving energy at the MAC layer like in PAMAS [117] and S-MAC [138], snooping on application-level packets in the network layers, piggybacking, and batching message transmission. Some of the techniques may be included in OS communication stack or services; others can be implemented in the application.

Computation Often sensor applications are neither computation nor I/O-bound [45]. There are many opportunities for idling the processor and peripherals. Event-driven OS like TinyOS (see Section 2.2.3.1) may implement energy-saving modes of operation. For example, when the task queue is empty, the system can go to a power-save mode until the next interrupt arrives.

Storage The memory hierarchy in sensor needs to cope not only with factors like speed and persistence but also energy-efficiency issues. For example, normally the use of RAM as cache for the EEPROM or Flash may be more energy-efficient that direct operation with those memories.

Energy harvesting For sensors deployed in environments where the replacement of the battery is not possible, the energy harvesting systems might offer an alternative solution. In the software, a management of these energy harvesting devices and of the primary and secondary storage is necessary.

Relation with OS There is a range of mechanisms for power saving in sensor networks. The OS can offer two types of mechanisms to perform power management: implicit and explicit [116]. The implicit power management is done by the OS without cooperation from user or application tasks. This means that the OS can power off some hardware components without participation of the tasks. In the explicit power management, the tasks give hints to the OS using system calls. This is more efficient than the implicit method since the OS has much more information for its power management decisions.

2.2.1.5 Reconfigurability Support

Normally, sensor network OS are composed of a set of modules that can be selected to fulfill the needs of a specific application, i.e., they are configurable. If it is possible to exchange fine-grained modules during execution, in order to adapt the OS to a new environment situation or application requirements, we are speaking about a reconfigurable OS. A reconfigurable OS normally has a reflection unit with is responsible for monitoring the current state of the system, and, based on this state, parameters and modules are changed, aiming at optimizing a desired objective function.

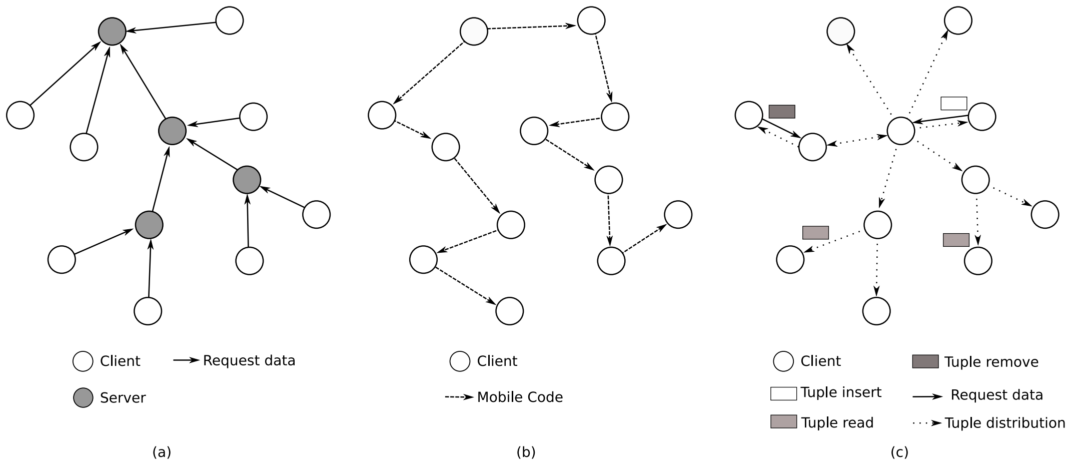


Figure 2.4: Three architectural paradigms for distribution: (a) client-server, (b) mobile code, and (c) tuple space. Source [79].

2.2.2 Group Concerns

In this section, we will deal with concerns related to the distribution of the application among the nodes of a sensor network and its communication. In [79], different approaches of distribution platform are characterized: operating system (OS), virtual machine (VM), and middleware. The aspects presented in this section are relevant for the tree types of platform.

We will discuss in the next sections the support of the OS/virtual machine/middleware to distributed applications.

2.2.2.1 Architectural Paradigms

We distinguish between tree possible architectural paradigms to support distributed applications: *client-server*, *mobile code*, and *tuple space* [79].

The *client-server* approach consists of a set of services providing functionalities accessed by the clients. A directory service is responsible for helping in the service discovery process. The idea here is that the client outsources some task to be processed on the server. The service is called using remote procedure calls (RPC) or, in an object-oriented environment, remote method invocation (RMI). In the remote task communication, normally a stub procedure marshals the call and the parameters, which are unmarshaled back to the server. A similar, but more efficient and commonly used way of communication in WSN, is the active message: it is similar to the RPC, but the sender is not blocked and continues the processing. When the response from the call arrives, an event is used to notify the caller.

In the *mobile code* paradigm, instead of moving data from client to a service, the code is moved to the data it should process. Mobile agents carry also its own state and are autonomous (e.g., they may decide to migrate by themselves). The virtual machine approaches in WSN normally implements mobile agents due to the fact that the functionality (code) can be expressed in a very compact form, being suitable for migration [52].

The concept of *tuple space* may be used for task communication and service discovery. Tuples are collections of passive data values. A tuple space is a pool of shared information, where tuples are inserted, removed, or read. Data are global and persistent in the tuple space and remain there until explicitly removed. For the communication, the partners must not know each other and do not need to exist at the same time.

The three paradigms are illustrated in Figure 2.4.

In this thesis, we will mainly focus on the *client-server* paradigm.

2.2.2.2 Service Discovery

Service discovery allows devices to automatically locate network services with their attributes and to advertise their own capabilities to the rest of the network. It is a major component of modern self-configurable sensor networks. The service discovery is important in the client-server model of distributed computation. Although normally the service discovery is part of a middleware level, as already argued, we believe that in WSN the OS should be merged with middleware functionality due to the resource restrictions.

In a typical service discovery process, there are mainly two tasks. A client can request a service issuing a request to the node's lower layers and expect a service reply. The reply includes the network address of the provider, to be used for subsequent communications. This subsequent communication is modeled by a *service invocation* and *invocation acknowledgment* handshake. Naturally, more or other data messages may be exchanged [50].

The provider node interface is simple: Applications shall be able to register and unregister their services with the service discovery layers.

Most of the service discovery protocols include the client-server paradigm as mode of operation. In this mode, the clients reactively send out service request messages and servers listen to such messages. If the requested service is supported, a reply message is generated and sent back. In another method, the service users listen passively to advertisements that are proactively generated at the servers side. A further alternative scheme involves service brokers (or directory agents) residing between clients and servers as a logical entity. Clients direct the requests to known service brokers, whereas servers register their services with these brokers. The first two models are known as directory-less architectures. It has been argued that the directory-less architecture is more suitable for MANETs due to the absence of infrastructure [71] which may need a costly maintenance.

However, as we will present further in this thesis, in our approach, we are decomposing the network in clusters. The cluster's representative (clusterhead) is suitable for assuming the directory agent role.

Table 2.1 shows different technologies used in local and mobile wireless networks to search for services. The classical service discovery protocols are used in traditional wired networks. The *Service Location Protocol* [56] is used to find services that match the client query. It can be implemented using *directory agents*. Alternatively, it uses multicasts to service providers in order to find services. The *Jini Lookup Service* [94] resides on a node where client and services send advertisements or requests. When using *Salutation* [31], both clients and services uses the Salutation Manager to advertise or request services. In a network without a Salutation Manager, broadcasts are used. *Universal Plug and Play* [36] is very similar to the *Service Location Protocol,* but it works without *directory agents*. Discovery requests are sent using multicast towards servers. After receiving the requests, matching services reply to it.

The classical protocols are not suitable for WSNs due to the fact that they mainly rely on centralized directory agents or maintain a costly multicast tree. Moreover, they are designed to be used in wired networks and are heavyweight.

Several other protocols have been developed to be used in ad hoc wireless networks. Besides the Tuple Space and GSD, almost all other protocols are based on requests/advertisements directed to the other nodes via broad/multicast or other kinds of directory agents (network manager, SANDMAN). Two problems can be recognized in those approaches: either they have a restricted scalability or require intensive communication.

Client-server technologies are limited to nearby nodes and do not scale for large WSNs. GSD and tuple space scale, but they require a large amount of communication, which can be reduced by increasing the distribution of service information [79]. Nevertheless, this brings an increasing memory cost per node.

The centralized approaches (using a service broker) may be used in restricted parts of the network without compromising the scalability. We select this option for our NanoOS, where the scope of the broker is constrained inside a bounded cluster.

Technology	Communication	Scalability	Requirements	Benefits	Drawbacks
		Classical Service Discovery Protocols			
Service Lookup Protocol (SLP)	Requests made to a directory agent or multicasted to servers	Directory agents are central entities, scalability problem	Register to a directory agent or multicast requests	Language independent protocol	Central Directory agents limits scalability
Jini Lookup Service	Lookup service receives requests	Poor, centralized structure, cascade of lookup service allowed	Nodes where lookup service reside	Ability to spontaneously integrate new services	Heavyweight RMI based protocol, centralized structure
Salutation	Clients and servers access their local Salutation Manager (SM) to discover or advertise services	Not well scalable	Salutation manager	When service required, SM with service must be found.	Service discovery can be performed across multiple SM
Universal Plug and Play (UPnP)	Multicast message for advertisement and search for services	For local network	Devices with UPnP layers	Automatic detection when service is down, directory service optional	Periodic multicast messages, made for standard network devices/OS
		MANETs Service Discovery Protocols			
Resource requests	Requests to neighbors	Restricted to neighbors	Resource declaration	One-hop communication	Scalability
Tuple space	Tuple operation	Balancing between memory and scale	Memory pool in each node	Source and target independence	Communication and memory load
Network manager	Name resolution requests to manager	Local manager area, but extensible	Resource managers, register to manager	Scalability due to naming	Name resolution, communication load
Hunting service	Broadcasts hunt service requests	Not restricted	Remote service identification	Lightweight after initiation	First hunt latency and communication load
Bluetooth SDP	Peer-to-peer link	Only nearby nodes at a time	Bluetooth protocol stack	Querying for available services	Scalability, no broadcast
RKS[137]	Advertises for potential clients	Only to nearby clients	Context definitions for services	Advertisements	Scalability
GSD service groups [29]	Service and group advertisements	n-hop diameter	Service registration	Request routing based on group advertisements	Communication load (both advertisements and requests used)
SANDMAN [113]	Service register and client requests to clusterhead	Moderate thorough clustering	Cluster maintenance	No broad/multicast	Cluster maintenance overhead, moderate scalability due to impossibility of request to bigger number of clusters

Table 2.1: Different technologies implementing service discovery [79, 56, 94, 31]

2.2.2.3 Task Allocation

Task allocation is responsible for the assignment of the tasks to the nodes and for the communication scheduler. An overview about this area for sensor network is presented in Chapter 4.

2.2.2.4 Code mobility

Given the fact that sensor nodes have a very limited amount of memory, the nodes cannot store all applications in the local memory. Moreover, during the life time of the network, applications of different kinds may be required in order to respond to diverse complex queries. A migration mechanism that enables new applications to be transferred from a node to the next is desirable. Moreover, our objective is the development of an OS that offers appropriate abstractions for supporting distributed applications. This implies that the migration mechanism is a very important feature of the OS.

For online task allocation, discussed in Chapter 4 and introduced in the previous section, the code mobility is a central mechanism to support it.

As the connections in a sensor network are constructed in an ad hoc fashion, which does not include pre-organization, the mobility of code is also important to deal with the network's dynamics.

2.2.2.5 Support of Network Dynamics

The sensor networks may exhibit a highly dynamic network topology due to node mobility, environmental obstructions, or even hardware failures. This facts lead to new challenges during the development of distributed applications.

The migration of code, as already described in the previous section, is a very important mechanism to support node mobility. Another important point that should be stressed is that the OS (or middleware) should also support the robust operation of the sensor network despite the network dynamics [106]. An important approach to support a "topology independent" processing is the data-centric communication, where network nodes are addressed by using the function or data they provide (e.g. "please return nodes into my vicinity that can measure temperature").

2.2.3 Examples of OS

2.2.3.1 TinyOS

TinyOS [39] was developed by the University of California at Berkeley . It has a very small footprint and provides an efficient management of hardware resources.

The execution model of *TinyOS* is similar to a finite state machine, but it is more easily programmed. It consists of a set of components that are included in the applications when necessary. TinyOS addresses the main challenges of a sensor network: constrained resources, concurrent operations, robustness, and application requirement support. Like other OS, it aims at reducing the burden of application development by providing convenient abstractions of physical devices. An additional goal is to provide a rich expression of concurrency by the component model.

Each TinyOS application consists of a scheduler and a graph of components. The components are described by their interface and internal implementation. An interface comprises synchronous commands and asynchronous events. Therefore, TinyOS is an event-based operating system.

The concurrency model in TinyOS is a two-level scheduling hierarchy: events preempt tasks, but tasks don't preempt other tasks. Each task can issue commands or put other tasks to work. The

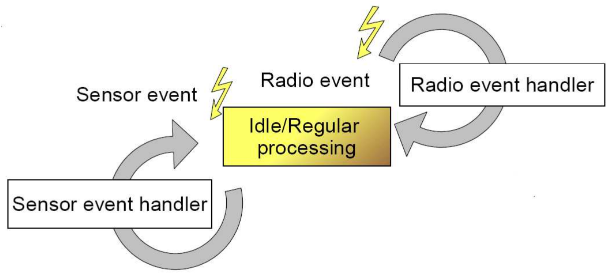


Figure 2.5: Event-based system [69]

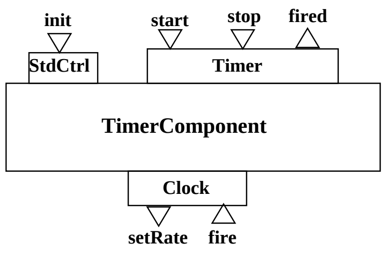


Figure 2.6: Example of a timer module

arbitration between tasks - multiple tasks can be triggered by different events and are ready to execute - is done by a First-In-Fist-Out (FIFO) scheduler.

Events are initiated by hardware interrupts at the lowest levels. They travel from lower to higher levels and can signal events, call commands, or post tasks. Commands cannot signal events.

Wherever a component could not accomplish the work in a bounded limit of time, it should post a task to continue the work. This is because a non-blocking approach is implemented in TinyOS, where locks or synchronization variables don't exist. This means that components must terminate. For that reason, the TinyOS just uses a global stack and each component has a static frame. The components are similar to re-entrant state machines.

As an event-based OS, the system waits for events occurrence and then react upon that. The event-based programming model for sensor networks is illustrated in Figure 2.5.

As commands and events are the only interaction medium between components, a large number of commands and events add up to a large program. The interface of the components consists of a set of commands that the component understands and of events that it may emit.

The language *nesC* allows the application developer to identify interface types that define commands and events that belong together. Components provide certain interfaces and use them from underlying components. An example of a timer component can be seen in Figure 2.6. It receives the command `init` for initialization, `start` for starting the timer, and `stop` for stopping the timer and triggers the event `fired` when the programmed time elapses. The lower component is a clock, from which the timer component receives periodic `fire` events.

Both, the TinyOS components and the application components are implemented using this programming model. There is no support for distributed processing.

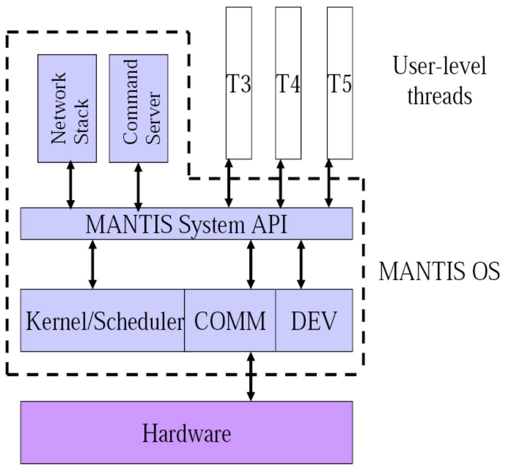


Figure 2.7: Organization of the MANTIS OS [16].

2.2.3.2 Mantis Operating System (MOS)

The Mantis operating system (MOS) is a sensor OS designed to behave similarly to UNIX and provides a larger functionality than *TinyOS*. It is a lightweight and energy-efficient multithreaded OS for sensor nodes.

The design goals of MOS are:

Easy to use : In order to reduce the learning curve of the WSN platform, Mantis is structured using the model of multilayer multithreaded OS. This means that multithreading with a preemptive scheduling scheme is supported by the kernel. Moreover, the kernel (and programs) is written in C, which allows the re-use of existing code.

Flexibility : The system provides flexibility for advanced research in sensor networks. Moreover, remote debugging and dynamic programming of sensors using the network are supported.

The internal organization of the Mantis OS is presented in Figure 2.7 [16]. The system API supports I/O and system interactions. The kernel of the operating system provides UNIX-like thread (and POSIX) functionalities.

In contrast to TinyOS, the MANTIS kernel uses a priority-based thread scheduling with round-robin semantics within one priority level. To avoid race conditions within the kernel, binary and counting semaphores are supported. Moreover, timers and sleep functions are provided. The OS offers a multiprogramming model similar to that seen in conventional OS, i.e., as already said, the OS complies with the traditional multithread POSIX-based paradigm.

All threads coexist in the same address space. The kernel allocates a block of data memory every time when a thread is spawned. The existence of multiple stacks (one per thread) makes MOS more resource-intensive than single-threaded OS (e.g. TinyOS).

The kernel of the Mantis OS also provides device drivers and a network stack. The network stack is implemented using user-level threads and focuses on efficient use of the limited memory. Different layers can be implemented in different threads, or all layers can also be implemented in a single thread. This leads to a trade-off between performance and flexibility. It is possible to the developers to easily modify or replace layer modules of the network stack.

Like TinyOS, Mantis does not offer substantial support in the OS level for distributed processing.

2.2.3.3 Yatos

Yatos [42] is an OS designed by the Federal University of Minas Gerais (Brazil) specifically to run in an WSN environment and has several interesting features. Like TinyOS, it is an event-based OS, where events are mapped to tasks.

The structure of both OS and applications is component-based. Complex behavior can be created using simpler modules. The communication between layers uses the same principle as TinyOS: events travel from lower to upper levels, whereas commands are sent from upper layers to lower ones.

The concurrency in Yatos is achieved using tasks and events. The scheduler has two levels: the high priority level (events) and the low priority level (tasks). The tasks are atomic structures that run to completion, nevertheless they can be preempted by the events. Tasks can send commands and events and schedule additional tasks. Events are generalizations of the interruptions and propagate the processing to the higher levels of the hierarchy (sending events) or to the lower levels (through command execution).

The events in Yatos are classified in tree types: aperiodic, periodic, and oneshot events. Aperiodic events are generated by the hardware (without use of timers). Periodic events are timer events. Oneshot events are programmed and executed only once.

The work on Yatos is currently in progress. It is implemented to run on the sensor node BEAN from the Sensor Net project.

2.2.3.4 Bertha OS

The Bertha OS was developed to manage the hardware of the Pushpin computing platform [86]. The goal of the Pushpin Project is to create sensor networks that self-organize in such a way that they allow preprocessing and condensing sensor data at local sensor level before (optionally) forwarding them to more centralized systems. The OS, hardware, and programming environment follow the design points coming from the Paintable Computing project [23].

Before starting to describe the Bertha OS in more depth, the programming model of the Pushpin project should be described. The main idea of the system is that small algorithmic process fragments can interact with the neighborhood. Based on these interactions, they generate a complex global algorithmic behavior.

The process fragment (called *PFrag*) is the atomic algorithmic unit in the algorithmic self-assembly. It is contained and executed within a single node. Migration is possible to neighboring nodes. Each process fragment implements *install*, *uninstall*, and *update* functions that are called by the BerthaOS.

Each Pushpin node has a complete instance of the OS to manage processor, memory, access to hardware, and system services.

Up to 11 process fragments can be accommodated in one OS instance (one node). The fragments can enter into the node through the wireless interface and are initialized using the *install* function. The *update* functions of the resident Pfrags are called in a *round-robin* fashion and run until completion. This means that the *Bertha OS* does not have preemptive task scheduling.

Abstractions provided by the OS for the communication are BBS (bulletin board system) and a Neighborhood Watch system that keeps a synopsis of the direct neighborhood BBS. The OS also offers migration mechanism to a neighbor node. Figure 2.8 illustrates how the OS and the several offered mechanisms and Pfrags are stored in the memory of the node. Both the OS and PFrags code are stored in a flash memory. This means that, for every migration, the flash RAM must be rewritten. The extended RAM installed in the Pushpin nodes is used to store the state of the Pfrags as well as the Neighborhood Watch and the Bulletin Board System.

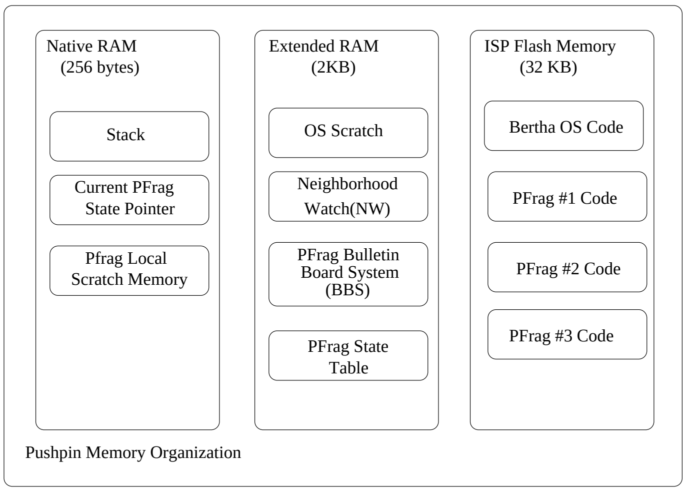


Figure 2.8: Pushpin's memory [86].

In addition to the abstractions already presented, the OS also offers an API that provides access to the different hardware modules.

2.2.3.5 Contiki

The *Contiki* [44] operating system was developed for sensor nodes with a limited amount of resources. It provides dynamic loading and unloading of programs and services during run-time. It supports also dynamic downloading of code enabling the software upgrade of already deployed nodes. All this functionality is offered at a moderate price: the system uses more memory than TinyOS but less than the Mantis operating system.

The main idea of *Contiki* is to combine the advantages of event-driven and preemptive multi-threading in one system: the kernel of the system is event-driven, but applications desiring to use multithreading facilities can simply use an optional library module for that.

A *Contiki* system is partitioned in core and loaded programs. This partition is determined at compilation time. The core comprises the kernel, program loader, run time libraries, and communication system (communication stack and drivers).

The components of a *Contiki* system are:

Process: A process can be an application program or a service. A service implements some functionality used by more than one process.

Kernel: Contains the basic functionality like CPU multiplexing and event handling.

Libraries: Extend the kernel features.

Program Loader: Responsible for loading services and programs on-the-fly.

Processes Processes are application programs or services. The processes share a common address space (i.e., there is no memory protection). The state of a process is held in the process local memory.

A process can be defined by event handler functions and an optional poll handler. Inter-process communication is made by means of posting events.

A service is a kind of shared library. It can be replaced at run time - therefore, it is dynamically linked. Examples of services are the communication protocol stacks, device drivers, and high level functionality like sensor data handling algorithms.

In order to find and manage the services, the OS has a *Service Layer*. It provides a way of binding services based on the textual strings that describe them. A service consists of a service interface and an implementation (that is a process). Applications use a stub library to communicate with the services.

As already said, a service can be replaced during run-time (reconfiguration ability). In order to provide such functionality, the internal state of the service should be preserved during the replacement. For that, the kernel has a call to inform the pointer of the state to the new service and a state description that is generated by the old service.

Kernel The kernel mainly consists of a lightweight event scheduler and a basic CPU multiplexing module. The events can be asynchronous (deferred procedure calls) and synchronous with immediate execution. The control returns to the posting process after the target process has executed the event. This provides an abstraction similar to the interprocess procedure call.

Besides normal events, the kernel provides a polling event with high priority. All events can be preempted by the interrupts. Such interrupts cannot request events in order to avoid race conditions. They must set a polling flag to ask for pool events.

The *Contiki* kernel has no abstractions to deal with power saving issues. Applications must implement power saving mechanisms.

Libraries Libraries offer additional functionality besides that already provided by the kernel. A program can be linked with three types of libraries: static core, part of loadable programs, and services that implement specific libraries that can be dynamically replaced.

An important functionality provided by a library is the stack management function for threads that need a separate stack due to preemption.

The *Contiki* kernel just provides a single shared stack to the processes.

2.2.3.6 Peeros

The objective of the Peeros[97] is the development of a real-time operating system that fits in the limited memory of a sensor node while supporting low power modes and causing small overhead.

The design criteria of the OS are the following: offer priority-based multitasking, offer real-time guarantees to the tasks, support low-power modes, and provide flexibility. Additionally, it should fit in the small memory of the sensor node. The authors of the *Peeros* argue that no other WSN OS can satisfy all those criteria.

The target hardware platform is the EYES sensor node (developed in the context of the EYES project).

The main abstraction of the OS is the task, which is basically a piece of code that can be executed. If several tasks are active at the same time, an EDFI-based algorithm is responsible for scheduling this task-set using preemption. All tasks should have a priority or a deadline, which indicates the relative importance of one task to the other tasks in the system.

In order to accomplish both internal and external communication, a messaging system was developed together with *Peeros*. It consists of three main blocks:

Internal Messaging System: Allows a task to send small data pieces to other tasks.

Serial Messaging System: Permits the data exchange using the serial port.

Radio Messaging System: Similar to the previous component, it allows the data exchange using a radio interface of the node.

In order to reduce the memory requirement of the OS, Peeros supports dynamic loading of device drivers from the EEPROM to the main memory when required. This means that at a specific point of time, not all modules must be loaded in the memory, they just are loaded when necessary.

Besides the interprocess communication subsystem already presented, Peeros doesn't provide more high level abstractions for programming distributed applications.

2.2.3.7 Cormos

Cormos [135] stands for *Communication-Oriented Runtime System for Sensor Networks*. The idea of *Cormos* is to provide a convenient programming abstraction that integrates processing and communication and to use simple and unified internal and external interfaces that makes it easy to provide new system or application components.

The authors of the *Cormos* argue that the communication in a wireless sensor network should be handled as the central abstraction (communication-centric) of the OS instead of being just an extension of the run-time system.

There are two main abstractions in the OS: events and handlers, organized in modules. Modules can extend either application or system functionality. In order to communicate internally and externally, the abstraction of event paths was introduced in the system.

The events trigger local and remote actions. An event is divided in a frame which contains the event state and an array list that comprises a set of handlers responsible for processing the event. The events are created by the handlers to trigger certain actions. After processing all the handlers, the event is automatically deallocated. When events cross node boundaries, they are deallocated in one node and allocated in the next one.

Handlers are processing functions that perform the processing in each node. A handler is executed when an event that contains that handler is scheduled (locally or remotely). During execution, they can create new events, call library functions, or access module variables. Handlers are atomic and run to completion.

The *event paths* are a very important abstraction. A path is a flow of data from a source to destination, specifying the modules that an event will pass through. Paths allow source modules to specify events that will occur in the system.

System Structure Figure 2.9 depicts the structure of the *Cormos* system. It consists of:

Modules: Encapsulate operations that extend the run-time system or are part of the user application. They comprise events, static functions, and variables. Driver modules are special kinds of modules that use hardware interrupts and provide abstractions to the hardware system devices.

Libraries: They are collections of code that can be used in any part of the system.

Run-time system core: Composed of scheduler, memory allocator, and module registry. The scheduler dispatches events following an event path specification. The memory manager maintains a static table for allocation and deallocation of events. The module registry has a table for storing information about all active modules and their handlers.

In order to save power, the scheduler puts the CPU to sleep when there is no pending computation.

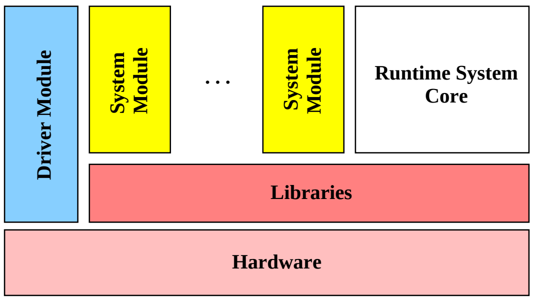


Figure 2.9: Structure of the Cormos system [135].

2.3 Middleware

A middleware for sensor networks is responsible for supporting the development, maintenance, deployment, and execution of sensing-based applications. This includes mechanisms for formulating complex high-level sensing tasks, communication of this task to the WSN, coordination of sensor nodes to split the task and distribute it to individual sensor nodes, and data fusion for merging the sensor readings into a high-level result [107].

2.3.1 Requirements

In [106] and [140], some possible requirements of a middleware for WSN are sketched:

Programming paradigm: Due to the fact that sensor networks are used to monitor environmental phenomena, mechanisms for specification of high-level sensing tasks and combination of sensory data from individual nodes to high-level results are required. A programming paradigm should support the development. Moreover, a data-centric mechanism may be included in the programming paradigm.

Restricted resources: The middleware components must be lightweight, and adaptation of the resource consumption to the actual application needs and availability of resources in the node is desired.

Network dynamics: The middleware should adapt itself to the network topology dynamics, that may change due to mobility, communication failure, hardware failure, etc.

Scale of deployment: The middleware should support mechanisms to self-organize the network even in the presence of thousands or millions of nodes. This means that self-configuration is necessary to achieve an operational state (set up a network topology, assign task to devices, collaboratively merge and evaluate collected data). It is desirable to use local interactions in order to achieve global goals.

Real-word integration: Space and time play a crucial role in sensor networks to identify real-word events and to distinguish these events. Hence, the establishment of a common time and space scale (localization) may be an important service of the middleware.

Application knowledge: Application knowledge can be used to tailor the design and implementation of the middleware services. A trade-off between the degree of application-specific services and generality of the middleware needs to be explored. Nevertheless, some services like data fusion are present in many applications and are often supported by the middleware.

Collection and processing of sensor data: These are the core functionalities of the sensor networks. Complex sensing tasks may require that the data of several nodes are fused in a system-level result. Sensor data may be processed at the local node (or at nearby nodes), features extracted and, after this, data is fused when traveling to the user interested in these data. The middleware should support such kind of processing.

2.3.2 Relation to OS

Normally, the middleware is designed to run on top of some existing OS, which already provides rich abstractions such as task and memory management. For sensor networks, however, the current OS are topic of active research. Due to resource constraints, the functionalities of the OS are rather limited when compared with traditional OS. [106] argues that one possible option is to give up the separation between OS and middleware and to go towards a distributed operating system that unifies traditional OS and middleware functionality. This is what we are aiming to achieve with our NanoOS proposal. Therefore, our operating system addresses much of the requirements listed in Section 2.3.1.

Due to the fact that the separation of middleware and OS in WSN is subtle and some OS have also middleware functionalities, section 2.2.2 is also analyzing group concerns that are implemented either by OS, VM, or middleware. Hence, the topics presented in that section are also valid here and will not be repeated.

2.3.3 Examples of Middlewares

We are classifying existing middleware proposals in database, task allocation, mobile agents, and events.

2.3.3.1 Database-Based Middlewares

The sensor network is considered a distributed database where SQL-like queries are issued and the network performs a certain task in response to the query.

The middlewares presented in this section are specialized in sensor query processing - they implement algorithms to run queries over sensor networks.

TinyDB TinyDB [92, 132] runs over the TinyOS and supports a single “virtual” database table sensors, where each column of the SQL query corresponds to a specific type of sensor. The query language is a subset of SQL with some extensions.

The SQL query triggers the aggregation of information in the WSN, and the TinyDB supports distributed aggregation of the data. The *Tiny Aggregation* consists of two phases: a distribution phase, in which the aggregate queries are inserted and propagated in the network, and a collection phase, where the aggregate values are routed from children to parents until arriving at the requester. The target of the Tiny Aggregation is to organize the aggregation of results in such a way that the number of messages is minimized.

Cougar Cougar [21, 136] is a loosely-coupled distributed architecture to support both aggregation and more complicated in-network computation. Each node has a *query proxy layer* that is responsible for handling locally the distributed query. A query optimizer is located on the gateway node and generates distributed query processing plans after receiving queries from outside.

The query plan specifies the data flow (between sensors) and the computation plan at each sensor. This plan is disseminated to all relevant sensors. After this dissemination phase, the query can be started.

The Cougar query runs over the so called *sensor database*. *Sensor database* is defined as the combination of pre-stored data (sensor nodes list, sensor node location) with the sensor sampled data.

SINA **S**ensor **I**nformation **N**etworking **A**rchitecture [114] is a middleware that allows sensor applications to issue queries and command tasks into the network and collect the results. Differently from the previous approach, in the SINA middleware, hierarchical clustering is used to facilitate scalable operations. Moreover, an attribute-based naming system is used to support data-centric queries. It is supposed that the nodes are aware about their location in the environment.

The network is conceptually viewed as a collection of datasheets, and each datasheet contains a collection of attributes of each sensor node. Each attribute is referred as a cell, and the collection of datasheets of the network represent the abstraction of an associative spreadsheet. Initially, the datasheet of each node contains a few predefined attributes. During run-time, some nodes may be requested to create new cells by evaluating valid cell construction expressions that may use information from other cells, invoke functions, or aggregate information from other datasheets.

The Sensor Query and Tasking Language is the programming interface between sensor applications and the SINA middleware. It is a procedural scripting language that can contain simple declarative queries.

The Sensor Execution Environment is the part of the middleware that runs in each node and, upon receiving an SQTL message, is responsible for propagating the message further and execute the script inside the message by means of an application (if the message is addressed to the node).

For applications that collect sensor information, the user may choose to invoke the built-in query interpreter instead of writing a procedural SQTL script. This query language is an adaptation of the Structured Query Language (SQL) to serve as primary mechanism for querying sensor nodes.

2.3.3.2 QoS-Based Middlewares

In this class of middleware, the quality of service requirements coming from the application are used in conjunction with the sensor and network information to distribute the tasks among the nodes of the sensor network.

MiLAN The MiLAN middleware [58] stays on top of the network stack and is responsible for linking the application requirements, described as a set of variables of interest (sensor data) and their respective QoS, with the sensor and network architecture.

The applications in Milan are data-driven (i.e. collect and analyze data from the environment) and state-based (i.e. the requirements with respect to the quality of the sampling data may change based on previously received data).

The middleware receives the description of the application requirements. These requirements specify in which variables the application is interested in (e.g. blood pressure in a body surveillance application) and which degree of quality the variables should meet. Moreover, the middleware also has as input the sensors description by means of which quality of measurement each sensor can provide for each of the variables. Based on these data, the middleware can calculate which sets of sensors

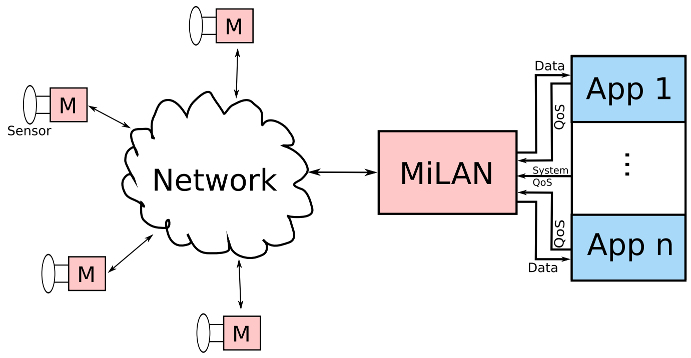


Figure 2.10: System that employs MiLAN. Each sensor runs a scaled-down version of MiLAN. MiLAN receives information about their QoS requirements, a system description of the interaction among applications, and information about the network (e.g. available components and resources). MiLAN, using this information, configures the network to support the application. Source: [58].

satisfy all the application's QoS requirements for each variable. These sets define the *application feasible set* F_A.

Moreover, MiLAN uses a service discovery protocol to learn from the actual network condition. This discovery retrieves information like accessibility of the nodes, energy level, modes of operation, etc. In addition, the roles that the nodes may assume in the network are also observed. The subsets of nodes that can be supported by the network define the so called *network feasible set* F_N. As only sets in F_A provide the required application QoS, the sets are intersected to get an overall set of feasible sets: $F = F_A \cap F_N$.

The MiLAN middleware chooses, among the elements of F, one element f_i that represents the best performance/cost trade-off.

Figure 2.10 presents an overview of a system employing the MiLAN middleware.

2.3.3.3 Event-Based Middlewares

Another approach to sensor network middleware is based on the notion of events [106]. Applications can specify interest in certain state changes of the real word (so called basic events) or certain patterns of events (composed events). Upon detection of such an event, the sensor nodes send notifications to the interested applications.

DSWare The data service middleware [84] has the goal of avoiding the re-implementation of the common data service part of various applications. It resides between the application layer and the network layer and provides data service abstractions.

The most important services are related to event detection. The architecture of the middleware is described in Figure 2.11. Each component of the middleware offers a different kind of service to the application.

In the next paragraphs, we will describe briefly the components of the middleware.

The data-centric storage is a service that provides mechanisms to store information according to its semantics. It has an efficient data look-up and is robust (upon node failures). Correlated data may be stored in geographically adjacent regions to enable possible aggregation. Queries are directed to any of the nodes that contains it. The system tries to avoid collision and to balance the load [120].

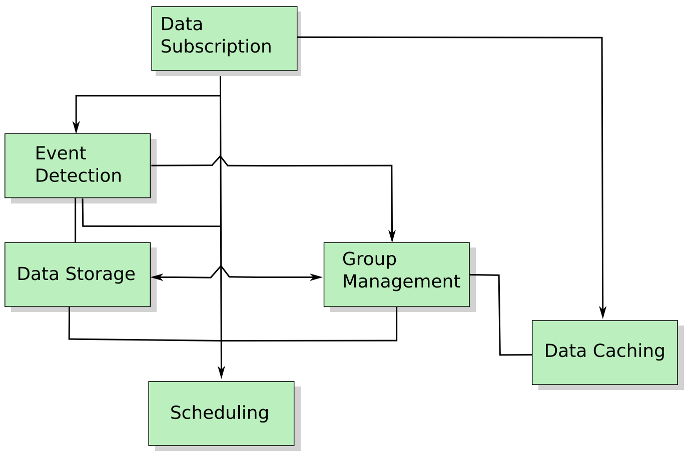


Figure 2.11: Framework of DSWare. Source [84].

The data caching service provides multiple copies of the most requested data. The service is spread over the routing path in order to reduce the communication overhead. It uses a simplified feedback control scheme to decide where to place the copies of data. This means that it monitors the current use of the copies and automatically increases or reduces them in reaction to this use.

The DSWare incorporates a group management component that provides localized cooperation among sensor nodes to accomplish a more global objective. A service responsible for managing nearby group of nodes is important because:

- When a group of nearby nodes agree upon a measured value, this should receive a higher confidence.
- Some tasks may require cooperation of multiple localized nodes.
- When the density of nodes is higher than necessary for the coverage, a subset of nodes may sleep, thus saving energy.

The groups are organized by the group management component based on the queries. When a query is leached, a criteria based on the query is used to select which nodes are part of the group. The group is dissolved when the query expires or the task is accomplished.

The event detection component is the more important component of the system and detects pre-registered target events. The events are classified in two types: the atomic and the compound ones. An atomic event refers to an event that can be determined merely based on the observation of a sensor, whereas a compound event must be inferred from detections of other atomic or compound events. The notion of confidence is used in order to formulate compound events based on degrees of certainty. Events and sub-events have an absolute validity interval associated with them. It depicts the temporal consistency between the environment and its observed measurement. This brings real-time semantics of events.

The data subscription service is a type of data dissemination service. It provides an optimized network configuration when providing data to multiple subscribers in a publish/subscribe interaction paradigm.

Finally, the scheduling component of the DSWare is responsible for scheduling all components of the middleware. A real-time scheduling mechanism is the main scheme used.

Impala Impala [89] is a middleware and an API for sensor application adaptivity and updates. It is a runtime system that acts as an event and device manager for each mobile sensor node.

An event-based programming model is used in the middleware, and actions are performed in response to events.

The Impala middleware is formed by three main components: *application updater*, *application adapter*, and *event filter*.

The application adapter is responsible for making adaptation and responds to a range of events. Impala uses compositional adaptation. Several versions of an application (protocol) are given as input to the middleware. The act of adaptation means selecting the most appropriate one, based on the current context. The adaptation has two objectives. The first one is to increase the performance, energy-efficiency, and other attributes of an application (protocol) by running the most suitable application for an existing environment. The second one is to increase the robustness by selecting applications (protocols) that do not relly on failing hardware. Switching rules are used to change the current running application/protocol.

The application updater is responsible for updating software on the fly in an Impala environment. A modular design model is required for applications. This modular design better supports the update because just local changes within a module must be applied, whereas when a monolithic design is used, small changes may have global repercussions in the code.

The event filter captures and dispatches events to other system units and initiates chains of processing. There are four types of events: timer, packet, send done, and device failure. The applications, application adapter and application updater are programmed as a set of event handlers that are invoked by the event filter when events are received [120].

2.3.3.4 Data Fusion Middlewares

The middleware of this category provides support for data fusion (aggregation) applications. The idea is to support the application's programmer by hiding several concerns like data synchronization, buffer management, and fusion point placement, when programming data fusion applications.

Data aggregation consists of an in-network operation that combines multiple messages coming from different sources in to a smaller representation that is equivalent or in a suitable manner representing the original messages in its content. It captures the redundancy among data collected by different sensors [91]. The data aggregation is often realized when multiple sources are sending data to a common sink. The multiple messages are aggregated in some key nodes when traveling in direction of the sink node.

DFuse The DFuse middleware [78] is an architecture for programming data fusion applications. It supports distributed data fusion with automatic placement and migration of the fusion points. This migration has the goal of maximizing/minimizing some given cost function. This means that the role assignment for each node is decided by the middleware considering the given cost function.

As example, the middleware offers the following cost functions: (1) minimize transmission cost without node power considerations, (2) minimize power variance, (3) minimize the ratio of transmission cost to power, (4) minimize transmission cost with node power considerations.

Using the DFuse middleware, the developer is only responsible for implementing the fusion functions and providing a data flow graph [120]. The distribution of this fusion points is made automatically.

Moreover, DFuse provides a Fusion API that affords the easy development of complex sensor fusion applications. The API allows custom synthesis operations on streaming data to be specified as a fusion function, ranging from simple to complex operations [78].

The fusion operation is defined very broadly in the DFuse middleware: it is the application of an arbitrary transformation to a correlated set of inputs, producing a combined output. The middleware is targeted to streaming fusion processes. The transformation may produce a smaller output than each single input (called contraction), a larger output (called expansion), or keep the *status quo* of the data rate flow. The identification of the kind of transformation is important in order to find the appropriate placement of each fusion point.

The following capabilities are provided by the fusion API [78]:

Structure management: This category offers "plumbing" capabilities. An abstraction called fusion channel is offered by DFuse to the user in order to realize the fusion tasks. The programmer provides the fusion function, and this fusion will be performed either on request or when input data are available.

Correlation control: Responsible for handling specification and collection of "correlation sets" (related input items that should be supplied to the fusion function). Fusion requires identification of a set of correlated input items.

Computation management: Handles the specification, application, and migration of fusion functions.

Memory management: Handles the caching, prefetching, and buffer management.

Failure/latency handling: Responsible for the capability of fusion points to perform partial fusion, i.e., fusion on incomplete input correlation sets. It deals with sensor failures and communication latency.

Status and feedback handling: Responsible for the communication between fusion functions and data sources (e.g. sensor nodes).

2.4 Virtual Machine

The virtual machine approaches offer hardware platform independence and code expressiveness for compactness of the code. This compactness enables the migration of the code among nodes with small overhead when compared to binary code.

The main objective of the virtual machine approaches is to obtain dynamic re-programmability at a reduced cost. For that, they assume that a sensor network system is composed by a common set of services and sub-systems, combined in different ways. The language interpreted by the virtual machine allows the composition of services and sub-systems to be described concisely. This has an advantage when compared to the transmission of raw binary code.

Moreover, it provides a programming model powerful enough to implement any distributed system while, at the same time, hiding unnecessary low-level details from the application's programmer [22].

Some authors [120, 106] classify virtual machines as a middleware approach, whereas others [79] use a special category for them. We separate the virtual machines as a specific category of system software for sensor network because, differently from the other middlewares, they do not only provide abstractions for communication or organization of the system, but also an execution environment independent from the underlying processor. This is an important difference from the middleware approaches. Nevertheless, much of the requirements described for middleware approaches are also valid for this class of system software.

2.4.1 Examples of Virtual Machines

Maté The Maté [82] virtual machine enables a wide range of sensor network applications to be composed based on a small set of high level primitives. It is a byte-code interpreter that runs on top of TinyOS and has a stack-based architecture.

Maté is implemented as a single TinyOS component that interprets small pieces of code called *capsules*. Each capsule has 24 instructions of one byte. Larger programs may be composed by several capsules. Each capsule fits in a single network package.

There are four types of capsules: message send, message receive, timer, and subroutine. Every capsule includes type and version information. The subroutine capsules form parts of the program that may be called from the other types of capsules. The message send, receive, and timer capsules are the routines executed as response to the events send packet, receive packet, and timer, respectively. The virtual machine has three execution contexts, for the three types of events. The execution of the capsules is always triggered by some event.

In the interpreted language of Maté, there is a command to send the capsules to all neighbors. Upon receiving a capsule, each node tests whether the version is more recent than the current installed one. If positive, the new version is automatically installed. Using this mechanism, it is possible to easily distribute new versions of an existing application as well as new applications. In the paper, the dissemination of code is compared to a virus. It makes possible for a user to enter a query (or other type of application) in a single point of the network, and this application will propagate itself until "infecting" all nodes.

Therefore, the Maté virtual machine provides a flexible way of programming a sensor network, having a small system requirement and providing an efficient way of WSN dynamic programming.

SensorWare SensorWare [22] is a virtual machine for distributed applications running on a sensor network. Differently from Maté it targets richer platforms, requiring more resources than it. The authors argue that such resource-richer platforms will be mainstream in an immediate future.

The idea of Sensor Ware is to avoid the assembly-like programming of Maté by means of using scripts of a high level interpreted language. The selected language for the implementation was *Tcl*. A set of function/commands was defined besides the basic script interpreter. This set is composed of the following APIs: radio, timer sensor, and mobility. These functions form the basic building blocks that are combined through the *Tcl* scripts. The scripts orchestrate the dataflow to assemble custom protocols and signal processing stacks.

The programming model of SensorWare resembles state machines that are influenced by external events (e.g. network message arrival, sensing data, expiration of timers). The idea is that an event is processed by a light event handler that performs its processing according to the actual state. This processing possibly generates new events or/and changes the current state.

Figure 2.12 presents the general sensor node architecture of SensorWare. The existence of a script in the network starts with the injection of the code made by some external user. After this initial injection, migration is responsible for the code dissemination. The scripts are interpreted by the SensorWare layer, and API function calls are redirected to the underlying OS. Moreover, as the figure depicts, scripts may divide the user space with other kinds of applications or services (native code).

Similarly to the Maté virtual machine, SensorWare provides a flexible way of dynamic programming of sensor networks, but, differently from Maté, it avoids the complexity of a low-level assembly-like language.

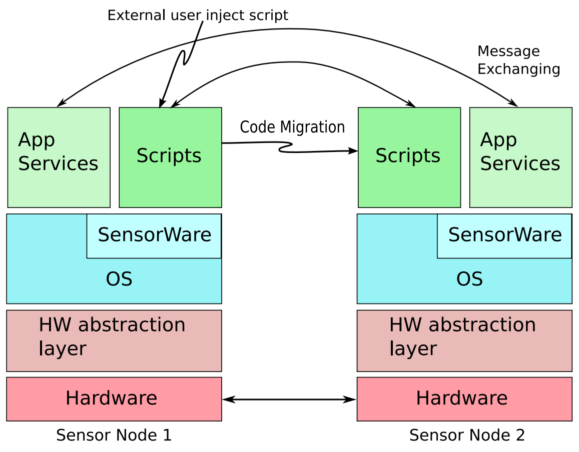


Figure 2.12: The sensor node architecture with SensorWare virtual machine. Source: [22]

MagnetOS The MagnetOS [88] is a virtual machine that provides a programming model where applications do not need to implement by themselves all required mechanisms to deal with the dynamics of ad hoc networks. The authors argue that mechanisms for remote communication, naming, and migration are very important in such environments. Moreover, the applications must be supported in order to deal with the dynamic and resource-constrained of the system. Finally, it is important to support facilities that enable the dynamic introduction of new functionality and its integration with the current running application.

In order to address the described requirements, the MagnetOS supports an alternative programming model, where a thin distributed virtual machine[1] makes the entire network appear to applications as a single Java virtual machine.

The higher level of abstraction provided by the MagnetOS simplifies the development of applications and enables the automatic placement and migration of modules of the application, thus saving energy.

The MagnetOS applications are designed as a set of interconnected, mobile event handlers (implemented as objects). The communication between the objects is made using events. The distribution of the event handlers is made automatically by the MagnetOS with the objective of spending the minimal energy for communications.

The system consists of a static application partitioning service that resides on hosts capable of injecting new code into the network, a run-time module on each node that performs dynamic monitoring and component migration, and a set of policies to guide object placement at run-time. The static partitioning service rewrites applications intended for a single JVM into objects (event handlers) that can be distributed over the network. The migration in MagnetOS relies on profiling the communication pattern of objects in discrete, asynchronous epochs. The NetPull and NetCenter [9] algorithms are used to migrate the objects of the application.

2.5 Discussion

In this chapter, we presented a survey of several OS, middleware, and virtual machine approaches for sensor networks. In the layer nearest to the sensor node hardware lays the operating system.

[1]In the paper they are using the term operating system (OS) to the MagnetOS. However, despite the name, it isn't an OS.

The general goal of the presented OS is to offer useful abstractions and extend node's functionality. At the same time, they are designed to consume as few resources as possible of the target platform and to manage the power consumption. A key resource is memory. Systems like TinyOS and Yatos support an event programming model which considerably reduces the memory requirement due to the fact that just one global stack is necessary for the entire system. Nevertheless, this comes at a high price: applications need to be developed as state machines without preemption. The challenge here is how to efficiently use the power of the state machine programming model without getting lost in the complexity of different state machines sending messages to each other. The event-based paradigm is adequate for pure sensoring applications where nodes should react upon events occurring in the physical environment (sense and forward paradigm).

Other OS provide a classical preemptive multi-threading programming environment trying to reduce the footprint by means of providing just a basic subset of capabilities. An example of this class is the MantisOS, which is a configurable OS with a programming environment similar to UNIX. The dynamic re-configuration provides the functionality of replacing and loading services, device drivers, and application tasks during run-time, avoiding the overhead of having the complete code in the memory all the time. Here we can highlight PeerOS which allows this dynamic reconfiguration.

There is also an attempt to combine event-driven and preemptive multitasking environments in the Contiki OS, which is composed by a kernel and a reconfigurable set of services. The services comprise, for example, a communication protocol stack, device drivers, or high level functionality and can be loaded dynamically. The main kernel of Contiki is event-driven, however, optional library modules offer preemptive multitasking functionality. This means that Contiki is even more flexible than the other approaches, and, due to its high adaptability to the target applications, the footprint can be reduced.

Nonetheless, all approaches have a common drawback: in order to reduce the footprint, the given functionality is also restricted. In the event-based systems, the comfort of preemptive multitasking is given away for the benefit of small footprint. For classical sensoring applications, this makes sense, because they can be naturally modeled as a finite state machine. Nevertheless, when (background) processing or complex distributed applications are desired, the paradigm has its disadvantages.

The preemptive multi-threading paradigm uses the dynamic reconfiguration for handling the lack of resources and to dynamically adapt itself to the application. However, at a given moment, the amount of functionality present in the system (OS plus application) is very limited. Moreover, an additional shortcoming of most of the OS is the lack of support of high level functionality, leaving this to the middleware level. The development of distributed sensing tasks or signal processing algorithms is not supported by the OS. Even very common sensor network tasks, like data fusion, have no high-level support from the OS. A better distributed programming model is offered only by Bertha OS, which also supports self-organization. Nonetheless, all OS include some abstraction for external communication.

A system that presents a completely new programming paradigm and distributes the applications is BerthaOS. The process fragments (PFrag) are the algorithmic unit of the application and may interact with neighboring PFrag in order to generate some emerging desired result. Moreover, migration (initiated by the PFrag) is allowed. This approach has a very good perspective. Nevertheless, it requires a completely new programming model, and it is rather difficult to design distributed algorithms.

The VMs are situated in the middle between OS and middleware: they provide rather higher level abstractions for distributed processing than the ones typically offered by OS. Nevertheless, on average, they offer less then a middleware. The drawbacks of the existing VMs for sensor networks are, in our opinion, mainly two. First, the majority of VMs provide an unusual programming model, where the programmer must deal with questions like code replication, migration, etc. Second, they don't provide the usual abstractions for development of distributed processing applications. They are based on the assumption that the sensor network will be used merely for sensor queries. Our proposal tries to bring again complex processing capabilities to the sensor nodes. We aim at making the sensor networks more autonomous in the sense that they may react to environmental phenomena without interaction with a base station.

The only VM that does not have the described drawbacks is MagnetOS. The user, in this case, does not need to worry about code migration and could create in a relatively easily way complex applications. Nevertheless, MagnetOS is a Java virtual machine and brings a large overhead with it. Moreover, typical sensor network applications like data fusion are not implemented in a adequate manner within the MagnetOS framework.

The middleware solutions present myriads of different approaches for application development for wireless sensor networks. Middlewares based on databases are very efficient on processing sampling queries from the user. Based on the query input, the processing on each node is automatically determined as well as the data aggregation points. Some of the database middlewares have a centralized decision of which role each node should assume (e.g. Cougar), whereas other ones have distributed decisions (TinyDB), which increase the scalability. The main drawback of the database middlewares is the lack of flexibility: they are mainly designed to process SQL-like queries. Expressing other kinds of applications or even more complex queries may not be possible.

QoS-based middlewares, like MiLAN, can automatically select nodes that meet QoS requirements defined by the user. It is used for collecting data applications. The middleware, opposed to our approach in NanoOS, has a centralized element that decides which nodes should be queried and is just specialized on collecting data applications. There is no support to distributed processing in the network.

Another group of middlewares that is restricted to some specific type of applications is the group of event-based middlewares. In such a middleware, the user can specify a set of events that should be notified. The Impala middleware allows dynamic updates and application adaptation besides the basic event-based processing. These are very interesting features but do not overcome the main limitation of this class of middleware: the state-machine based programming model and the absence of methods for functionality distribution, like in our OS.

Finally, data fusion middlewares are designed also for a very specific task in a sensor network, without the possibility of supporting other types of in-network processing besides data fusion. Moreover, they do not allow the dynamic re-assignment of the functionality of the middleware/application in response to topology/energy changes, like the features present in our NanoOS.

Due to the fact that a focus of this thesis is the distributed processing in the WSN, we are presenting in Table 2.5 some examples of distributed processing mechanisms present in the different approaches. Some task allocation mechanisms showed on this table will be discussed in Section 4.2.2.

System Name	Service Discovery	Task Allocation	Remote Communication	Task Migration
		OS Architectures		
TinyOS	Not supported	Not supported	Active messages	Not supported
Bertha OS	Not supported	Not supported	Bulletin Board System	Binary Code
MOS	Not supported	Not supported	Not supported	Binary code download
Yatos	Not supported	Not supported	Messages	Not supported
		VM-based Architectures		
Sensorware	Not supported	Script population specification	Not supported	TCL script migration
MagnetOS	Not supported	Automatic object placement	DVM	Mobile Java objects
Mate	Not supported	Not supported	Not supported	Code capsule update
		Middleware architectures		
TinyDB	Not supported	Query optimizer	Not supported	Not supported
SINA	Not supported	Attribute matching	Not supported	SQTL scripts
Cougar	Not supported	Query optimizer	Not supported	Not supported
MiLAN	SLP, Bluetooth, SDP	Configuration Adaptation	Not supported	Not supported
DFuse	Not supported	Automatic fusion point placement	Not supported	Fusion point migration

Table 2.2: Distribution features of selected approaches. Sources: [79], [78]

Chapter 3

NanoOS Architecture

In this chapter, the basic architecture of our operating system for sensor networks is presented. Moreover, a model that represents the link quality in wireless ad hoc networks is also introduced.

3.1 Motivation

In the Section 2.5 of the last chapter, a discussion about the existing OS, VM, and Middlewares for wireless sensor networks was presented. The different capabilities, programming models, and application scenarios of the approaches were discussed. The development of the NanoOS operating system has the intention of fulfilling the following gaps of the existing systems:

- Lack of support for generic, complex distributed in-network processing. Several approaches in the middleware area support certain type of in-network processing (e.g. DFuse for data fusion), but there is almost no system with a generic programming model where different kinds of services implementing distributed processing can be easily and systematically developed. The NanoOS supports the development of typical WSN services like data aggregation as well as processing intensive services like distributed fourier transform or data encryption service. Even OS abstractions like file system may be implemented as mobile services. This brings the next drawback of the existing approaches:

- Impossibility of complex OS functionality in constrained nodes. Our idea is to overcome this limitation through distributing the application and also the OS functionality among the nodes of the system. Whenever there are enough resources for the complete OS and application on every single node, the system can behave as a normal multi-threading OS and all the services may be started in the single node. But in situations where the applications require more services that can fit on the resources physically presented in a single node, the services will be distributed to external nodes and accessed remotely.

- Insufficient support of client-server programming model. Several approaches present alternative programming models due to different reasons. They may be more suitable for the type of application envisioned. For example, BerthaOS or the virtual machines offer a programming environment where small code fragments can replicate, migrate, and thus disseminate themselves through the network. This may be a very useful programming model for arbitrary sensor data queries that should be spread through the network and collect results. Other possible reason is the hardware limitation: the event-based systems of TinyOS and Yatos are very suitable for heavily constrained nodes because they demand just one global stack. They are suitable for the type of application they are envisioning. But besides the Java-based MagnetOS, there isn't a systematic approach to support the client-server programming model with automatic distribution of modules. This allows an easy development of distributed sensor applications.

- Absence of a generic, dynamic, distributed, and automatic code migration mechanism for client-server paradigm. Some of the present systems are able to calculate a suitable placement of some system components (e.g. DFuse middleware has automatic placement of fusion points, Tiny Aggregation, which selects the suitable points for data aggregation). Nevertheless, these placement mechanisms are developed specifically for the kind of processing done by the middleware. In our case, our service distribution algorithm can distribute any kind of service based on a dynamic assessment of the current traffic and network topology.

Hence, our proposal is an integrated approach, where OS basic functionality is integrated with high-level support for distributed applications on sensor networks. The same mechanisms offered to manage the services at the application level are also used by the OS services. With the support of complex distributed processing, we aim at making a WSN more autonomous from the base station, allowing new challenges to be mastered by sensor networks in environments where there isn't continuous contact with the base station. Our programming model has costs: the programmer has to design the application and partition it manually. Moreover, due to the distribution, the energy overhead increases considerably. In addition, we need id-based networking instead of merely data-driven protocols for the realization of our operating system. We argue that a combination of both paradigms is the best solution in order to deal with the conventional problems of distributed processing (distributed algorithms, more node-driven) with the new challenges brought by canonical problems of WSN (data-driven problems, like queries).

3.2 System Overview

In this section, a small overview of our complete system will be described. This overview is necessary to a better understanding of the details presented in the following sections.

Our system is composed of three main components: the hardware, the operating system, and the application running on top of it (see Figure 3.1).

The hardware platform consists of a set of distributed sensor nodes. On top of this platform, runs the *NanoOS* that provides the adequate set of services to the application. Besides the basic operating system services like processor and memory management and synchronization, the NanoOS provides a set of special services to support the distributed processing.

One or more applications run on top of the operating system. Each applications has one goal and is composed by a set of fixed tasks and application services; they are the atomic unit of the distributed application. This means that their program code must not exceed the resource availability of one single node. This will be in-depth discussed in further sections.

For the purpose of reducing the OS footprint in each node and therefore to enable the execution of a rather complex OS application in hardware constrained nodes, the NanoOS uses a novel approach: it distributes the services of the OS and application among the nodes. This means that each node of the system has just a small part of the kernel of the complete operating system and some modules of the distributed application; a group of nodes together forms an instance of the OS with one or more applications. The services are shared among tasks sitting on different nodes. At any instant of time,

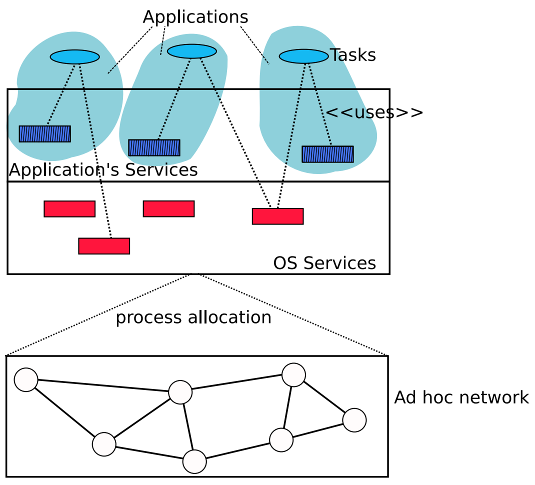


Figure 3.1: System overview

one node may connect and use a service residing in an external node using a remote method invocation (RMI).

In order to facilitate the development of applications, the OS provides a uniform service interface where the OS and application services may be requested even if they are not present at the current node and must be accessed from remote nodes. This service interface must guarantee the access to services in a dynamic topology environment.

Figure 3.1 presents an overview of the system. The tasks may use services of the operating system and other services made available by the applications themselves. In order to reduce the resource requirement in each node, those services are shared among different application tasks and it is possible that services are executed in remote nodes.

We will discuss the OS architecture more deeply in the following sections.

3.2.1 Applications Scenario

Our service distribution architecture is designed to support WSN scenarios where complex processing is required. We envision WSN applications that are more than the traditional sense-send-sleep loop. Our target are applications that need complex, in-network processing components and sophisticate OS support. For example, distributed collaborative signal processing and distributed compression may be implemented as mobile services in our platform.

Besides the provision of a set of OS service and a framework that enables the developer to design its own services, the advantage of the architecture is the automatic management of these services. This means that even standard WSN services like data aggregation and data compression based on correlation among different measurements (spatial and temporal) can be provided in an standard way by the OS and are automatically placed on the network.

Due to the possibility of increasing the OS and application complexity using distributed shared services, new applications scenarios can be considered. For example, it is possible to increase the autonomy of the sensor network and its interaction with the environment. We can view of the network as a kind of environment-embedded distributed computer that may sense the environment, make complex calculations and decisions about the sample data, and, if actuators are available, react upon detected events.

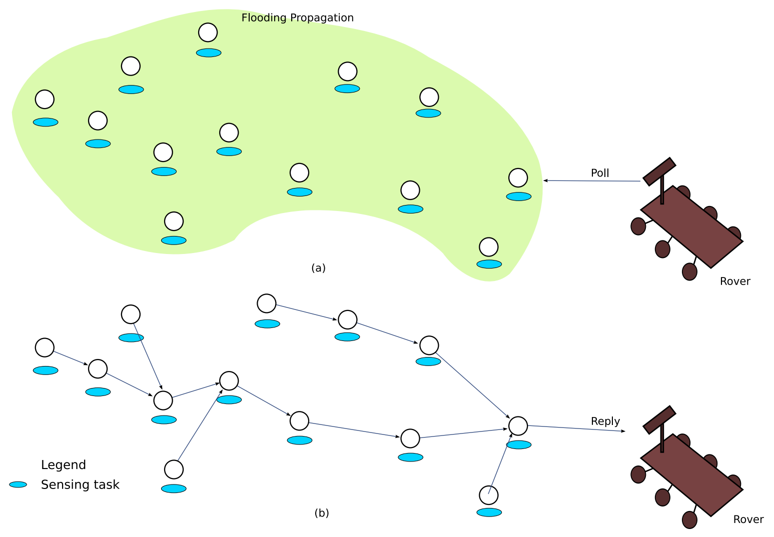


Figure 3.2: Space exploration application using WSN. (a) The polling phase where the rover requests the sampled data using a flooding mechanism. (b) Every node reply the request and the data is routed back to the rover.

In the next sections, we will present two existing WSN applications and the possibilities opened by our operating system.

3.2.1.1 Scientific Exploration

Self-organizing sensor networks will probably play a key role in space exploration in the future. An example of an envisioned application of sensor networks on the exploration of the surface of Mars is presented in [64].

The types of data sensed by the nodes are, for example, seismic, chemicals, temperature, etc. One or more landers or rovers function as base station and periodically collect the measurements and relay the aggregated sensor field results to Earth.

In the application, the region covered by the sensor nodes is large, so that multi-hop relay is used. A poll-reply communication model is used to collect the data. Namely, each base station periodically broadcasts a polling request and every sensor node returns the collected data to the base station. This fact is illustrated in Figure 3.2.

We envision, with NanoOS, a solution where the sensor network has more autonomy. In our hypothetical scenario, several disconnected regions of Mars receive a large pool of sensor nodes. They are autonomous in the sense that the data are just requested by the rover (base station) in large intervals. Just at this point the rover may send them to Earth. Instead of the intensive polling, the sensors should process the collected data by themselves and save the results in some flash memory (designed to save intermediate results). The rover in this scenario may approach each disconnected group of nodes at some sparsely distributed time to gather the results.

The idea of saving raw data in the nodes is impracticable due to memory restrictions. Therefore, the data should be analyzed by the network itself and the results (the detection of some environmental characteristic, for example) should be saved in a compact form and sent to the rover upon request. For that, the architecture presented in Figure 3.3 was proposed.

In the figure, the sampling tasks send their read data to services that are responsible for analyz-

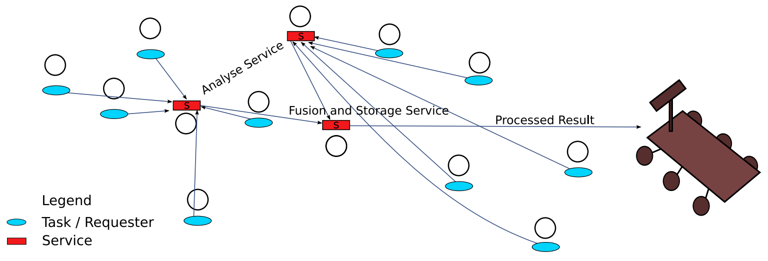


Figure 3.3: Autonomous network for space exploration. Instead of polling the area in an intensive way, the rover just needs to approach the WSN in sporadic intervals to download some selected compressed data and the analyzed result. The processing and compression of the data is done by the services in the sensor network.

ing the data and detect desired patterns. This signal processing services may be also distributed, and are placed in the network automatically, aiming to reduce the communication overhead and the energy consumption. The results are sent to a data fusion service that stores the results until the rover approaches the network and polls the service.

A similar idea may be used in a scenario developed in the scope of the EU project e-Cubes [1]. The idea is to make atmosphere assessment of Mars by means of a sensor network that is dropped in the atmosphere. During falling down, they should collect data. After arriving an the surface of the planet, a rover is responsible for receiving the measured data and sent to Earth. In this case, the WSN must be autonomous during the period of time it is measuring. Moreover, it is important that the nodes of the network remain alive during this phase, after that, the energy of the nodes is not that important anymore. This means that our approach of exchanging energy for processing complexity may represent a very attractive trade-off.

3.2.1.2 Habitat Monitoring

In this section, we will present an example how our architecture can be used to improve an existing application on habitat monitoring.

In the work [127], a two-tiered sensor network is proposed for habitat monitoring. The main target of the system is to recognize and localize specific types of birdcalls. To specify the birdcalls of interest on the system input, the biologists typically have recorded birdcall waveforms.

The idea of using a two-tiered hardware platform for habitat monitoring was inspired in the work of Cerpa et al [27]. The smaller, less capable nodes are used to exploit spatial diversity, whereas the more powerful nodes combine and process the micro-node sensing data.

The organization of the network, as presented in the work [128], can be seen in Figure 3.4. The macro-nodes are very powerful nodes with Pentium II CPU, up to 64MB RAM, and a full spectrum of peripheral devices. Micro-nodes are Berkeley motes with 128KB program memory, 4KB data memory, and 512KB secondary storage. Both nodes are equipped with acoustic sensor.

The micro-nodes are densely distributed, whereas macro-nodes are sparsely due to their higher power consumption and cost. The nodes form clusters with macro-nodes as clusterheads. GPS on macro-nodes provides location and time reference. Location of other nodes can be determined interactively.

Given the network, the macro-nodes receive the waveform of the birds calls and convert it to an internal format used by the recognition. Spectrograms are complete descriptions of bio-acoustic characteristics of birdcalls and are widely used. Macro-nodes have enough computational resources

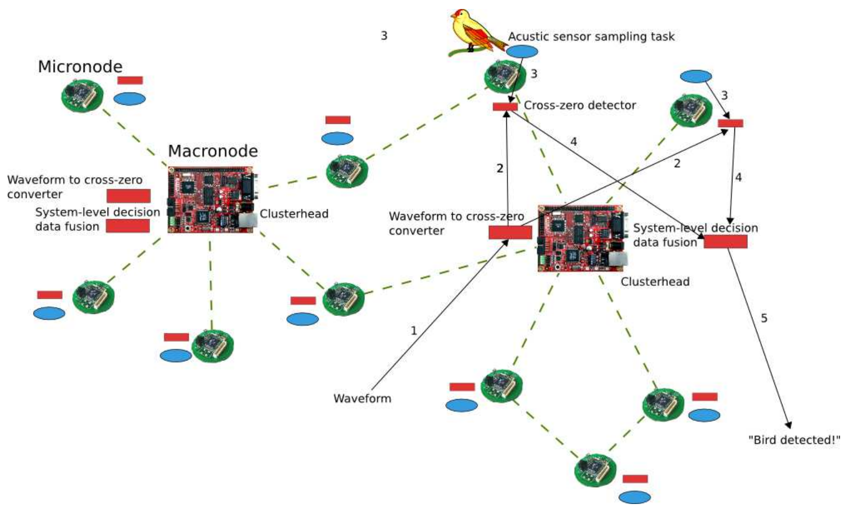

3
Acustic sensor sampling task
Micronode
3
Cross-zero detector
3
4
Macronode
2
2
4
Waveform to cross-zero converter
System-level decision data fusion
Clusterhead
Waveform to cross-zero converter
System-level decision data fusion
Clusterhead
5
1
Waveform
"Bird detected!"

Figure 3.4: Two-tiered sensor network for bird monitoring. Macro-nodes are PC 104s. Micro-nodes are Berkeley modes. Source: [127]. The figure also shows the task decomposition of the target bird recognition. (1) Waveforms of target birds are sent to macro-nodes. A service calculates the cross-zero rate representation for the micro-nodes. (2) The representation is sent to the micro-nodes. (3) The task samples the acoustic sensor and sends the data to be compared with the cross-zero representation. (4) After being recognized by the cross-zero target detector service, this information is sent to the macro-node where a system level decision (data fusion) will be done. (5) Finally, a system decision of the detection is sent to the gateway.

to use spectrograms internally, however, micro-nodes not.

Therefore, a simpler internal representation using cross-zero rates is used in the micro-nodes. The target recognition task is divided in two steps. First, all nodes independently determine whether their acoustics signals are of the specified type of birdcall using a pre-processing method with cross-zero rate. After this, macro-nodes fuse all individual decisions into a more reliable, system-level, decision. The details of the decision fusion were not discussed in the publication. In Figure 3.4, the target recognition task has its components depicted.

The target location task was also divided into two steps. First, waveforms are recorded at nodes distributed at different locations. Second, the data are accumulated on one macro-node, and beam-forming is applied to determine the target location using arrival time differences. Before sending the data to the macro-node, each micro-node simply compress it. The target localization task can be seen in Figure 3.5.

The presented tiered sensor network approach for habitat monitoring has some disadvantages:

- There isn't a self-organization mechanism in the task distribution between macro-nodes and micro-nodes.
- The deployment must be carefully executed, random deployment is not possible.
- If some macro-node experiences failure, there isn't a way of substituting it by nearby nodes.
- The data fusion isn't done at the optimal position (regarding communication energy usage).
- Cross-zero rate method looses some information from the spectrogram. With noise in the environment, the distorted cross-zero rate is not enough to detect the birdcall. A frequency filter

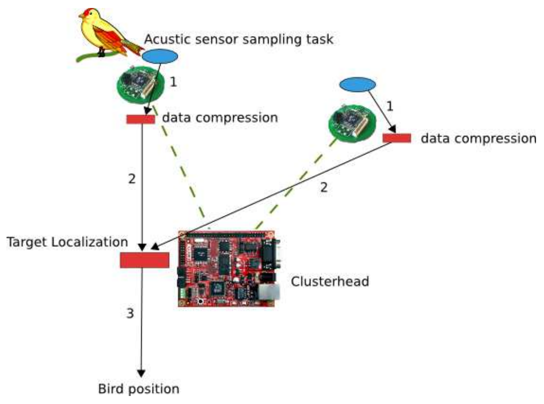


Figure 3.5: Task decomposition of the target localization task. (1) The task samples the acoustic sensor and, after the recognition of the desired bird species, data are sent to the compression service. (2) After being compressed, the data are sent to the target localization task in the macro-node. Using beamforming estimation, the position of the target bird is calculated. (3) The resulting bird position is sent to the gateway (access point).

may help, although some target birdcalls may be discarded even with filters due to environment noise.

Using the NanoOS approach, we can design a bird monitoring system as following.

The target recognition task is shown in Figure 3.6. Every service present in the original architecture is also present in our proposal. Nonetheless, instead of heaving a large service responsible for one complete task, several distributed sub-services are used, due to the hardware limitations of each single node. In our case, the waveform is divided in blocks (chunks) and each block is processed by one service. For example, if the habitat monitoring would use visual data (cameras), each service would be responsible for extracting one type of feature of the image. The birdcall detector sub-services are shared among two birdcall detection services. Each sub-service has a spectrogram of the partial waveform. If the services would be duplicated, this same waveform would need to be stored in multiple services, increasing the memory consumption.

An advantage of our architecture is the automatic placement of the services. For example, the data fusion (system-level detection) service may migrate to any node with enough resources to execute it. Our placement algorithm (described in the next chapter) aims at finding the optimal position of each service in the system. If the system is heterogeneous (with macro-nodes), a large bunch of services will be automatically placed on those macro-nodes. Instead of carefully designed, the task placement of the system (like in the original paper) should be done in a self-organized fashion.

The second task of the system (target localization) is shown in Figure 3.7. Here, the architecture is similar to the original one, nevertheless the services may be placed at appropriate positions and the target location service uses external supporting sub-services when necessary.

3.3 Requirements

In this section, we will present some requirements of the NanoOS.

Geographically distributed services The services should be distributed among the geographically distributed sensor nodes. Moreover, they should be shared among the application's tasks and

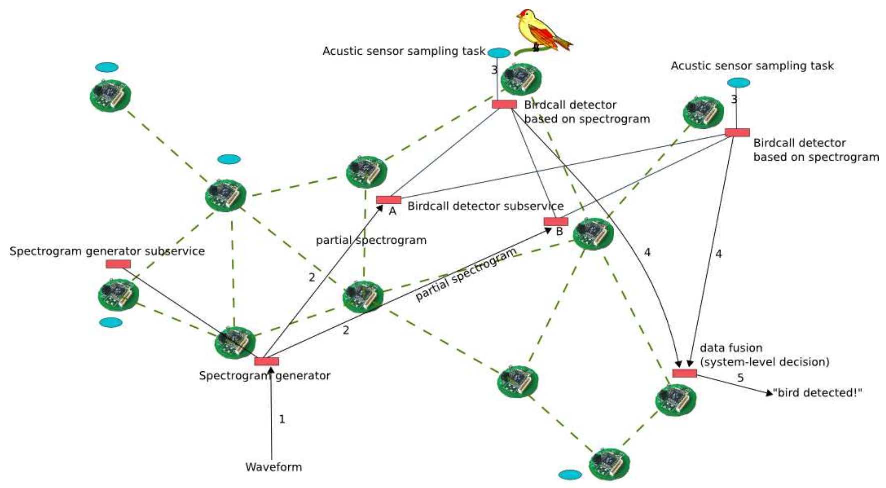

Acustic sensor sampling task
Acustic sensor sampling task
Birdcall detector based on spectrogram
Birdcall detector based on spectrogram
Birdcall detector subservice
A
B
Spectrogram generator subservice
partial spectrogram
partial spectrogram
Spectrogram generator
Waveform
data fusion (system-level decision)
"bird detected!"
1
2
2
3
3
4
4
5

Figure 3.6: Target recognition task using the NanoOS approach. (1) Waveform is sent to the spectrogram generator service. It calculates the spectrogram of chunks of the original waveform with help of sub-services. (2) Spectrogram of a part (chunk) of the original waveform is sent to the corresponding detection sub-service. (3) Sampling task sends data to the detection service. The detection is made using sub-services for every waveform chunk. (4) Detection is sent to a data fusion service. (5) The detection of the target bird is announced.

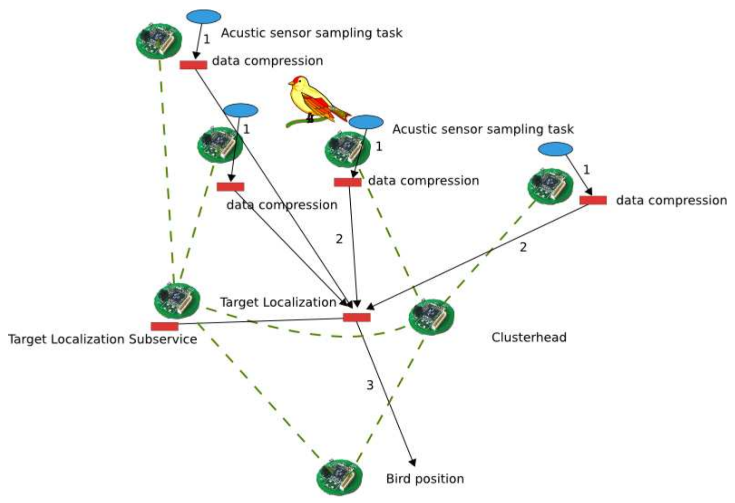


Figure 3.7: Target location task using the NanoOS approach. (1) Waveform is sent to the data compression service. (2) After this, the compressed data are sent to the target localization service. The target is localized using beamforming procedure and with help of distributed supporting services. (3) The bird position is the outcome of this procedure.

other services.

Uniform environment of execution It should be possible, for tasks and other services, to access service providers locally and remotely. In order to simplify the development of applications, uniform environment of execution is desired, i.e., the tasks may assume that a given set of services is available independently from external factors like node position. These services are accessed using a unified service interface.

Low local resource utilization In the situation where the resource requirements of the application and OS are larger than the availability of a single node, the services should be distributed in order to comply with the hardware constraints.

Ad hoc networking The communication takes place using the wireless links present in the sensor nodes. The network should be self-organizing. No globally centralized control is desired.

Dynamic adaptation Due to the high dynamics of the system (due to topology changes), dynamic adaptability of the OS to the current network configuration and to the requirements of the applications is desired. This adaptability is achieved using:

- *Compositional dynamic adaptation* [93]: algorithmic or structural system components may be exchanged in order to improve the OS performance in the current environment. The environment comprises task requests and network state. This brings also the possibility to tailor the services to the requirements of the application, avoiding unnecessary overhead.
- *Service redistribution*: We use the redistribution of the services in the network to react to topology changes and request pattern changes. This redistribution aims at reducing the communication cost (reflecting reduced energy expense). In this thesis, we will concentrate on this topic.

Self-organization After deployment of the WSN in the target environment, the organization of the OS and application should be managed without any central entity. The system should organize itself by means of launching the appropriate software components and connecting them appropriately. An example of mechanism used to connect the requesters of services with the corresponding provider is the service discovery.

Self-optimization This requirement is linked with the dynamic adaptation. We aim at improving the system performance during run-time, changing system parameters, exchanging components or changing the placement of the objects in the system. We will focus on this last method aiming at reducing energy consumption.

Self-healing Several errors may happen during the system life: node's hardware failures, software failures, and topology changes. The connection pattern (topology) in an ad hoc network isn't stable. Nodes can suffer of transitory (or even permanent) disconnection. This means that we should design the OS in a way that make it robust against such failures. We are tackling this challenge by means of two approaches:

- Designing algorithms that are robust by nature: We are using self-organization principles in our proposals. One motivation for that is that they are robust against unpredicted situation and partial failures.

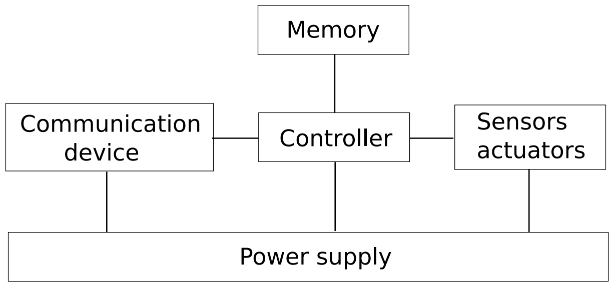


Figure 3.8: Generic sensor node architecture. Source: [69].

- Using standard fault-tolerance techniques: In order to prevent interruption of service due to hardware or software failures, services backups (replicas) may be used. This is out of scope of this thesis and requests further development.

Transparency The following types of transparency are supported in the NanoOS: *access* (e.g to a service, whether it is local or remote), *location* (the set of available services is the same at any place), *topology* (topology changes mostly do not affect the execution of the tasks). The *failure* transparency should be further developed.

Scalability Our system should scale to thousands of nodes. Due to the fact that the application scenario assumes large areas of sensing, it may be necessary to use a huge number of nodes. Therefore, we cannot rely on a central controller for organizing our network.

In this thesis, we focus on some of the requirements of the NanoOS. In fact, we are mainly tackling the OS self-optimization requirement using our service distribution heuristic and the self-organization of the network through our clustering algorithm. For the developed algorithms, the scalability goal is also observed.

3.4 NanoOS Approach

In this section we will present an overview of our complete system architecture. Subsequently, we will present with more details the organization of our sensor operating system for distributed applications.

3.4.1 Hardware Platform

The hardware platform consists of a set of distributed sensor nodes. Each node comprises a communication device, a small processing unit, small memory, sensors, and actuators. A generic architecture of a sensor node is presented in Figure 3.8.

We are considering typical sensor networks with the following characteristics:

Processing unit: We are considering nodes with a microcontroler as processing unit.

Memory: As usual, two types of memory are installed in each node. Volatile memory (RAM) is used to store intermediate sensor readings, packets, and processing states. The programs are stored in non-volatile memory and are executed there. Normally, the non-volatile memory is larger than the volatile one. We assume that the complete OS/application code is stored in the non-volatile memory. This memory is normally a Flash. Flash memories have long read and write access delays and require more energy than the RAM.

Although we will not deal with different architectures in this work, the NanoOS could be also adapted to other possible types of nodes: nodes where the code is loaded to the RAM before executing or nodes where the non-volatile memory isn't enough to store the complete OS/Application. In this case, the migration of a service (that will be described latter) demands the transfer of both state and code.

Communication Device: We are considering, in our work, that the devices have wireless communication based on radio frequency. Devices with high energy efficiency are desired. Moreover, we assume that the data rate of the device is chosen in order to cope with the communication needs of our distributed OS plus application. We are also assuming that there is no power control in the communication devices, i.e., they use always the same power to transmit. The range of the device is specified based on the application's requirements.

Sensor and actuators: Depends just upon application requirements. For the OS, any set of sensors or actuators may be used.

Power Supply: This is a crucial component in our system. Due to the fact that we are playing, in our approach, with the trade-off between energy consumption and complexity of the OS/application, an efficient and high capacity battery is supposed. Moreover, due to the relatively high energy consumption brought by the OS/Application distribution, a good energy scavenging system may be required.

3.4.1.1 Models for Reconfiguration

For the purpose of supporting the dynamic adaptation realized through the heuristics presented in the following chapters, the load of the different hardware modules has to be exposed permanently to the OS. Besides the load of the hardware modules, the communication pattern is another important variable in the introspection of the OS. The following models are necessary in order to support the introspection process:

Hardware Load Model This model is responsible for the characterization of the load in the different hardware resources. Normally, such a module captures the *processor utilization* and the *amount of used (and free) memory*. Because our target are tiny sensor nodes, the *available energy* and *actual power consumption* are important information that should be also in the hardware load model.

This relevant information for the reconfiguration process is extracted from the current state of the hardware and is exposed to the heuristics presented in the next chapters.

Communication has to be modeled as well. In our case this means the wireless link states on the ad hoc network. *Link utilization, (history of) up-time*, and *signal strength* are examples of important metrics of a wireless link of the network. Those metrics will be summarized in a value called *virtual distance*. This model will be presented in more details in Section 3.5.

A neighborhood link table containing the virtual distance to the vicinity nodes is the main data structure of our communication model. This will be heavily used in our service distribution algorithm and in the cluster construction heuristic presented in the next chapters.

3.4.2 Software Components

Before describing in detail how the application and the OS are organized in our system, let us define the software components (or blocks) used in their construction:

Processing thread: A processing thread describes the execution of code associated with a state. There are two subtypes of processing threads:

- **Task:** Tasks are stored at nodes for a particular purpose. For example, they can process locally sensed data or execute an application-specific functionality. They can be thought as part of the distributed application. They do not leave the node and can terminate at their own will.
- **(Mobile) Service:** Nodes may start services at request. Services can be seen as parts of a distributed application that can be used by other parts. This means that the services are offering functionality that is shared among tasks and other services. Services maintain state information associated with each service requester. A service requester is some processing thread that is using the functionality of an external service. It can be either a task or a service. Services are created an request and are mobile entities, i.e., they may migrate among the sensor nodes. Services can be divided in two types:
  - **(Mobile) OS Service:** A service made available by the OS that may migrate. Normally, they are generic and do not implement any application-specific function.
  - **Application Service:** The applications may also implement functions that are shared. Normally, they offer more high level (and specific) functions than the OS services. Those services are treated by the OS in a similar way to the OS services, i.e., the service distribution algorithm acts also on application services.

Additional components are:

Local Service: A service of the OS that does not migrate and may be called just in the local node. It represents a conventional OS service.

Other components of the OS that have no relation to tasks or services will be presented in the appropriate section.

Figure 3.9 shows the relation among processing threads, tasks, and services.

3.4.3 Application

An application in our system is a collection of tasks and application services that has a specific purpose. The tasks and services are the atomic units of the distributed application. This means that their program code must not exceed the resource availability of one single node. Application services migrate using the migration policy offered by the OS (presented in the next chapter).

As already said, the tasks have a fixed placement in a given node, whereas services may migrate in order to improve the system performance. The developer is responsible for the division of the application into several cooperative tasks and services. The idea is that one application has normally fixed tasks placed on nodes, responsible for reading the sensor data, and mobile services, responsible for handling the data and processing it.

An example of an application composed by a task and two services is shown in Figure 3.10. In the example, the task is responsible for reading the data from the node sensor. After this, the domain of

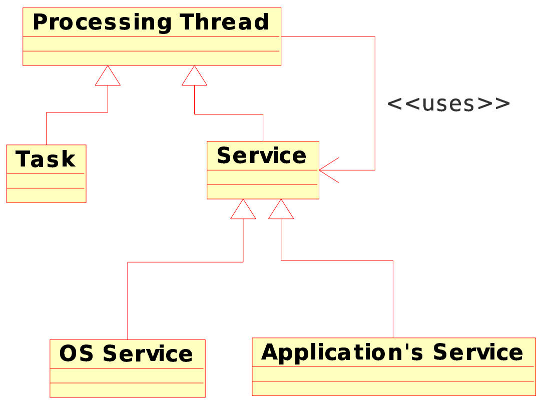


Figure 3.9: The relation among processing threads, tasks, and services.

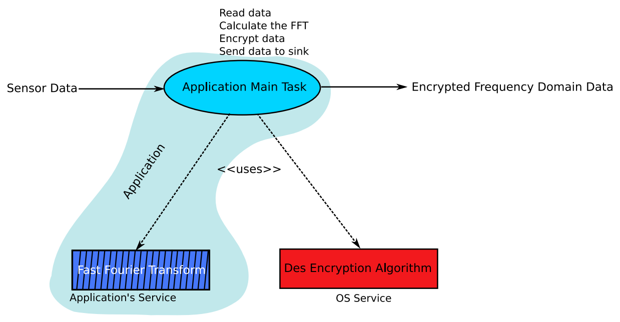


Figure 3.10: Example of a task using two different services.

the temporal series of samples is changed using a user-level service that implements the Fast Fourier transformation. Afterwards, the spectral data is encrypted by an OS service and, finally, sent to the destination (access point of the WSN).

3.4.4 NanoOS Structure

3.4.4.1 Architecture

The operating system is responsible for managing the resources of the system, controlling peripheral devices, and providing software abstractions to the applications [120]. These tasks in the NanoOS are managed using two layers:

Local Management Layer: Device drivers, process management, and memory management are realized at each local node. The hardware dependent part of the operating system lies in this layer.

Mobile Services Layer: Part of the OS functionality is implemented by means of mobile services that may migrate among the nodes in the system. These mobile services implement hardware-independent functions that are used by several applications. Due to the resource restrictions of each single node, instead of having all functions in a single node, they are distributed among a group of nodes.

In this section, we will present the architecture of the NanoOS. Figure 3.11 presents the basic architecture of the OS. The application is also presented in the figure.

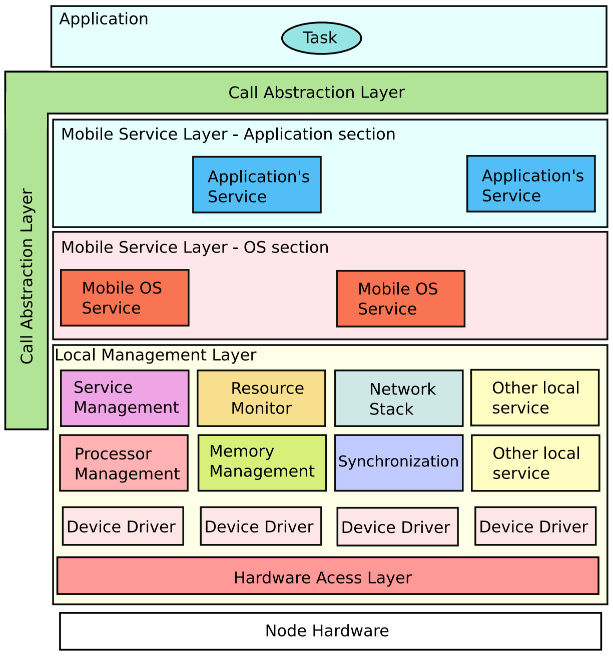


Figure 3.11: Architecture of local node's OS

The functional parts of our OS are:

Hardware Access Layer Translates the abstract hardware calls from the drivers inside the operating system kernel to real hardware accesses.

Device Driver Like in a conventional OS, the device drivers are responsible for abstracting the low level interaction with the devices present on the node.

Processor Management Like conventional embedded OS, our system use a preemptive multi-tasking approach to manage the processor. We aim at reducing as much as possible the number of processing threads in the system in order to avoid the overhead of saving a large number of process states in local stacks. Each thread has its own stack.

Memory Management Also present in each node, it is responsible for managing the volatile memory of the system.

Synchronization Responsible for offering synchronization primitives like semaphores to the application and services and to the OS kernel itself.

Network Stack Here the complete network protocol stack of the system resides. It is responsible for the establishment of an ad-hoc communication infrastructure.

Resource Monitor In a reflexive system, the existence of architecture structures capable of holding the current state and semantics of both the application and system software is essential. The models presented in Section 3.4.1.1 that capture the state of the system and also of the communication links are included in the resource monitor.

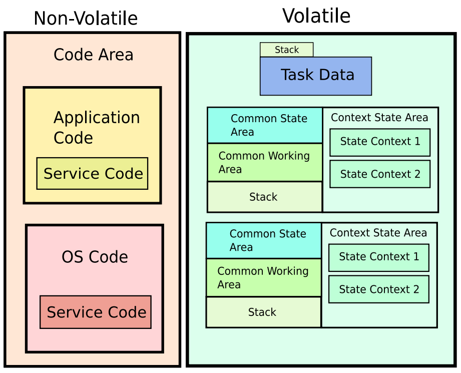


Figure 3.12: Example of memory occupation of the NanoOS. This is a simplified view, just tasks and mobile services are considered.

Service Management Responsible for managing the allocation of a service to a requester. This means that when a certain kind of service is required, the service management starts a service discovery process. Moreover, it is also responsible for creating a new instances of a service (if, for example, the searched served is not found). It also manages a table containing the currently used services (by some requester in the sensor node).

Mobile OS Service It has been already defined in Section 3.4.2.

Mobile Application Service It has been already defined in Section 3.4.2.

Call Abstraction Layer Responsible for receiving the system calls from the user-level tasks or from the mobile services. Depending on the sort of call, it may be handled by a local OS component or by a mobile service. In the last case, the call is redirected to the mobile service by means of a local call (if the service is locally present) or the call is forwarded to the node hosting the service currently. The Call Abstraction Layer is the WSN service interface.

This set of capabilities provides a basic hardware management at the local node as well as an infrastructure to allow mobile services to be transparent to the requester. The upper functions are often present in the middleware layer, nevertheless we decided to include them in the NanoOS in order to reduce the overhead of having extra layers on top of the OS, thus providing a better integration of the mobile services with the OS. Moreover, cross-layer optimization also benefits from this design decision.

It is important to highlight that, with this architecture, the distribution of mobile services in the system is completely transparent to the application (and to the mobile services besides the reconfiguration module). Every service may be installed in every node in the system. The single constraint on the free placement of services is the amount of resources present in the target node.

3.4.4.2 Memory Organization

In this section, the memory organization of the NanoOS is presented. In Figure 3.12.

As already mentioned, we assume nodes with a large non-volatile memory where the code of the OS and application is stored (including services). The data of the services and application are stored in the smaller volatile memory. Moreover, one stack per thread is required and stored in the volatile memory. The different elements of the state of a service are described in the next section.

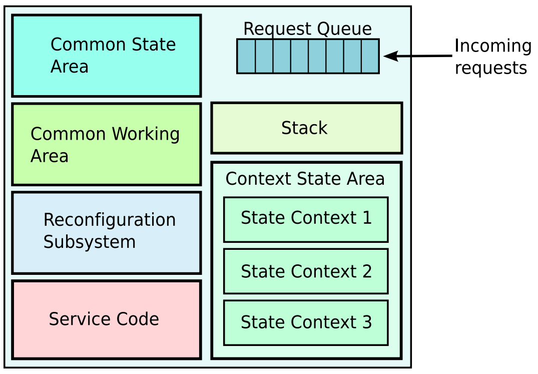


Figure 3.13: Internal organization of the services

3.4.5 Dynamic Mobile Services

The dynamic mobile services are the basic units of the reconfigurable part of our OS. The application services are managed by the OS in a similar manner to the OS mobile services. Therefore, when we refer to a service, we mean both of them.

As already said, services may be installed in any node of the system and are shared among several requesters. The requests may come from the local node as well as from remote nodes.

In the next section, we will define the basic modules of each service.

3.4.5.1 Service Architecture

In this section, the architecture of the mobile services will be presented in detail.

The internal architecture of a service is depicted in Figure 3.13. The following areas are part of a service:

Request Queue: Store the remote procedure call (RPC) requests coming from the service requesters. Here we use a first come first served (FCFS) policy to select which request should be handled next.

Stack: Each service has just one execution thread, and, therefore, just one stack to store the process context is necessary.

Context State area: In the context state area, the state of all currently running contexts of the service are stored. Normally, we have as contexts as many independent applications are calling the services. The contexts will be elucidated in Section 3.4.5.2.

Common state area: This area stores parameters and shared global variables that are not dependent on any specific running context and are shared among all contexts. In an encryption service, for example, this will be the key that is used to encrypt the data coming from all requesters.

Common working area: This area can be used by a context to store temporary values when executing some request. As just one context is active at the same time (because just one thread is available), at the end of the request, it must be assumed that the area may be overwritten (by the context activated in the sequence).

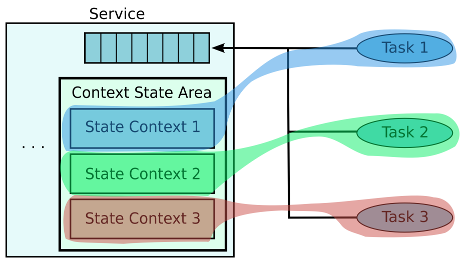


Figure 3.14: Example of a service instance containing tree contexts.

Reconfiguration subsystem: The reconfiguration subsystem is responsible for storing the reflective information regarding the service and determining the position of the service in the wireless network. For that, it uses its own reflective information and global information stored inside the OS.

Code area: Memory region where the code of the server is stored.

3.4.5.2 Service Instance and Contexts

The following terms are used to describe the running entities and states of a mobile service:

Service Instance: An executed copy of a certain service, with dedicated thread. Each memory block described in the last section has its memory allocated. A certain type of service (e.g. encryption service) may have more than one instance running at a given point of time in the NanoOS system.

Service Context: Each requester (or group of requesters) using the service for a particular purpose has the illusion that the service is running just on its own behalf. This defines a service context: an independent run of the service, working on its own data. In the object-oriented nomenclature, it is equivalent to an “instance” of an object. To better explain, an example is depicted in Figure 3.14. We have in this example three tasks. Each task is independent and is willing to use service *s*. A service context is assigned to each of them. From the point of view of each requester (task), they are using the service exclusively. The state associated to each requester is stored separately.

The main reason for the introduction of the service contexts is to use the resources more efficiently. If, for each independent requester, a service context is used instead of creating a completely new instance of the service to manage the requests, overheads can be avoided:

- ◇ Common state area. Here common parameters that are stored are identical for all requesters.
- ◇ Common working area. Due to the fact that each request is processed until termination (because each service has only one thread of execution), the memory that is allocated for temporary processing may be reused for the next request (coming perhaps from other requester).
- ◇ Stack. Just one local stack is used for all contexts. When the service thread is preempted, just one context can be active, which means that its state must be stored. If several instances of the service are started (instead of contexts), each one will require a private stack.

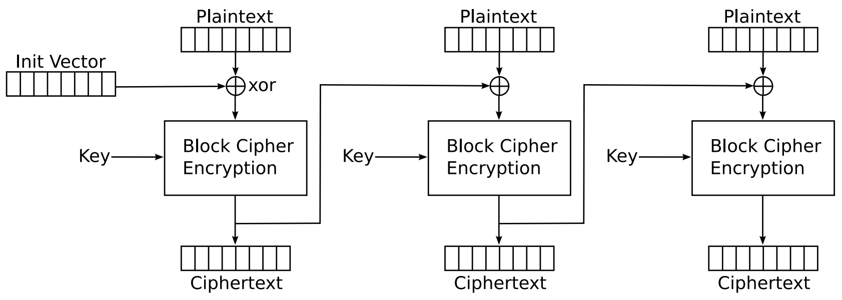


Figure 3.15: Cipher Block Chaining (CBC) mode encryption.

Relationship between contexts and requesters A service context may handle related requests coming from more than one requester. These requests are correlated from the point of view of the service, i.e., they are using the same state. An example may be a service that compresses sensor data using spatial and temporal correlation among the samples and sends this data to a sink. Several requesters are sending their sampled data to the same service context, which compresses it and sends it, at end, to a sink, which is another requester.

The example shows that the relationship of a service context to requesters is one-to-many.

Service Replicas Another concept related to the service instance are service replicas. A replica maintains the same state of the principal service. It is used to ensure fault tolerance and support self-healing. It is not addressed in this work.

3.4.5.3 Examples

In order to clarify how the services are designed and work, we will give here two examples.

Encryption Service The first example is a conventional processing service. The service is composed by an encryption algorithm (in our example the well known DES algorithm).

DES is a block cipher, i.e., it takes a fixed length string of plaintext bits and transforms it through a series of complicated operations into another ciphertext bitstring of the same length. We suppose the use of DES in the cipher-block chaining (CBC) mode. The way of operation is shown in Figure 3.15. In the CBC mode, each block of plaintext is combined using a xor operation with the previous ciphertext block before being encrypted. In this way, the ciphertext is dependent on all plaintext blocks processed up to that point.

Let us now present how the service architecture looks like. The service accepts *n* requesters and for each requester a context of the service will be launched. This means that up to *n* context states are needed in the state area. A special call initializes the key value that is stored in the common state area, because for all instances using the service the same key will be used. This saves memory that would be wasted storing a copy of the key for every instance.

In the context state area, the last encrypted block is stored in order to be processed in the CBC mode. Upon receiving a request of block encryption, the service uses the DES algorithm to encrypt it using the key stored and the last ciphertext block from the respective context state. The result of the encryption is sent to the requester, and, at the same time, the new cipherblock is stored in the context state. The common working area is used for storage of the intermediate results of the 16 rounds of the DES algorithm. Because the service is composed just by one single thread, upon receiving a request

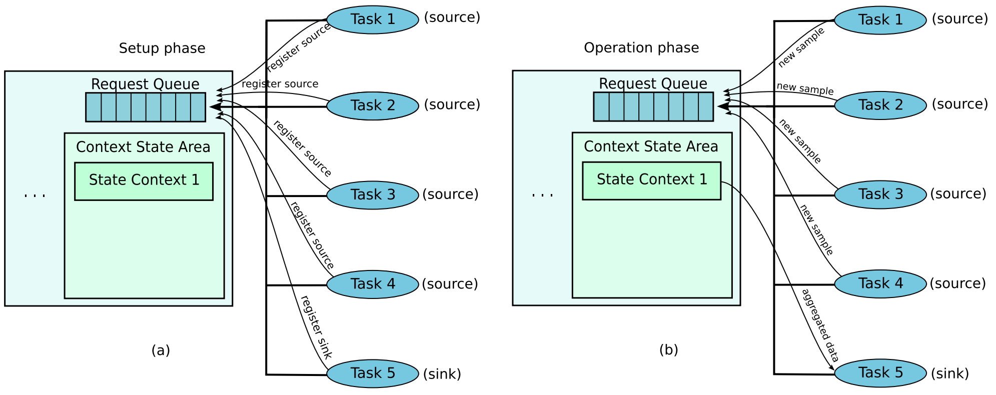


Figure 3.16: Example of data aggregation service using the NanoOS architecture.

for block encryption, this request is ran to completion before the next one be taken from the request queue. Nevertheless, the thread may be preempted by another service or by the OS. The current execution state is then stored in the local stack of the service.

Data Aggregation Service In this example, we show how our service architecture may be used in order to implement a typical sensor service, the data aggregation service.

Normally, data aggregation occurs in a scenario with data sources (which generate data, e.g. from samples from sensor reading) and sinks (consumers of the data). While traveling through the network, an in-network processing possibility is aggregation of the data. This means that the data coming from different sources are summarized in a smaller representation. An example may be computing the mean or the maximum of measured values. There are also complex operations in sophisticated data aggregation based on signal processing techniques. The benefit of such aggregation depends on the position of the data source, relative to the data sink [69].

A common mechanism takes advantage of the fact that the data flows from source to sink along a tree, therefore intermediate nodes may apply aggregation. One important question (among others) is where the aggregation point should be placed.

We propose a data aggregation method based on our service architecture. For that, the aggregation is implemented as an OS service. Each service context may have *n* requesters in the data source role and one requester in the data sink role. In Figure 3.16, we show an example of an aggregation service with four sources and one sink. In (a), the tasks 1 to 4 register themselves as data source. The task 5 is the sink and will receive the aggregated data. After the set-up phase, the aggregation service receives the data samples coming from the sources, summarize them in a smaller representation, and send it to the sink node. This is shown in Figure 3.16(b).

An example of a possible configuration of the network with the aggregation service is shown in Figure 3.17. In the NanoOS approach, we decompose the network in clusters (see Section 3.4.9 and chapter 5). We define for this service that all sources must be inside one cluster (therefore, locality is observed). As we will see in next sections, in the default situation, all requesters of a service are located in the service's cluster. This fit with our conception of the locality of sources. We make an exception for the sink: it may be external to the cluster.

Our approach has some similarities with the LEACH protocol [57]. In LEACH, the network is organized in terms of clusters, and the clusterhead is responsible for receiving the data from members, aggregate them and forward it to the sink. Because we are also organizing the network in clusters, our

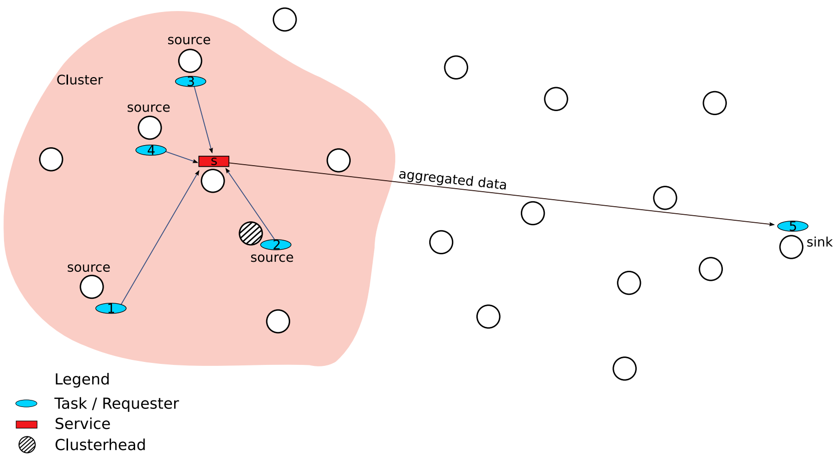


Figure 3.17: The data sources and sink and aggregation services placed in the WSN.

aggregation service executes a similar service to the clusterhead in the LEACH. Nevertheless, there is a notable advantage: to the service distribution heuristic, the system tries to optimize the placement of the aggregation service automatically. This is not the case in LEACH.

Therefore, our advantage, besides the structuring of all services needed in the network into a systematic framework, is the automatic placement of the services, whose goal is the reduction of the communication. Moreover, differently from LECH, the reaction to topology changes is automatic.

3.4.6 Service Management

The service management is responsible for two tasks: the discovery of requested services in the WSN and the creation, on demand, of new instances of services.

3.4.6.1 Service Discovery

The services of the OS are distributed among the nodes of the system. Therefore, before a task of the application or other services can use a given type of service, a node that currently hosts a service of the required type has to be located. This process is called *service discovery* and is reviewed in Chapter 2.

Due to the fact that we are organizing the network in clusters (see Section 3.4.9), we are using a central service directory located in the clusterhead to provide service discovery. Every new service must register itself at the clusterhead. Nodes requiring services will send service discovery requests to the clusterhead, which will return the current location from its internal table.

In the future, a more scalable service discovery mechanism may be used in the NanoOS.

3.4.6.2 Service Instantiation

The service discovery method described above allows a requester to find a required type of service in the network. When the required service is not found, or it is not available (because it is already serving the maximum number of requesters), a new service instance must be created. For that, two methods may be used:

- When a service has been discovered, a quality metric [1] is evaluated, which calculates both the

[1]If no instance of the service was found by the discovery, the quality is zero.

virtual distance between requesting and service node and the load of the service instance (how many other requesters are served). If the quality is below some threshold, a new instance of the service will be created.

- In this method, the service discovery only creates a new service instance if no available service of the requested type was found. If an available service was located, it will be used by the requester.

The new instance of the service is either created in the local node (when there are enough resources) or in the next free node (measured using our link metric). The service distribution algorithm described in Chapter 4 then is responsible for migrating the service to a better network position when necessary.

3.4.6.3 Service Termination

Each service that has no requesters associated to it and is idle for a configured period of time is considered unused. The service manager is responsible for deleting these services, freeing their resources. In this case, any remaining state that eventually exists is erased.

3.4.7 Distribution Methods

After the service discovery phase, the communication phase between the requesters and a given service takes place. We now assume a situation, where many requesters distributed in the system are communicating with many services, which are distributed as well. A single path routing algorithm is responsible for finding a good route between the nodes.

The main goal of the distribution heuristic is to optimize the position of the services by migration, in order to minimize some objective function. In this thesis, the communication overhead minimization is our objective function. In Chapter 4, a service distribution method following this objective is presented.

3.4.8 OS Network Organization

In a large system with a great number of nodes, the organization overhead of the service discovery and distribution can be excessive. For example, in the discovery phase, it may happen that a node at the other end of the system has to be contacted. Pheromone tables (used in the service distribution presented in the chapter 4) will get large. These effects result in bad scalability. In addition, in an open network environment, there is no efficient way of controlling how many instances of a given service are running on the system.

Moreover, depending on which protocols are implemented in the WSN, there may be a need for some local coordinator, in order to avoid the overhead of all nodes storing information about the complete network. This local central coordinator also allows the use of algorithms with a central controller that are sometimes necessary. Nonetheless, this centralistic approach is enclosed just in a certain area of the network. In addition, organizing the network in clusters (and therefore creating hierarchy) may be also useful to support the topology control.

3.4.9 Organizing the Network in Clusters

Since the goal of the distribution methods is the minimization of communication between the requesters and services, a natural grouping of these objects at nearby positions in the network can be expected. If this weak kind of clustering happens anyway, it would be an advantage to define a hard separation of the nodes of the system into clusters. In each cluster, a complete instance of the OS will run. This brings a reduction of the organization overhead, since the discovery process is constrained inside the local cluster and the pheromone tables must only store values for services used by requesters inside the cluster. Moreover, the clusterhead also embraces additional controlling functions like storing the list of all OS service instances (service discovery broker) and also a routing table to other clusters. Each node must just know some information about the cluster to which it belongs to.

In Chapter 5, we investigate thoughtfully methods to solve our clustering problem, called *minimum intracommunication-cost clustering*.

The *minimum intracommunication-cost clustering* problem corresponds to the partition of the nodes of the network into multi-hop groups with a guaranteed minimum amount of resources q (or budget) per cluster. At the same time, it looks for the minimization of the sum of the internal communication costs of each cluster (measured using the link metric). This comes from the requirement of the existence of a certain amount of resources inside a cluster. This amount must be enough to host a complete instance of the OS, application and necessary mobile services inside each cluster. A cluster, from the point of view of OS and distributed services, can be defined as follows:

Definition 3.4.1. *A cluster is a set of nodes where all the services required by any application running in this set are available in some member of the set.*

Each cluster can have $n \in I\!N$ applications using mobile services. The idea is that a cluster can be seen as an "execution environment" where all necessary services required by the applications are present.

When the network has been decomposed in to clusters using the clustering algorithm presented in Chapter 5, the clusterhead is the representative and controller of the instance of the operating system running on the cluster. Moreover, it hosts the central broker for service discovery.

A necessary task of the NanoOS in dynamic environments is to control whether all necessary services are still inside the cluster when the topology changes, causing a re-clustering process. Instead of a central controller polling the location of the components of the system repeatedly, the process of restructuring the cluster is triggered by the moved entity. We recognize two kinds of entities that may change the cluster affiliation:

- Services - When a service, due to topology changes, arrives undesired in another cluster, it must be replaced in the original one.
- Application tasks - The same may happen with application tasks. They may arrive in another cluster due to a topology change plus re-clustering process. Since the tasks' location is not controlled by the OS, some mechanism should adapt the new cluster to the arrived task. This means that the state of the services that are in the old cluster must be transferred to the new one.

Mechanisms to handle these two cases are presented in the next section.

3.4.9.1 Constraining Services Inside the Cluster

Each service and each requester has an internal variable `assigned_cluster` that indicates to which cluster it belongs to. Each time that a service receives the notification `new_cluster`, it checks whether the node still belong to the `assigned_cluster`. When not, this means that, due to a cluster reconstruction activity (triggered by, for example, a topology change), the node has to change the cluster which it belongs to. This requires a migration of the service back to the corresponding cluster. For

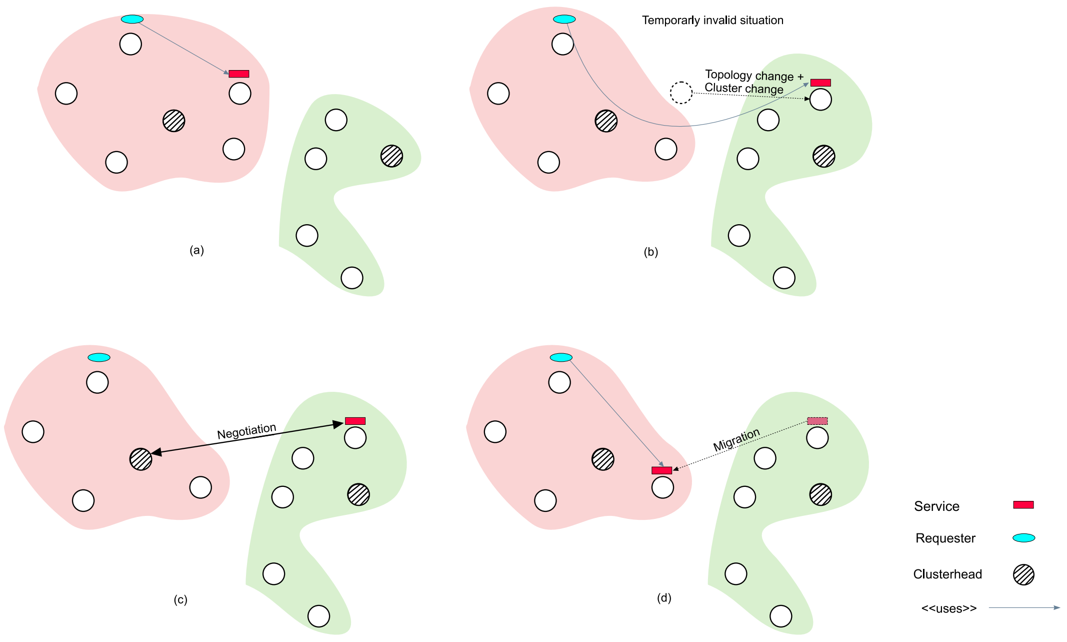


Figure 3.18: Inter-cluster service migration

that, the service contacts the clusterhead of the target cluster, and it assigns a node to the service. An example of this situation is shown in Figure 3.18. In (a), an example of two clusters and a service-requester in the first one is shown. In (b), the node with the service moves and a topology change happens. As answer to this situation, a re-clustering process takes place and the node now belongs to cluster two. In (c), the service notices this situation and negotiates with the clusterhead of its `assigned_cluster` a new node to host it. After this migration, that can be seen in (c), a valid situation is achieved again.

3.4.9.2 Reacting upon Task's Cluster-Membership Changes

In this subsection, a mechanism to react upon cluster membership changes of the hosting node of a task is presented. Differently from services, tasks can not be replaced by the OS. When a task arrives in a different cluster due to some re-clustering process, an approach is required to prepare this new cluster to execute the task.

Like services, tasks have an internal variable `assigned_cluster` to indicate which cluster they belong to. When the node is assigned to a new cluster, the cluster sends the notification `new_cluster` to all tasks and services. Upon receiving it, it checks whether the current cluster has been changed. If positive, the service contexts that include the actual task as the only requester associated with them should be transferred to the new cluster.

For each of these contexts, a negotiation occurs between the task and the clusterhead of the new cluster, to check whether the same type of service already exists in the new cluster. If positive, the service context will be migrated to that service. If not, a new instance of the service is created. At the same time, the service instance containing the context is contacted to check whether the migration can occur. If positive, the context related with the task is migrated from the original cluster to the new one.

In Figure 3.19, we depict this situation using an example. In (a), the tasks t_1 and t_2 are using the service s_1 inside the cluster 1. Each of them has its own state and represents independent executions of the service context. In (b), the node hosting t_2 changes its physical position. As reaction, the re-clustering process takes place and now the node is in the cluster 2.

This situation is detected, and a negotiation takes place in order to migrate the context related with t_2 from the service s_1, located in the cluster 1, to the cluster 2 (Figure 3.19 (b)). Due to the fact that no service of the same type is currently instantiated in the cluster 2, a new instance is created and receives the context state related to the task t_2 (Figure 3.19 (c)). The task may now resume its execution.

It is important to remark that our proposals are best effort approaches, i.e., there is no guarantee that a service continues to be provided in the case of large topology changes. The applications must be programmed to cope with situations where the service they were using isn't anymore accessible.

3.5 Communication Link Model

In this section, we will present the communication link model used as base in the algorithms developed in this thesis. It is implemented in the logical link layer of the sensor nodes.

3.5.1 Links in a Wireless Network

A very important difference between the wired and the wireless networks is the behavior of the network links. In a wired network, the links have a relatively stable quality. The parameter that has a high influence on this type of link is the load of the network.

On the other hand, in ad hoc wireless networks, there are several parameters that influence the link quality. First of all, the propagation of the waves in a wireless medium is affected by phenomena like attenuation, distortion, exponential path loss, etc. Moreover, the environment is dynamic, with changing obstacles, temperature, and pressure that affect the transmission properties.

As already shown, the transmission over wireless channels are subject to several physical phenomena that distort the original signal. This distortion introduces uncertainty at the receiver about the original signal, resulting in bit errors. The wave propagation phenomena that contributes to the distortion are reflection, diffraction, scattering, and doppler fading [69]. Moreover, noise and interference lead also to reception errors.

Because the quality of a link is an important factor in a wireless network, our model the links is based on a *link rating* provided by the logical link control layer. This rating reflects the "usefulness" or "quality" of a link. The ad hoc network being modeled by an undirected graph $G = (V,E)$, where V is the set of wireless nodes and an edge $\{u,v\} \in E$ if and only if a communication link is established between node $u \in V$ and $v \in V$, we define, for each link, a weighting function that assigns a positive weight $w : E \longrightarrow [0,1]$, where 0 means an "excellent" quality and 1 means "very poor" quality. For a link $\{u,v\} \notin E$, we define $w(u,v) = \infty$.

The properties of a wireless link make the task of finding the appropriate link rating a challenge. How the quality of a wireless link may change under a very uniform environment can be seen in the experiment reported in by [133]. In this experiment, Berkeley Mica Motes running TinyOS were arranged linearly with a spacing of $60cm$. They measure the packet loss rates at different distances in different pairs of nodes. In Figure 3.20, a scatter plot of how links vary over distance for a collection of nodes on the ground of a tennis court is shown. Although in such an ideal environment a behavior near to the theoretical path loss curve was expected, the results depict a very different reality. After a certain distance ($4m$), the difference on the reception success rate between nodes at the same distance was

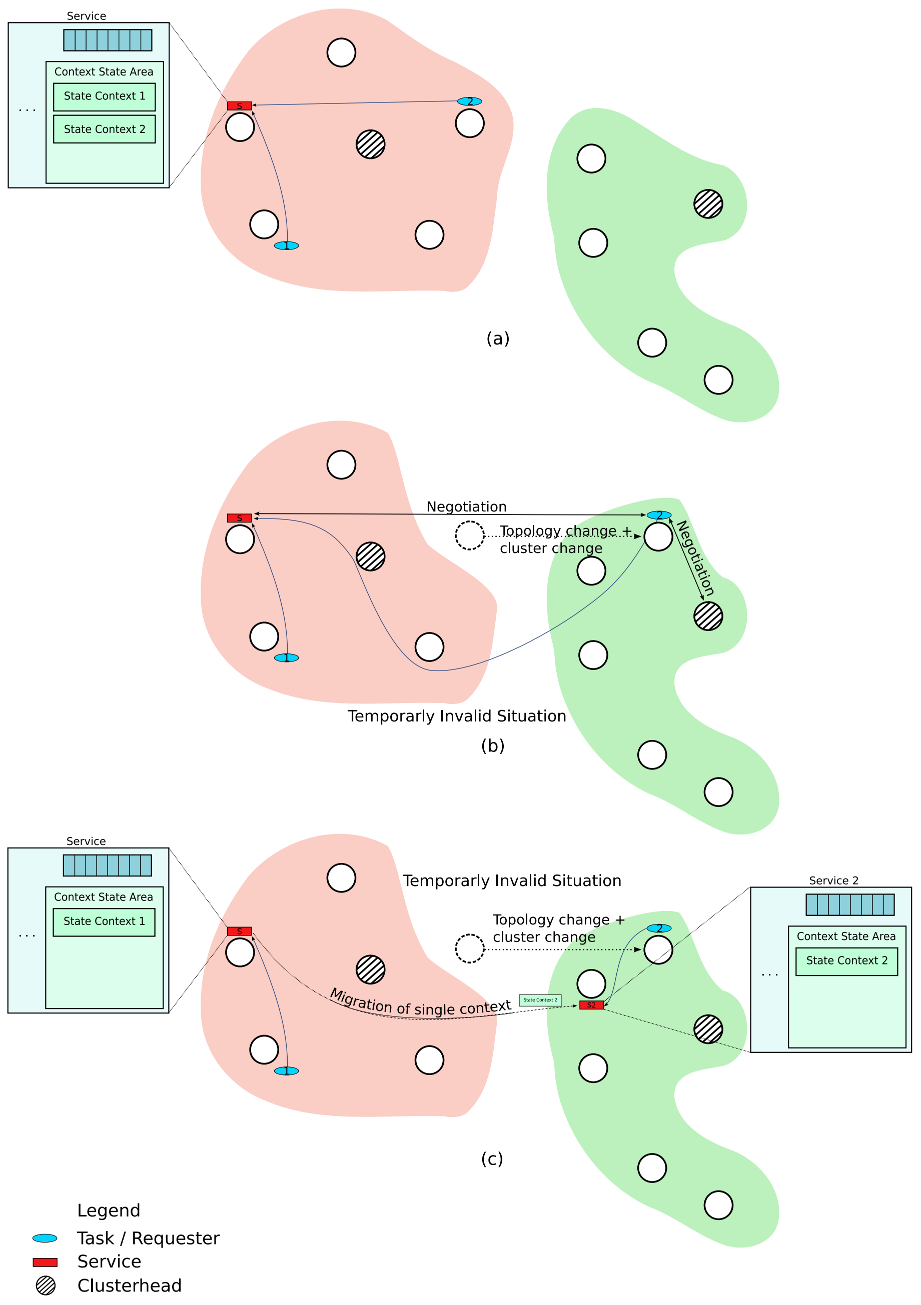


Figure 3.19: Example of a task changing the current cluster and the service reorganization.

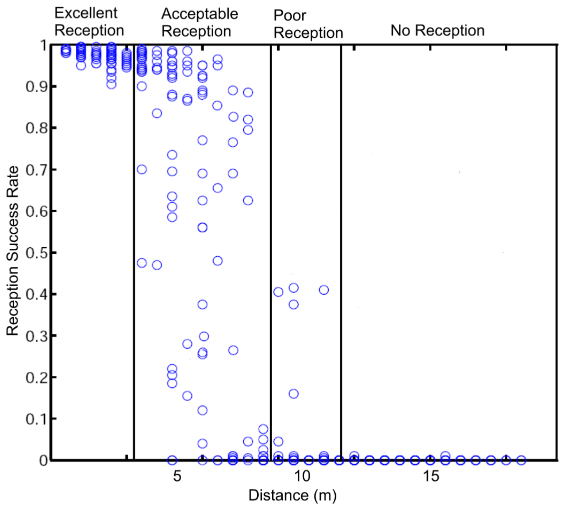


Figure 3.20: Reception success rate versus distance of the transmiter/receiver (data source: [133]).

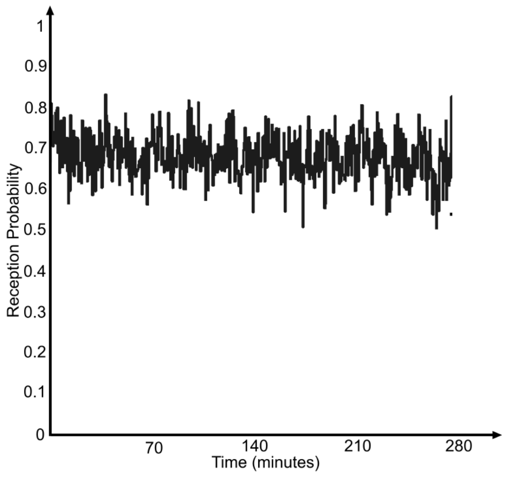


Figure 3.21: Stability of a link between two stationary nodes [133].

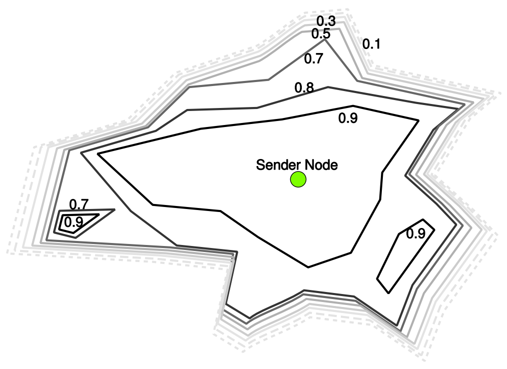


Figure 3.22: 2-D chart demonstrating that the link quality does not depend only on the distance, but also on the position of the sender and receiver. Data source: [133]

very significant. This could be verified in the regions marked by “acceptable” and “poor” receptions (in the paper the areas are called *transitional regions*).

The labels in the picture were assigned based on the average reception success rate (RSR). We can define thresholds for the inferior limit of the four defined regions, i.e., $RSR_{iexcellent}$, $RSR_{iacceptable}$ and RSR_{ibad}.

In Figure 3.21, the stability of a link between nodes that are 2.4m far from each other is shown. Although the mean quality is relatively stable, there are significant variations in the instantaneous link quality.

Figure 3.22 depicts the reception success rate in a 2-dimensional chart. We can conclude, from this chart, that the link quality dependents not just on the distance between the communicating partners, but also on the node’s physical position. Small changes in the position may generate considerable changes in the reception success rate [68].

The experiments described here illustrate how challenging is the development of a trustworthy link metric.

Besides this, many approaches are based on a *bimodal* link quality, where a link may exist or not. Although this may often be a true assumption for wired networks, it is not a reasonable approximation for wireless networks. Algorithms based on this simplistic assumption often choose low-capacity, long-range links instead of high-capacity, low-error links. This affects negatively the performance. It happens because bad links are good enough for control packet exchange, but during data transmissions, much of the capacity is consumed by retransmissions and error corrections.

In the publications [37, 38], this effect is observed for routing protocols. The authors try to minimize the hop-count, bringing a prioritization of bad signal strength and maximizing the loss ration. The effects of this selection are analyzed in the papers.

3.5.2 Link Quality Estimation

To estimate the link quality based on the observations of the transmissions that already took place is an example of filtering problem. The following properties are desirable from a good estimation [69]:

Precision The collected statistics should give useful prediction precision.

Agility When a rapid change happens in a link (for example, the node moves), the metric should react quickly.

Stability The metric should be immune to sporadic noise/fluctuations. For that, the filtering theory can be applied.

Efficiency The snoop of packets from the neighborhood and the storage of old values in order to improve the filtering are costly operations in a wireless sensor network and should be avoided when possible.

The most important properties are agility and stability. Ideally, a network estimator should be agile when possible, nevertheless stable when necessary - it should adapt to the prevailing network conditions [70].

There are two types of estimators. An active estimator uses special packets to measure the link quality, whereas, in a passive estimator, the estimation is made based on snooping the neighborhood transmission and detecting packet loss based on sequence numbering.

In the work [131], the trade-offs between stability (in term of variance or standard deviation), agility and history are studied. The problem observed is that the relation between the standard deviation and the number of samples is quadratic, i.e., for a stable estimator, a large number of samples may be required, which leads to a less agile estimator. Moreover, different kinds of estimators are analyzed in the work.

The exponentially weighted moving average (EWMA) is simple and memory efficient. Besides that, it uses infinite history by means of a linear combination of all past events weighted exponentially. It has the property of being reactive to small shifts. The basic principle of this filter can be summarized by $P = \alpha \cdot P + (1 - \alpha)r$, where P is the estimator outcome and r is the actual input from some sensor. α is the tuning parameter.

The publication [131] makes a comparison between the EWMA and other simple estimators like the flip-flop EWMA, moving average, time weighted moving average, and window mean with EWMA. They found that EWMA performs best overall over an average in a time window (WMEWMA).

In the work [70], the comparison of four filters and their adaptation to the network prevailing conditions are shown. The filters were flip-flop EWMA, stability filter, error-based filter, and Kalman filter. Although the Kalman filter, when applied to a linear system, is *optimal*, it requires a significant knowledge of the system that is not available when estimating network performance. Therefore, reasonable guesses were employed and led to good results.

They conclude in the work that the flip-flip filter brought better results in terms of agility and stability.

As we will present in the next section, instead of using just one parameter to obtain a link estimate (like here the error rate was used), we decide to make a combination of several parameters. Some are used in order to give the estimator a fast response to changes, others to improve the long term prediction. Moreover, we decided to use, for some metrics, the basic exponentially weighted moving average, due to its straightforward implementation and its low memory and computational requirement. This is an advantage for sensor networks.

3.5.3 The Combined Link Metric

In this section, we will define a link metric that summarizes the “goodness” of a link. Each link receives a real value that describes its quality. The algorithms developed in the next chapters of this thesis use this link metric instead of using the bimodal link model as usual for many existing approaches.

The following variables will be used in order to estimate the link quality. They are summarized in our combined link metric.

1. Success Rate
2. Received Signal Strength
3. History
4. Energy Reserve

3.5.3.1 Success Rate

The idea is to use past samples of the success rate in order to estimate the quality of a link. Besides the filter analysis presented in the previous section, there are more practical approaches using the success rate as link metric. The approach presented in [37, 38] uses a metric called *expected transmission count* (ETX) to estimate the success rate. The metric predicts the number of transmissions (including retransmissions) required to successfully deliver a packet. It uses the values of d_f (forward delivery ratio) and d_r (reverse delivery ratio) to calculate ETX ($ETX = \frac{1}{d_f \cdot d_r}$). The forward delivery ratio is the measured probability that a given data packet successfully arrives at the receiver. The reverse delivery ratio is the probability that the acknowledgment packets are successfully received. The values of d_f and d_r are measured using dedicated beacon packets.

In the work [40], the authors use rewards associated to transmission and receptions events in order to calculate the delivery ratio. Also promiscuously received packets are used in the metric.

Success rate is a relatively reliable method to predict the quality of a link. Nevertheless, there are also some drawbacks: at the beginning of the observation, there is no data to be used for the prediction; moreover, it reacts slowly to changes in the topology (a node has moved but the link rating still indicates a good link). In addition, very old measures could not estimate accurately the current situation.

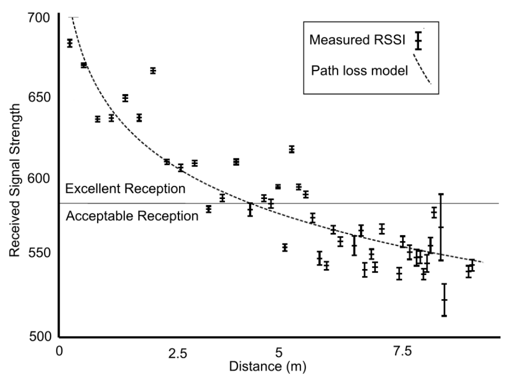


Figure 3.23: Correlation between signal strengh indication and distance for two nodes. Data source [141]

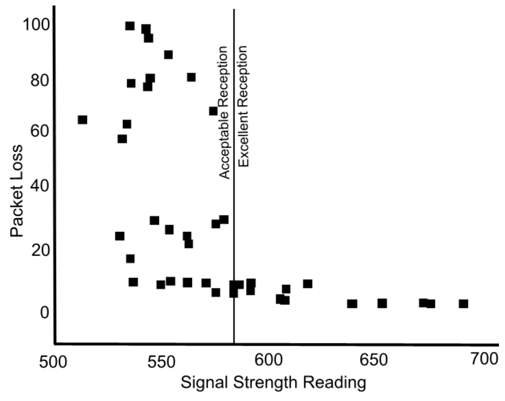


Figure 3.24: Correlation between signal strangh and data loss rate. Data source [141]

3.5.3.2 Received Signal Strength

The received signal strength indication (RSSI) as link metric is a proposed substitute of the bimodal links in some approaches, because most network hardware provides it.

The correlation between the received signal strength and the distance of two nodes is rather far from the ideal path loss curve. This can be seen by the experiment performed by Zhao and Govindan [141], showed in Figure 3.23. It has been measured in a office hallway. In the picture, the deviation of the measured RSSI from the theoretical curve can be seen. Although the measured RSSI roughly follows the curve, high deviations in both sides of the path loss model can be seen. Moreover, the larger the distance the higher the variance of the measured data.

When analyzed out of context, the figure may give the impression that the RSSI alone could be a good estimator, due to the fact that the measured data roughly follow the path loss. Figure 3.24 opposes this naive assumption. It analyzes the relationship between the signal strength rate and the packet loss (complement of the reception success rate). For signal strength readings higher than 580, almost all packets are received. In the range from 500 to 580, the packet loss is scattered over the whole range of the packet loss. This fact shows a drawback of using the RSSI as the only estimator for link quality. However, a small correlation between the signal strength and the packet loss can be seen even in this region.

Because the received signal strength indicator roughly follows the path loss, we can compare Figure 3.24 with Figure 3.20. The results from the experiments are similar: the behavior of the link can not be clearly defined by means of the RSSI (or distance).

Cerpa et al. [28] performed similar experiments with mica motes in an indoor office, an outdoor urban, and outdoor habitat environments. They obtained similar results. However, for outdoor environments, the variability of the links were even higher than the results from [141]. They characterized micas' links as asymmetrical (some links have different reception rates between sending and receiving), non-isotropic (the connectivity was not necessarily the same in all the directions from the source), and with non-monotonic distance decay (nodes geographically far away from the source may get better connectivity than closer ones).

As conclusion of this section, we argue that the signal strength may be used just as a roughly indicator of the quality of the link. Therefore, we integrate it with other indicators in our combined metric.

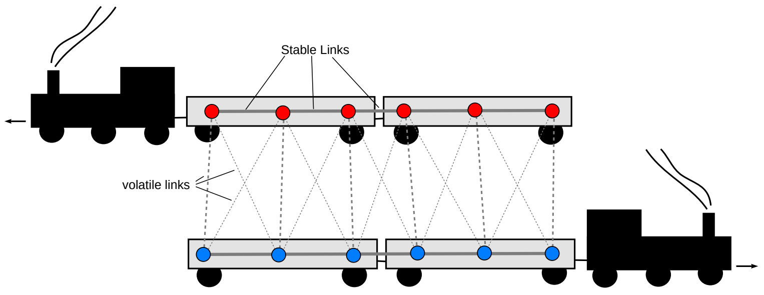


Figure 3.25: Example of stable versus volatile links in motion of two groups of nodes.

3.5.3.3 History

In the algorithms developed in this work, it is important to select trustworthy and stable links instead of newly created ones. In Figure 3.25, a situation where two groups of nodes are placed in two different trains moving in opposite directions is presented. The connections between the nodes from one group to the other on are volatile, because the connections will break soon. For the cluster construction or service distribution (presented later in this thesis), using nodes from the opposite group has drawback, because the link will be broken soon and a high overhead to reorganize the clusters (or services) will result.

To prevent the use of temporarily links, an additional parameter is used in the metric. It measures how long is the existence of the link and penalizes very new links.

It is important to say that for other applications (e.g. routing protocols), the use of volatile links may bring some advantage because those links may reduce considerably the route between two nodes.

3.5.3.4 Energy Reserve

In a sensor network environment, the energy is a precious resource and the pattern how energy is spent makes a real difference concerning the complete network life time. In [69], the network lifetime is defined by the time during which a sensor network can fulfill its purpose. Some possible definitions for this event are:

- Time to first node death
- Network half time - 50% of the nodes ran out of energy
- Time to partition
- Time to loss of coverage
- Time to failure of the first event notification

More generally, it is also possible to consider the nodes lifetime distribution. One question could be which percentage of the nodes are still operational at any given moment. Curves where the probability of many nodes functioning in a short term sacrificing long life networks that have few nodes at the end are a preferable situation, i.e., the energy should be spent in a uniform manner.

We decide to include the amount of energy of a node in the link metric to restrict the use of exhausted nodes, because the link metric tends to evaluate them worse than links among nodes plenty of energy. This brings a more uniform consumption of energy. The energy reserve parameter in the link metric may improve a uniform energy use, specially by routing protocols.

3.5.3.5 Combined Metric

We combine the presented parameters in a link metric that indicates the goodness of a link. The statistic-based observation of transmission success is a good indication of the future success rate, nevertheless it reacts slowly to changes and at beginning has no data to be calculated. The received signal strength indication makes possible quick indications, but it is not very precise. Therefore, the combined metric uses these two parameters. Moreover, in order to prioritize stable links, the history is also used. Finally, the energy is also included in the link metric to promote a uniform use of the nodes in the sensor network. The combined metric is defined in eq. 3.1.

$$M_{combined} = 1 - \left(k_1 \cdot M_{RSSI} + k_2 \cdot M_{RSR} + k_3 \cdot M_{history} + k_4 \cdot M_{energy} + 0 \cdot k_5\right) \tag{3.1}$$

where $M_{RSSI} \in [0,1]$ indicates the normalized signal strength indication, $M_{RSR} \in [0,1]$ is the reception success rate, $M_{history} \in [0,1]$ returns 0 for new links and 1 for old ones, $M_{energy} \in [0,1]$ returns 0 for depleted nodes and 1 for full nodes. The last parameter indicates the cost of the hop, i.e., a fixed cost that should be added to all links.

The terms virtual distance and link metric are synonyms in this work, i.e., $w(u,v) = M_{combined}(u,v)$ where $M_{combined}(u,v)$ is the combined link metric measure in the link $\{u,v\}$. A small virtual distance means good connection, whereas a large one means a link with high error rate.

In Figure 3.26, examples of how the link metric works are shown. In (a), we have a network with unrated links. If the node 1 wants to send a message to 2, there are three paths using the minimal hop count of 2. In (b), the same network is shown, but the links are rated using the link metric described in this section. Now a differentiation between the possible routes from node 1 to 2 can be made, and the route with the best metric would be selected. In (c), a new situation is illustrated: a new node appears. At the beginning, the links to this node are relatively poorly rated, because of the lack of confidence about the stability of the link. After some time, in (d), the metrics have been stabilized.

The effect of an almost depleted node is shown in (e),(f). In (e), the network in its normal operation is presented. After routing several packets, the battery of node 3 becomes almost depleted. This reflects in the link metric, and now another route between 1 and 2, not using node 3, is the preferred one. This improves the uniformity of the energy consumption among nodes in the network.

Modifications of this basic link metric may be used in order to better assess the quality of the link taking in account the environment where the nodes are deployed or to drive the higher level protocols to achieve some desired property. For example, in order to force the clustering heuristic presented in the chapter 5 to produce a higher number of flat clusters instead of deep ones (to reduce the interference among different clusters and improve the spacial correlations between sensors), a linerization of the RSSI can be used. In the same way, other variations may be included in the link metric to achieve a desired objective.

We now will present how the sub-metrics used in the equation 3.1 are calculated. The value of M_{RSSI} is adjusted upon reception of any packet (addressed to the node or acquired in promiscuous mode). An average of the received values and the current M_RSSI with an aging factor α is calculated, i.e., $M_{RSSI} = \alpha \cdot M_{RSSI} + (1-\alpha) \cdot am_{RSSI}$, where am_{RSSI} denotes the adjusted measured signal strength. The adjustment in the signal strength is done in order to improve its performance by cutting out extremes where the signal is excellent ($RSSI_{excellent}$) or very poor ($RSSI_{verypoor}$).

Therefore,

$$am_{RSSI} = \begin{cases} 1 & \text{if} \quad meas_{RSSI} > RSSI_{excellent} \\ 0 & \text{if} \quad meas_{RSSI} < RSSI_{verypoor} \\ \frac{meas_{RSSI} - RSSI_{verypoor}}{RSSI_{excellent} - RSSI_{verypoor}} & \text{otherwise} \end{cases}$$

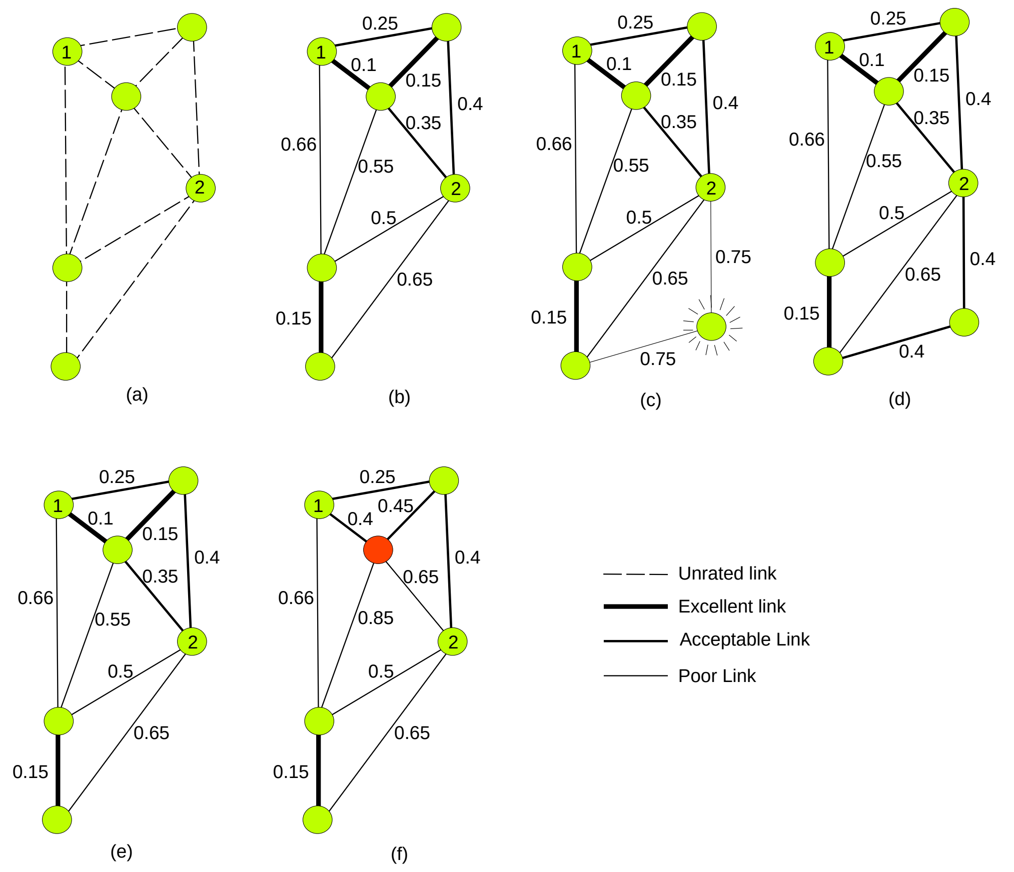


Figure 3.26: Example of link metric ratings. (a) Network with a bimodal metric. Hop count may be used as distance metric. (b) The same network with our combined metric. (c) Situation where a new node joins the network. Because these new links have been just created, the rating is poor. (d) After confirming the stability of the links, the ratings got to a stable situation. (e) Same original network. (f) Node 4 runs out of energy, link ratings reflect this fact. In (b),(c),(d), $k_1 = 0.33$, $k_2 = 0.33$, $k_3 = 0.33$ and $k_4 = 0$. In (e),(f), $k_1 = 0.33$, $k_2 = 0.33$, $k_3 = 0$ and $k_4 = 0.33$.

where $meas_{RSSI}$ is the raw measured RSSI.

The metric M_{RSR} is just the combination of the current measured reception success rate with the existing one, i.e., $M_{RSR} = \alpha \cdot M_{RSR} + (1-\alpha) \cdot meas_{RSR}$. The measured reception success rate ($meas_{RSR}$) is calculated based on the monitoring of packet transmission and correlated acknowledgment in the MAC layer.

The history metric ($M_{history}$) is calculated using the number of received packets. Let C_{rx} be the number of received packets of the link. This counter is decremented periodically (down to 0) in order to cope with extinguishing links. We define $M_{history} = min(1, \frac{C_{rx}}{stable\_link\_count})$, where $stable\_link\_count$ is the number of packets necessary to consider a link as fully active.

Finally, the energy reserve measures how much energy a node has, i.e. M_{energy} returns one when the battery is full and zero when depleted.

3.6 Discussion

In this chapter, we presented the basic architecture of our innovative operating system for wireless sensor networks. Our approach is unified in the sense that it tries to combine basic OS functionality with rather high-level support of distributed applications and an infrastructure that supports the easily development of distributed services. Differently from most existing systems, we are not targeting only at the classical data-driven sensor network applications, but also distributed processing (e.g. distributed signal processing) and diversity of services.

Our basic abstractions for the application development are the mobile service and the task. Tasks have static position in the system and are responsible, for example, for reading sensing data. The mobile services are responsible for transforming this sensing data in a more high-level system response. In order to overcome the resource limitations of each node, these mobile services may be installed in any node of the system and accessed remotely.

When designed as a conventional service middleware (e.g. Corba [96]), one main problem that remains in this architecture is that for each requesting application, a service instance should be created and a local stack should be used. This results in a large memory requirement for each requester. In order to avoid that, we propose the service context concept. Each instance of the service may accept several requesting applications, and, instead of a new instance being created for each of them, a context is created and the instance of the service remains only with one stack. The drawback here is that each request must be run to completion, there is no preemption between different contexts inside a service, but just between different services, tasks, and OS core inside a node.

With the proposed architecture, it is possible to design classical WSN applications (like data-fusion) as well as background processing tasks. Moreover, the system may include at some instant of time a large amount of different services distributed inside a cluster of nodes.

Due to the facts that we are aiming at dynamic networks and we support self-organization and self-optimization, an algorithm that controls the migration of the services to suitable positions is necessary and presented in the next chapter.

Instead of supporting the overhead of keeping information of very distant nodes, we are also logically joining together nearby nodes that form clusters. The clusters are the space where the service distribution takes place. Moreover, “centralized” approaches may be used inside a cluster without compromising the scalability of the system.

In order to measure how distant two nodes are, we decide to focus on the communication. We aim at measuring how error-prone a certain connection between two nodes is using our combined linked metric called virtual distance. Differently from many usual approaches in ad hoc or sensor networks, we use our metric instead of the hop count to assess the communication cost between two nodes. Our link metric is the base of all other algorithms developed in the scope of this thesis.

In the next chapter, we will provide a description of the service distribution algorithm. It is decentralized and uses just local information to take its decisions, thus complying with the philosophy of our OS presented in this chapter. Moreover, in Chapter 5, two different heuristics aiming at the decompositions of the network in clusters with a guaranteed size are presented. The architecture of the NanoOS is important to calculate this amount of service. It is necessary to know the applications that will run in each cluster, their requirements in terms of services (and memory usage), and the OS requirements in order to assess an adequate minimum cluster size to run an OS and application instance.

Even if a more unified way of programming is desired, where the WSN is seen as an aggregate, our OS can be used. An extension of the NanoOS can be implemented to allow the injection of new tasks (queries) in the system and their autonomic replication. This replication may be to all or part of the nodes of the WSN, like in Maté or Sensorware. Our system can provide a large set of mobile services, where these services migrate automatically and are used by the injected tasks. The tasks may orchestrate with help of the local services, sensors and mobile services the desired complex distributed algorithm.

Chapter 4

Service Distribution

4.1 Introduction

In this chapter we will deal with a fundamental question in our system: where should the mobile services be placed in the wireless sensor network. Several different metrics may be used to evaluate how good is a given allocation. We decide to minimize the global communication cost measured by means of our virtual distance metric. The idea is that the minimization of the communication overhead has a direct impact on the energy used by the system.

There are several possible approaches to control the placement and migration of modules in a distributed system. We review them in the section 4.2. In the section 4.3, we define formally our specific problem. Subsequently, we present two heuristics (section 4.4) that are responsible to control the placement and migration of our services in the system. The basic and extended versions of our heuristic aim at minimization of the objective function formally presented in the section 4.3. It was very important in the design of the heuristics to assure that they use just local communications, impose a small overhead on the node and network, and avoid as much as possible the use of control messages. In addition, the cost of the new placement shouldn't overcome its benefit in realistic conditions.

4.2 Related Work

4.2.1 Global Distributed Scheduling

4.2.1.1 Introduction

The placement and migration of components in a distributed system is a problem that has been studied in various application areas. In the parallel/distributed systems area, an extensible used approach is to share the total processing load among the existing CPU resources.

Scheduling of tasks in a system with distributed load involves deciding when to execute a job and where to execute it. Normally, the two tasks are handled separately by two components: the allocator and the scheduler.

The allocator decides where to run a certain task. The decision in each local node of when to select the task for execution is done by node's local time-sharing based scheduler (this problem is also called "local scheduling problem"). This means that each node decides when to run the existing local tasks, but the higher level decision of allocating a processor to a task is made by the allocator. These scheme brings modularity and separate the load distribution concerns from the details of the local schedule.

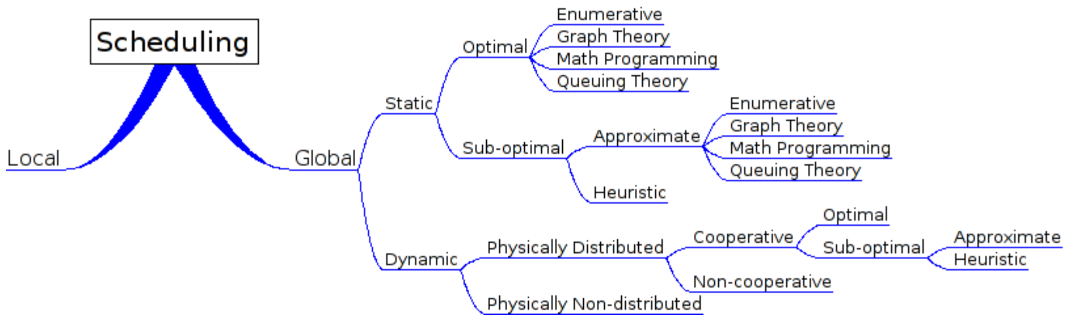


Figure 4.1: Scheduling Taxonomy.

The problem of deciding where some task in a system will be executed is called in the literature process allocation problem or global scheduling problem. A variety of widely differing techniques and methodologies for scheduling processes in a distributed system have been proposed.

In order to classify the approaches, a taxonomy was developed.

4.2.1.2 Scheduling Taxonomy

In the paper [26], a hierarchical classification of the scheduling methods focusing in the global scheduling is made. The structure of this classification is shown in Fig. 4.1.

At the top of the hierarchy, the scheduling methods are classified in *local* and *global*, as already discussed in the introduction section. Concerning global schedulers, a further classification is *static* and *dynamic*. The *static* global schedulers assume an initial information about the total mix of tasks as well as the communication and the dependence among tasks. Using this information, a process allocation is calculated and the tasks have a static assignment to target processors.

In a *dynamic* scheduler, the current and previous state information of the system are take into account to make dynamic decisions (on-the-fly) where the tasks should be placed.

The *static* approaches can be optimal and sub-optimal. In the optimal ones, based in some objective function, the optimal assignment can be done. As the assignment problem is normally computationally infeasible, sub-optimal solutions may be tried. This can be an *approximated* solution, where some bound is desired and the same formal model of the algorithm for the optimal solution is used. It can be also a *heuristic* solution, where realistic assumptions are made about *a priory* knowledge concerning process and system loading characteristics.

The optimal solutions may be found searching the solution space, using graph theoretic approaches, mathematical programming or queuing theory.

The next classification concerning *dynamic* schedulers regards where the decision of the allocation takes place: *non-distributed*, where the scheduler resides physically in a single process or *distributed*, where it is physically distributed among processors.

Distributed allocation may be done using *cooperative* or *non-cooperative* techniques. In the *cooperative* mechanism, cooperation between the distributed components is used to take the decision where to place a task in the system. In the *non-cooperative* technique, each node individually makes its decision about the placement of the components. In the cooperative method, all processors are working toward a common system-wide goal, avoiding each entity trying to maximize just its local performance (in an egoistic way).

4.2.1.3 Desirable Features

The following features (among others) are desirable in a global scheduling algorithm [118]:

- No a priory knowledge about tasks.
- Be able to deal with dynamically changing load. For that, it is necessary the existence of migration mechanisms in order to rearrange the tasks in the system in response to a new system load.
- Quick decision-making capability, in order to respond promptly to new situations in the system.
- Balanced system performance and scheduling overhead. Several scheduling algorithms collect global information to make their process assignment decisions. This may impose a high burden in the system.
- Stability. The system should not spend all the time making migration and calculating new assignments without accomplishing any useful work.
- Scalability

From this list, we identify that a static scheduler can not meet some of them. Therefore, static schedulers have a restricted applicability in real systems. Nevertheless, in our approach, we use a static optimal scheduler that uses the current snapshot of the system in order to evaluate the results coming from our dynamic assignment heuristic.

4.2.1.4 Static Scheduling

A static scheduler makes the decisions just with information available at compilation time. Information regarding task execution times and processing resources is assumed to be known at compilation time. A task is always executed on the processor to which it is assigned [115].

There are several alternatives for the objective function of the scheduling problem. Nevertheless, typically, the goal is to minimize the overall execution time of a concurrent program while minimizing the communication delays.

Two distinct models of parallel programming have often been considered in the context of static scheduling: the task interaction graph (TIG) model and the task precedence graph (TPG) model [80].

In the task interaction graph, the vertices represent parallel processes and edges denote the inter-process interaction [17]. It is usually used in static scheduling of loosely coupled communication processes (because it assumes that all tasks are executing in an independent and simultaneous form). Normally, the objective of scheduling is to minimize parallel program completion time by mapping the tasks to the processors. For that, the balance of the computation load among processors while keeping the communication costs as low as possible are the goals [80].

In the task precedence graph model, the nodes represent the tasks and the directed edges represent the execution dependencies and the amount of communication. It is mainly used by tightly coupled tasks on multiprocessors. A very good overview of the different task models and different mapping problems can be found in [98].

In the section 4.3, we formally present the optimization problem describing our service distribution. For that, we use a modified version of the task interaction graph model.

In the next paragraphs, we will review some theoretical analysis of the task assignment problem.

Stone [124] presents a method for assignment on a two-processor system based on a Max Flow/Min Cut algorithm for sources and sinks in a weighted directed graph. This method finds the optimal placement. Lo [90] extends the Stone model in order to increase the concurrency introducing the interference costs among tasks in addition to the Stone model that just takes the communication and the processing necessity of each task in account. Moreover, they present a heuristic that combines the Max Flow/Min Cut algorithm with a greedy-type algorithm to find suboptimal assignments of tasks to processors. Price and Salama [102] describes three heuristics for assigning precedence-constrained tasks to a network of identical processors. Other approaches are presented in [103, 112].

Although the presented theoretic analysis can achieve a good static schedule, those methods do not take in account the network topology, just the communication graph and the execution time of the tasks. In our approach, the main metric is the communication cost carried by tasks/service communication in a point to point network.

In the literature there are also approaches that take into account the communication and computation time of the tasks together with the connection topology of the processors (with the bandwidth of those connections). In [126], a graph-theoretic formulation of the problem of mapping communicating tasks to processors is presented. A heuristic algorithm is introduced as well. This algorithm takes as input a task graph where the vertices represent the tasks and the edges represent the communication between tasks and a resource (network) graph where vertices represent nodes and edges the links of the network. The algorithm outputs the mapping from tasks to processors. They cluster the heavy-communicating tasks and assign these clusters to processors. A greedy heuristic that uses a similar model of the communicating tasks and network of processors is presented in [100]. [72] presents a heuristic that first places the highly communicative tasks on adjacent nodes of the processor network. Then, the remaining tasks are placed beginning from those close to this *backbone* from tasks. Another branch of studies on task scheduling in heterogeneous environments [48, 51] is done based on scheduling DAGs, in which a task graph represents dependencies between tasks. Those methods are more appropriate for set of tasks with small intercommunication.

In our approach, we are using a dynamic distributed non-cooperative scheduling strategy, i.e., the current state of the system is used in order to drive the migration of the services in the system. Moreover, each service is an autonomous agent that by itself takes the decisions when to migrate and to which node. Nevertheless, in the evaluation of our heuristic, the models introduced in the theoretic static analysis are useful in order to formulate instances of the problem and to compare the generated results with the optimal assignment.

4.2.1.5 Dynamic Scheduling

In this section, we will describe some details about systems using a dynamic scheduler and also present some approaches.

The dynamic scheduling is based on the redistribution of processes among the processors during execution time (on-the-fly). This redistribution is performed by transferring tasks from the heavily loaded processors to the lightly loaded ones with the aim of improving the performance of the application (by some metric) [115]. Common used metrics are minimizing the execution time of an application, maximizing the system throughput or maximizing the processors utilization.

There are basically two main approaches concerning systems with dynamic scheduling.

Dynamic Load-sharing approach: attempts to conserve the ability of the system to perform work by assuring that no node is idle while processes wait for being processed [118]. We will call this approach simple as load sharing.

Dynamic Load-balancing approach: this is a stricter form of load sharing, wherein the system strives to balance the load on all machines at all times [55]. We will call this approach simple as load-balancing.

In this section we will present approaches that are from both types.

A loading sharing/balancing algorithm can be separated in the following components based on the four independent functionalities [55]:

Transfer policy: Determines whether a node is suitable for process transfer either as sender or as receiver

Selection policy: Selects a process from the queue to migrate

Location policy: Finds a suitable partner of the migration among the potential sender/receiver recognized by the transfer policy.

Information policy: Responsible for gathering system state information to be used in allocation decisions.

Now we will describe some approaches of dynamic scheduling that share some characteristics with our heuristic for service distribution. The highlighted term before each approach stress some share characteristic with our heuristics and will be better explained in the discussion section.

Autonomous Mobile Entity In [81], the authors present an approach where each process is an autonomous entity that determines by itself the best location for the placement. Every time that a task is started, a agent in behalf of the task observes the load in each machine of the system (or a random subset) and selects the best one to place the process. In the paper, just the initial placement is analyzed.

Forces Attracting Entities In [60] a decentralized dynamic load balancing mechanism based on forces that correspond to independent optimization goals is proposed. The algorithm explicitly considers communication and migration overhead. The algorithm is inspired in a physical model that uses notion of forces in fluids: in a flat container with even bottom, different amount of non-mixable fluids are placed at different positions. Gravity forces the fluids to run out, but frictional resistance and cohesion forces are working against. Thinner fluids tend to spread out, where more viscous fluids stick together. After some time, the fluid distribution will reach some stable state (balanced forces).

The correspondence to the load balancing problem is the following. The tasks in the parallel computation are considered as particles of the fluid. Load potential of each node is the gravity force that attracts those particles. Communication relations along with their intensities are associated with a cohesion force, with direction and magnitude. The frictional resistance is associated with the migration costs and are together, with the cohesion, working against the load balancing. Using this model, the algorithm pursues the following objectives: minimization of load unbalance, minimization of communication costs, avoidance of unproductive migration and stability.

Every time when the load situation changes, all neighboring nodes are informed and the forces described are calculated. A resulting force is calculated by a linear combination of the components. The higher force is elected to initiate a migration. After this migration, the algorithm starts again, what could bring a domino effect. The algorithm is a sort of *distributed gradient search*[60] which converges into a local minimum. The authors argue, as the landscape of the objective function is always changing, this is not a big drawback.

Greedy Approach Furthermore, a greedy distributed load sharing algorithm is proposed by [34]. The system load is used to decide where a job should be placed. The decisions are made for the local goodness of a job and the assignments are always accepted.

There are several other approaches. A broad survey on distributed scheduling can be found in [115, 30].

4.2.2 Migration of Service in WSN

In this section, the specific approaches of migration of services in wireless sensor networks will be described. Although they are comparable and some belong to the dynamic scheduling presented in the last section, we decide to introduce them in a separated section because the objective of the distribution may deviate from the traditional dynamic scheduling algorithms. There are approaches that even the distribution objective is decided by each task (or *agent*), and different *agents* may have different policies.

As already discussed, the allocator is responsible for the assignment of the tasks to the nodes. A range of middleware and virtual machine approaches for WSN present different kinds of task allocation methods. Nevertheless, at this moment, the majority of operating systems for sensor networks do not provide an appropriate task allocation mechanism.

Most of the task allocation mechanisms used in WSNs are online, i.e. they decide during run-time where to place the software components. This means that the code mobility is necessary for such approaches.

The table 4.1 shows some of the task allocation techniques used in VM and middlewares. In the Sensorware [22] virtual machine, the application consists of scripts that are deployed on a subset of nodes of the network. Each script looks like a state machine that is influenced by external events. The scripts can replicate themselves, i.e, the application has the control of the task allocation. This means that each agent may have a different strategy or even the strategy may be adapted to the current environment. The application programmer is responsible to select and implement the allocation strategies. There isn't a standard migration police.

MagnetOS [9] uses the online, power-aware algorithms called NetCenter and NetPull to decide where to place a system component based on the communication pattern. While NetPull profiles the communication at the link level, migrating to the direction of the highest amount of communication, NetCenter operates at network level migrating to the host where the object that it communicates at most resides. Those heuristics try to allocate tasks dynamically in order to reduce the communication overhead, diminishing with that the average energy consumption of the network.

In Cougar [136], the queries are broadcasted to the nodes of the network and the results are aggregated and forwarded to a given leader node. The query optimizer, located on the gateway node, is responsible to analyze the query and generate a good query execution plan, which contains the data flow inside the node and network. A query plan contains, for example, which node will be the leader and which sensor reading, aggregation and forwarding will be realized at each node of the sensor network. A complex query can be composed by a large number of parameters and operators. This brings to a larger space of querying processing plans, and a query optimizer is responsible to select a good plan using some objective function, like energy usage. As the actual approach of the query optimizer relies on a centralized node that calculates the query plans, this cannot be compared with our distributed service distribution algorithm.

SINA [114] uses the attribute matching approach to select the nodes that will process the received SQTL script by means of forwarding it to an running application on the node. Each SQTL message has a SQTL wrapper that indicates, by means of attributes, which nodes should receive and forward

Technology	Approach	Scalability	Requirements	Benefits	Drawbacks	Used by
Script population specification	Specification in migrating scripts	Scalable	Control in application scripts	No control required	Expressivity of specification	Sensorware (VM)
Automatic object Placement	Activating and moving objects new to source	Scalable	Object placement algorithm	Reduced data communication	Complexity, overhead (1-hop migration)	MagnetOS (VM)
Query Optimizer	Optimizing query routing to network	Optimization in gateway node	Disseminated query plan	Only required subset of nodes activated	Network load of query plans	TinyDB, Cougar
Attribute matching	Matching script attributes with local parameters	Scalable	Attributed Specifications	Local late binding	Restricted expressivity; just suitable for query-based systems	SINA
Automatic fusion point placement	Distributed heuristic for fusion point placement	Initial placement of fusion point (tree structure)	Scalable	Dynamic, reduce data communication	Specific for data fusion, overhead (1-hop role change)	DFuse

Table 4.1: Task allocation solutions used by middleware / virtual machines

the script to applications running on the node. When information gathering is required, the front-end node is responsible to select an appropriate method to collect the results of the query based on the network state and the nature of the query. For example, in the diffused computation operation mode, the attributes of the SQTL are not only responsible to select which nodes are replaying to a given query and which sensor values should be sampled in the nodes, but also the aggregation logic is programmed in the SQTL script. After the dissemination of the script, each node knows how to aggregate the information and route it to the frontend.

The automatic fusion point placement [78] is responsible for the node role assignment in the DFuse middleware. The role assignment in the case of the DFuse is the mapping from a fusion point in an application task graph to a network node. The existing roles in the DFuse are: *end point* (source or sink), *relay* (node that routes the request) or *fusion point* (node that accomplish the fusion task).

The role assignment heuristic has two parts. In the first part, an initial naive assignment is calculated in order to initiate the transmission from the sources to the sink. The second part is the optimization phase. In this phase, every fusion point node can decide locally if it wants to transfer the role to any of its neighbor nodes. The decision for role transfer is taken based on local information. The fusion node periodically informs its neighbors about its role and its health. The health is an indicator of how good is a given node to host the fusion role. Upon receving the message, the neighbors calculate their own health. If some neighbor determines that it can play a better role than the sender, it informs the fusion point sending its own health. The actual fusion point selects from the coming healths, the best neighbor to migrate the role.

There are several functions to calculate the heath of a given node. Each one has a different objective. Examples are *minimize the transmission cost*, *minimize power variance*, *minimize ratio of*

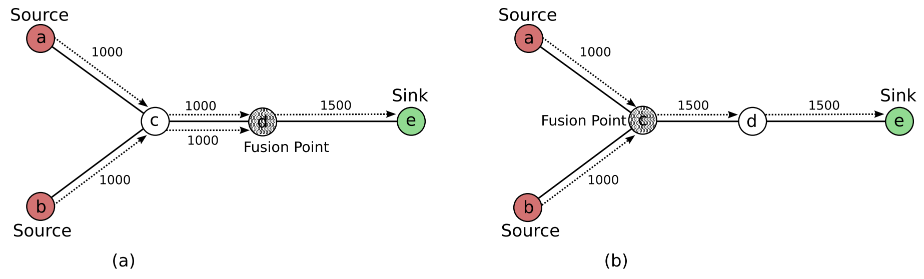


Figure 4.2: Example of a linear optimization. In (a), two sources are generating data (nodes *a* and *b*). The data stream coming from the sources are relayed by node *c* and data fusion occurs in *d*. Finally, node *e* receive the aggregated data. In (b), node *c* has a better health than *d* because if the fusion point will be placed on it, the *fan-in* flows (in the example, two flows) will have a reduction of one hop, whereas the *fan-out* flow (in the example, just one flow) will increase its path by one hop. Therefore, a so called linear optimization occurs with the transference of the fusion point from *d* to *c*, saving 500 "communication units" $\times$ 1 hop.

transmission cost to power. The health function that aims the minimization of transmission cost calculates the future communication cost that will be generated if the role is transfered to the candidate neighbor. This cost is measured using the traffic of the incoming flows multiplied by the distance (in hops) traveled added to the traffic of the outcoming flows multiplied by the distance (in hops) traveled. Smaller health means less communication cost and it is preferable. If the transfer of the fusion role from the actual fusion node to the neighbor (candidate) will reduce the amount of traffic (because the neighbor has a smaller health value), it is realized.

Two types of role transfer are mainly induced by the heuristic, called linear and triangular optimizations. They are depicted in the Figure 4.2 and Figure 4.3. In the linear optimization, all the inputs of a fusion point are coming via a relay node. There is data contraction (for details, see section 2.3.3.4) at the fusion point. Therefore, the relay node becomes the new fusion point. The fusion point is moving away from sink and coming closer to the data source points. In the case of data expansion, the fusion point tends to go to the direction of the sink.

In the triangular optimization, there are multiple paths for inputs and outputs to the fusion point, and data contraction is being realized. The fusion point will move along the input path the maximize the savings. An example is shown in the figure 4.3.

4.2.3 Discussion

Several approaches have been studied to deal with the global scheduling problem. In the static scheduling, the assignment of the tasks to the processing elements is done at compilation time. The minimization of the program's completion time together, as secondary objective, with the minimization of communication delays is the most common optimization objective in this kind of scheduling. This is different from our problem; we aim to minimize the communication overhead, which is measured by means of amount of data transported over a given distance (link metric). The general problem of the static task assignment is NP-complete, even when the communication delays are not accounted [115]. This is also true for our different formulation of the problem, described in the section 4.3.

There are several models to describe the static assignment. For our work, the most important is the task interaction graph (TIG). We use the TIG to model our system in a given instant of time (system snapshot), and a problem very similar to the static assignment is solved in order to calculate

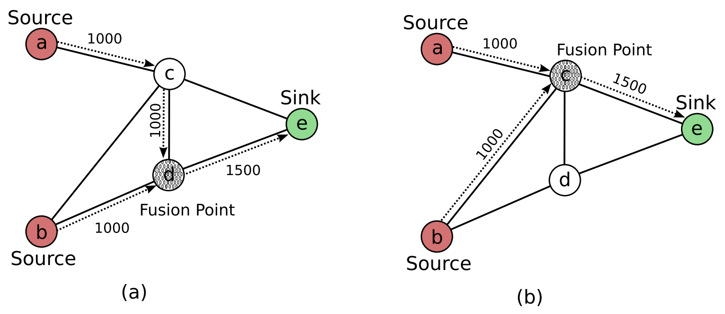


Figure 4.3: Example of a triangular optimization. In (a), two sources are generating data (nodes *a* and *b*).Node *c* is relaying the data coming from *a*, while *d* is resposible for the data fusion. Node *e* is the sink and it receives the fused data. In (b), node *c* recognizes the its direct link to node *b* and *e*, having a better health than node *d*. This triggers the so called triangular optimization and the fusion point is transfered to node *c*, saving 1000 "communication units" $\times$ 1 hop.

the optimum assignment at that point of time. This value is compared to the result of our heuristic and it is possible to assess how good it is performing. It is possible to test whether our heuristic, under a given input, converges to a result that is not so far from the optimal solution.

Due the fact that in the static scheduling, tasks must be executed in the node decided by the initial assignment and all system details should be known before the scheduling is calculated, they cannot be used in a dynamic environment such as a sensor network.

Our heuristic is similar to a dynamic distributed non-cooperative scheduling strategy. There are several approaches that have common concepts with our heuristic, described later in this chapter. In our heuristic, the services are the migrating units. Each service decides by itself when to migrate and to which node it should be transferred (new placement). This has some similarities to the work of [81], where each process is an autonomous entity that selects by itself the best placement. Nevertheless, the topology of the network is not observed, just the load of the nodes.

In [60], a concept of forces is used, which has some similarity with our approach. In our heuristic, the pheromone deposited by the communication traffic in the nodes acts as a force that attracts the services of the system. Nevertheless, in the approach of [60], the load potential is considered as the attraction element. In the sensor network, the load is not the main factor to be analyzed. Moreover, they are just analyzing neighboring loads, what turns such an approach useless in our WSN environment. In addition, there is no concern about the different nature of the WSN network and its applications, which differ from conventional ones.

Our heuristic has some greedy characteristics like the work presented in [34]. Nevertheless, again the system load is the main parameter used to guide the migration.

As we can see, although the dynamic load balancing/sharing share some characteristics with our heuristic for service distribution, the main idea of most of the algorithms is to share the load among nodes and not to minimize the network communication between task and OS services as our method. In addition, they are designed to be used in an infrastructure network with traditional applications, opposite to our approach. For example, the link quality and the dynamic topology of WSN are not considered by any of the algorithms.

Other approaches of task allocation focused in the requirements of sensor network applications are presented in the section 4.2.2. A possible classification of the approaches considers whether the location policy resides in the system software or in the application.

In the Sensorware approach, the location policy is implemented by the applications. Each application (or part of it, a script) may decide as an autonomous entity when to migrate or to copy itself (replicate) to the neighbors. Each script contains the information of how it will populate the nodes of the network after the initial injection at the access point. This process of populating the nodes can continue depending on events and the current state. This is radically different from our approach. In our system, the operating system controls the migration of services trying to optimize a given objective function. Different from Sensorware, every service in our system follows the OS location policy. This means that the user, different from Sensorware, are not overloaded with the responsibility of writing a location policy in each application.

The NetPull and NetCenter placement algorithms from the MagnetOS, like our heuristic, promote a transparent migration of parts of the system in order to reduce the communication overhead. Like the NanoOS, the location policy is implemented by the system software. Differently from our system, NetCenter transfers the system components directly to the node hosting the object with the highest interaction. This may bring a non-optimal placement, due the fact the the sum of the communication coming from other objects residing in different nodes may easily exceed the communication traffic generated by the single component with highest communication interaction.

Differently, the NetPull algorithm moves a system component in to the direction of the neighbor from where the highest communication traffic comes from. When we just allow one hop migrations (parameter $\mathsf{allowed\_h} = 1$), the behavior of our basic heuristic has some similarities with the NetPull algorithm. However, instead of dividing the time in epochs and profiling the communication pattern within one epoch and deciding about the migration based on this epoch result, we use a pheromone level to describe the communication patterns. New communication events are changing the pheromone on the node continuously, and the evaporation rate is erasing this pheromone also in a continuous way, reducing the importance of old measures. This brings to a smoother mode of operation, without sudden changes based on the events occurring in a very loaded epoch. Moreover, when the service migrates to the next hop, the pheromone trails are already deposited in the neighboring nodes of this new host, i.e., a completely new communication profiling is not necessary.

Moreover, our basic heuristic makes the exploration of a given number of hops before selecting one node that is the target of the migration (parameter $\mathsf{allowed\_h} = n$). For that, we use a potential pheromone concept to simulate the communication pattern that could occur if the service migrates to the node being analyzed at the moment. This multi-hop migration has several advantages. First, it avoids a large overhead of migrating the service several hops in sequence. Second, in NetPull, if the neighbor selected as destination of the migrating component hasn't enough resources for it, the migration cannot take place. A blocking situation occurs. An additional point is that we analyze also the amount of resources of the nodes in the migration path and their neighbors to select a more appropriate node. This means that not only the communication, but also the actual resource availability of the nodes is evaluated in order to select the destination of the service.

It is not necessary to compare NetPull with our extended heuristic, because it encloses all characteristics of the basic one. Beyond this, the extended heuristic has several additional mechanisms to enhance the module placement that are not present in the NetPull algorithm.

Another algorithm that share propreties with our developed heuristic is the automatic fusion point of the DFuse middleware. Similarly to our approach, the neighborhood of the node containing the fusion point is examined for its migration. The neighbor with the highest saving potential is selected to receive the fusion service. The best next position of the fusion point depends on the operation realized (contraction or expansion). For contraction, places near the source may save communication and for expansion, nodes near to the sink tend to reduce the communication. The two kinds of transfer mainly induced by the heuristic were presented in the section 4.2.2. The linear optimization can be encountered in our heuristic, in the both versions. But different from the automatic fusion point, where

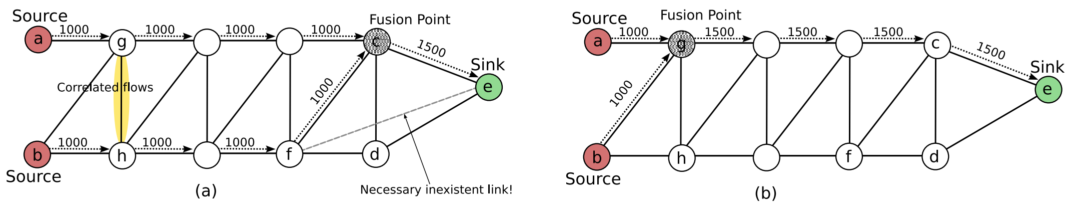


Figure 4.4: Problem of the triangular optimization.

the flows are constant and known, and the distance from every node to the sink and source must be known, we realize the linear optimization with variable flows (measured by the pheromone levels) and there is no necessity to know the distance from the neighbors to the requesters and providers. Moreover, similarly to NetPull, just hop-by-hop migrations are allowed by the automatic fusion point placement, which brings a large overhead when the migrating module is large.

A second kind of optimization generated by DFuse heuristic is the triangular optimization. It uses the knowledge of links not being used at the moment for communication. It changes the fusion point saving one hop of communication (see figure 4.3). Although it has the potential to reduce the communications cost, this potential is limited, in contrary to a more powerful concept introduced in our extended heuristic. In Figure 4.4, we depict the limits of the triangular optimization. In the figure, we present a scenario where multiple triangular optimizations would be stimulated. Although such a topology has very low probability to occur in a geometric random graph, it would be a stimulating topology for multiple triangular optimizations. We aim to have a chain of such optimizations. In 4.4(a), we can see that this, in fact, is not possible. After the first optimization (from node *d* to *c*, similar to the optimization of the Figure 4.3), it is not possible to realize another triangular optimization. This happens because a link to the sink node (or to the relay node receiving the output of the fusion from the first optimization) is necessary for further triangular optimizations. In the case of the figure, a link from node *f* to *d* is needed for the next optimization. Such a link cannot exist, because when it will exist, the node *d* would never be part of the tree generated to deliver the data from the sources to the sink. DFuse uses a bimodal link model. Therefore, no chain triangular optimizations can occur in the presented scenario.

Our concept of correlated flows, presented in the section 4.4.2.1, can cope with such situations in a much better way. The two flows, coming from the source in node *a* and *b* would be recognized as correlated by nodes *g* and *f* of the figure 4.4(a). Due this correlation, they would act together to atract the migrating module to their “direction” in the network. In the figure 4.4(b), the resulting migration if our extended heuristic would be used in the DFuse is shown. As can be seen in the figure, with our concept, a much larger migration with increased savings is possible.

Another important point is that the fusion point migration does not profile the communication, because the user must enter the data flow graph and the operation (contraction or expansion) is already know. Not less important to remark, the automatic fusion point placement is designed only taking in mind data fusion applications, in contrast to our generic service migration.

An additional important point is that both DFuse and NetPull use a bimodal link metric for routing and distance calculations, which could yield to unfavorable situations, where bad links are used for communication or migrations, decreasing the system performance.

4.3 Problem Definition

In our approach we are optimizing the position of the services of the system through *migration*, i.e., we try to find the optimal configuration where the communication overhead caused by the remote requests is minimized.

For a determined system configuration, an assignment of the services to the sensor nodes exists, where the total communication cost of the assignment is minimized. This optimal service allocation stays only valid when there is no modification in the configuration of the application/services (in terms of new services being created or some being extinguished, variation of the communication pattern and network topology changes). We will call this situation as *stable configuration phase*.

We developed an heuristic that dynamically re-assigns the services in the system in order to reduce the communication overhead. After some interaction, if the network stays in a *stable configuration phase*, the reassignment of the services will achieve a stable situation with some communication cost (the system will converge to a certain configuration). For evaluating our heuristic, we define the problem to be solved in each *stable configuration phase* as an optimization problem.

The formal optimization problem is described here.

The system is represented by two graphs. The first is the network (resource) graph and the second one is the processing thread (task/service) graph (similar to the task interaction graph).

The ad hoc network is modeled by an undirected graph $G = (V, E)$, where V is the set of wireless nodes and a edge $\{u, v\} \in E$ if and only if a communication link is established between node $u \in V$ and $v \in V$. The two nodes in this case are neighbors.

For each link, a weighting function attributes a positive weight. $w : E \to \mathbb{R}^+$. This weight measures the quality of a wireless link (for details, see *virtual distance* concept, section 3.5.3.5). We define for each edge not in the graph ($\{u, v\} \notin V$), $w(u, v) = \infty$.

For each node, an additional weighting function r is responsible to characterize the amount of resources available in the node. $r : E \to \mathbb{R}^+$. This models the resource capacity of the node.

The processing thread (task/service) graph $T = (M, C)$ models the communication requirements between the diverse processing threads of the OS and application. M is the set of tasks and services (processing threads) running at the moment in the system and an edge $\{m_1, m_2\} \in C$ when there exist a interaction (with communication) between the executable units m_1 and m_2.

For each interaction $c \in C$, a function b attributes a positive weight that measures average of traffic between the tasks/services. $b : C \to I\!R^+$. This function defines the amount of interaction between two modules of the system.

Moreover, the function $e : M \to I\!R^+$ attributes the amount of resources necessary for the execution of each task/service.

Finally, the function $f : M \to V$ defines the fixed assignment, i.e., the tasks that are fixed assigned to a determined node and should not be moved.

The *service allocation in ad hoc network problem* consists of allocating the tasks and services of the task graph T in the nodes of the netwok graph G, minimizing the amount of communication. The amount of communication is measured by the sum of all products of the amount of communication by the distance of the communicating entities. This distance is measured in terms of our link metric.

A schematic diagram of the input and result of the allocation is shown in the Figure 4.5.

More formally, we describe our system as the following optimization problem:

Input: A processing threads (tasks and service) graph with weighted nodes, weighted links, and fixed assignment function (T, b, e, f) and a network graph with weighted nodes and links (G, w, r)

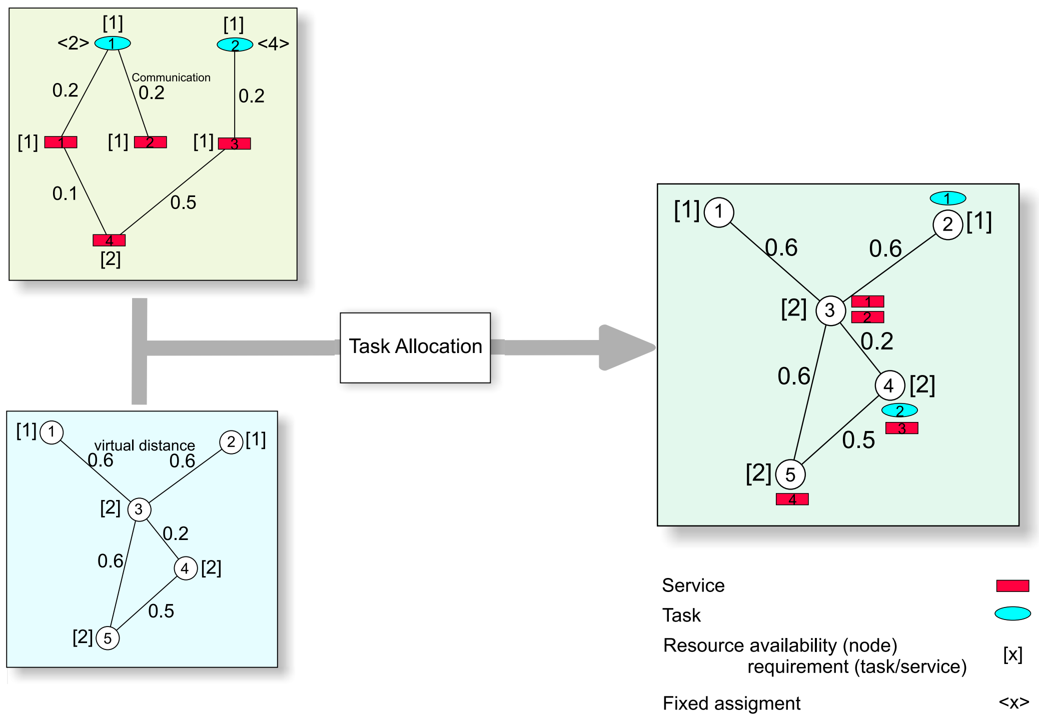


Figure 4.5: Example of an instance of process allocation problem.

Constraints: For every input instance (G,w,r,T,b,e,f),

Let $S=\{s_1,s_2,..,s_n\}=\{s\in M|f(s)=\emptyset\}$ be the set of mobile services (without a fixed assignment)

The valid solution space is given by:

$\mathscr{M}(G,w,r,T,b,e,f)=$

$=\left\{(g_1,g_2,..,g_n)\in V^n|\forall v\in V,\sum_{\{i\in\mathbb{N}|g_i=v\}}e(s_i)+\sum_{\{m\in M|f(m)=v\}}e(m)\leq r(v)\right\}$

The tuple $(g_1,g_2,..,g_n)$ is an assignment and has the following meaning: service s_i is assigned to node g_i. The constraint assures that the services and tasks assigned to the node v do not request more resource than the availability on the node.

Costs: For every assignment $(g_1,g_2,..,g_n)\in\mathscr{M}(G,w,r,T,b,e,f)$, the cost is calculated as follows:

Let the function $q:M\rightarrow V$ be:

$$q(m\in M)=\begin{cases} f(m) & \text{if} \quad f(m)\neq\emptyset \\ g_i|s_i=m & \text{otherwise} \end{cases}$$

$$cost\left((g_1,g_2,..,g_n),(G,w,r,T,b,e,f)\right)=\sum_{\{m_1,m_2\}\in C}b(\{m_1,m_2\})\cdot D\left(q(m_1),q(m_2)\right) \tag{4.1}$$

Where $D(u,v)$ is the virtual distance between nodes $u,v\in V$.

Now, we define how the virtual distance is calculated.

Let $P(u,v)=\left\{p_1^{(u,v)},p_2^{(u,v)},..,p_j^{(u,v)}\right\}$ be the set of all possible paths between nodes $u\in V$ and $v\in V$.

$p_h^{(u,v)} \in Pot(E)$ is the h^{th} possible path where:
$p_h^{(u,v)} = \{\{u,x_1^h\},\{x_1^h,x_2^h\},..,\{x_{k-1}^h,x_k^h\},\{x_k^h,v\}\}$, $x_i^h \in V$, $i = 1,2,..,k$, $k \in I\!N$

Now, we introduce the cost of a path:

$$PCost(p_h^{(u,v)}) = w(u,x_1^h) + \sum_{i=1}^{k-1} w(x_i^h,x_{i+1}^h) + w(x_k^h,v)$$

The virtual distance between u and v is the cost of the smallest path:

$$D(u,v) = PCost(p_h^{(u,v)}), \text{ where } PCost(p_h^{(u,v)}) = \min_b \left(PCost(p_b^{(u,v)})\right), \text{ for } b = 1,2,..,j$$

Goal: *Minimum*

The problem is NP-complete (for a similar NP-complete allocation problem, see [49]), since it generalizes the well-known NP-complete quadratic assignment Problem (QAP) [110]. The QAP is a special case of our problem when the services are in the same number as the processors and just a single service (anyone) may be assigned to each processor.

4.4 Ant Based Service Distribution

In this section our heuristic to distribute the services in the sensor (or ad hoc) network will be presented.

4.4.1 Basic Heuristic

In our approach, we are optimizing the position of the services in the system through *migration*, i.e., we try to find the optimal configuration where the communication overhead caused by the remote requests is minimized. In order to solve this online discrete optimization problem, we decide to use an ant inspired algorithm that is described in this section. It is relatively simple and has shown good performance.

We assume, in our heuristic, that a initial distribution of the services in the network already exists. This initial distribution is achieved through the service instantiation method of the service manager layer of the NanoOS. This is explained in the section 3.4.6.2.

Given this initial distribution, the heuristic described here is responsible to re-distribute (migrate) the services in order to react to new demands or topology changes.

In order to describe our heuristic, some additional definitions are necessary.

The set P contains the types of all possible services of the system. Each service s is an instance of some type $p \in P$. Every task $a \in \{M - S\}$ has no type.

Let $r \in M$ be the requester (a service or a task) of some service $s \in S$. The service state S_r^i represents the connection between the requester r to the service s (a flow of communication, generated by the requests and responses). The set of all flows of the system we will call W.

In our system, each node $v \in V$ has a pheromone table $P_v = [p_{S_r^s}^v]_{r \in M, s \in S}$, where $p_{S_r^s}^v \in [0,1]$.

This pheromone level represents the request rate (and traffic) made by the requester r to the service i that is crossing the node v. In our approach, all nodes are responsible for the service distribution, since each node's evaluation is based on its *local* view. Moreover, the needed information is constantly changing, due to frequent pheromone updates so that transferring the decision to just certain nodes would incur an high additional communication overhead with questionable efficiency gain.

Using an analogy with the ant foraging behavior [19], the services in our approach are the equivalent of the food source. The calls made by the requesters are the agents (or ants) and the requesters are the nests. The wireless links form the pathway used by the ants. While the requests are being routed to the destination service, they leave pheromone on the nodes.

The pheromone tables in each node are updated according to the equation:

$$p_{S_r^i}(t+1) = \frac{p_{S_r^i}(t) + \delta p(h)}{1 + \delta p(h)} \tag{4.2}$$

where the $\delta p(h)$ is the variation of the pheromone and it is a function of the size of the packet.

After the introduction of some basic concepts of our heuristic, we will present here the component policy of our migration mechanism:

Transfer policy: In our heuristic, each service is independent and may decide itself the moment of starting a migration. Therefore, there is no pre-selection of nodes that will start a migration. The target of a service migration is every node that has enough resources to accommodate the incoming service.

Selection policy: There isn't a node-wide selection of which service should migrate in our system. As already said, this is a individual decision of each service. It is based on a threshold θ that is compared to the measure of the current communication overhead of the service (we denote here $\gamma_{S_r^s}$, the communication overhead brought by the interaction between the requester r with the service s). Each packet coming from the requester r to the service s brings the virtual distance traveled. Multiplying the size of the packet by the traveled distance, the communication cost of the packet is calculated. γS_r^s is the sum of all communication costs from r to s in a given interval of time t_m. For a service s, if $\sum_{r \in M} \gamma S_r^s > \theta$, the service s is selected to migrate.

Location policy: The main part of the heuristic is about the location policy, i.e., which node should receive a migrating service. This will be described in the next section.

Information policy: We use in our heuristic almost just passive information gathering. In several load balancing algorithms, there is a active broadcast of the current status of a node, for example, informing that the node is idle. We avoid this approach in order to save the scarced energy resource. The gathering of information is made in the form of pheromone tables in the nodes of the network.

Now, we will describe our location policy in detail.

The general idea is to migrate the service to some node that rely in some requests flow (path) or near to it, in the direction of a requester. Each service has several flows coming from the diverse requesters.

In order to determine which node should receive the service s, an explorer packet will be used. The explorer packet is just a special packet that travels through the nodes of the network. The next hop is defined based on the pheromone value of the neighborhood. The final location of the exploration packet will eventually be the target node for the migration of s.

4.4.1.1 Exploration Packet

The explorer packet has the following fields:

packet_id: The explorer packet identification

service_id: The identification of the service

source: Node actually hosting the service s, from where the packet originally comes.

allowed_h: The number of hops that the packet may still migrate

history: A ordered list of the visited nodes. Let $l \in V$ be the last node visited by the explorer packet.

potential pheromone: Stores the potential pheromone value (will be described later). We will use the term $p_{potential}$ to describe it.

In the initial situation, the node that actually hosts the service s receives the explorer packet *pak* with allowed_h=k. The history field is empty and the potential pheromone is 0.

Now we will describe the two main phases of the selection of the new target node to the service s through the migration of the exploration packet. The first part is called exploration phase and the second one settlement phase.

4.4.1.2 Exploration Phase

Upon receiving the packet, a node decrements the counter and verifies whether allowed_h=0, which means that the heuristic should execute the settlement phase and no more new nodes exploration is done. If allowed_h$>$ 0, the next destination of the packet will be selected using the following procedure:

Let $u \in V$ be the actual location (node) of the explorer packet. Ngh_u is the set of neighbors of u, and $d \in Ngh_u$ is a neighbor of u.

$$b^s_{u,d} = \begin{cases} \frac{\sum_{x \in M} p^d_{S^s_x}}{\sum_{y \in (Ngh_u - l)} \sum_{x \in M} p^y_{S^s_x} + pot\_pher} & \text{if} \qquad d \neq l \\ \frac{pot\_pher}{\sum_{y \in (Ngh_u - l)} \sum_{x \in M} p^y_{S^s_x} + pot\_pher} & \text{otherwise} \end{cases} \tag{4.3}$$

$b^s_{u,d}$ represents the sum of the pheromone of all flows coming through node d to the service s normalized over the total amount of pheromone related to requests to the service s in the neighborhood. It represents relatively how much of the traffic directed to the service s is using the node d as path (proportional use of d for the requests). The $b^s_{u,d}$, in the exploration phase, will act like a force attracting the exploration packet to the corresponding node.

The potential pheromone field is used to store the sum of all other pheromones related to the service s, coming from the neighbors not selected as next hop for the exploration packet *pak*. It will be used, during the travel of the exploration packet, to estimate the level of pheromone potentially caused by those flows if the service would migrate to the node being evaluated. An example can be seen in the Figure 4.6.

The main idea is try to forecast which situation would happen if the service would migrate to the current exploration packet position and which would be the next hop for a possible migration. The assumption made here is that the request flows not attended by the first migration decision would have their path size increased exactly by the pathway executed by the exploration packet. This means,

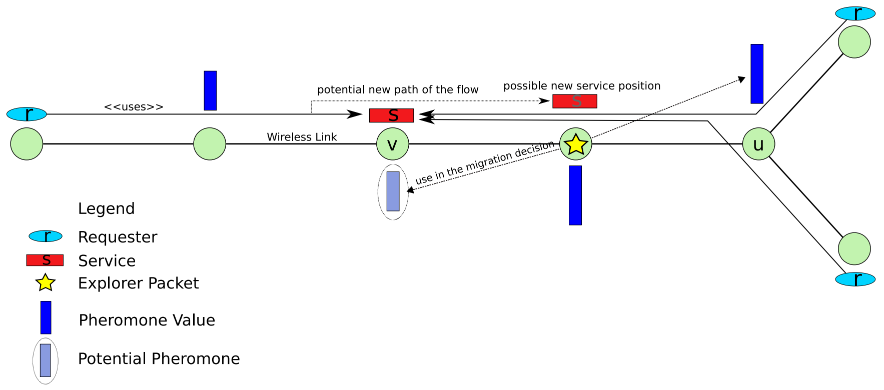


Figure 4.6: Example showing the new potential path of a flow when service would migrate to the next hop.

although the pheromone level from these flows would not appear to the exploration packet when far away from the node (v) hosting s, they should be considered when deciding the next exploration packet hop. This is shown in Figure 4.6, where the exploration packet is in the node u. It uses the real pheromone of the node j and, in the case of node v, the potential pheromone level measured by the first migration of the exploration packet. The potential pheromone level is the sum of all pheromone levels related to the service s that are in all other nodes than u, because u was selected as target for the first exploration packet migration. In this example, the potential pheromone level is exactly the same level of the pheromone on node h. It will be formally defined later on.

The next hop of the explorer packet is selected using the equation 4.4. Let's call j the selected node.

$$e_i = max_{\{d \in Ngh_u\}}(b^i_{v \to d}), d \in Ngh_u \tag{4.4}$$

If the selected node is the last visited node (i.e., $j = l$), the exploration phase ends and the settlement phase (see next section) is executed. Otherwise, the allowed_h field is decremented, and the current node is inserted in the history field. Moreover, if the exploration packet is in the same host of its correlated service, the field potential pheromone is updated using the equation given in 4.5.

$$pot\_pher = \sum_{\{h \in Ngh_u | h \neq j\}} p^h_{S^s_x}, \text{ for } x \in M \tag{4.5}$$

Finally, the packet is sent to the new destination j, where the exploration phase restarts.

4.4.1.3 Settlement Phase

After the exploration of possible candidates to host the service s, this phase is responsible to find the appropriated node with enough resources to host the service.

Let's call u the actual node of the exploration packet.

The idea of this phase is to proof whether there are enough resources in the candidate node to host the service s. In the positive case, the service will migrate to the node. In the negative one, the neighborhood will be checked and, according the neighborhood actual situation, a neighbor may be selected or the exploration packet may migrate to the last visited potential candidate (retrieved from the history field), to search there for the final destination of the service s.

The following procedure is executed in the settlement phase:

- The current node u is tested whether it may host the service s. The test consist of checking whether node u has enough free resources. The formalization of the test can be seen in eq. 4.6.

$$e(s) \leq r(u) - \sum_{\{m \in M | q(m) = u\}} e(m) \tag{4.6}$$

- If the resources are enough, the settlement phase is terminated and the node u sends a message to the service s to trigger the migration process.

- Otherwise, the same test is made in all the nodes of the direct neighborhood of u. The virtual distance is used for ordering the test process. Nodes within smallest virtual distance are tested first. The process ended when a suitable node is found, i.e., the node with enough resources and the smallest virtual distance to u is selected. Let's denote this node as f.

- If $w(u, f) < w(u, last(history))$, i.e., the virtual distance between u and f is smaller than the virtual distance between u and the last visited node by the exploration package (before reaching u), the node g is selected definitively to be the new host of s. A message is sent to s in order to start the migration.

- Otherwise, the exploration package is sent back to the $last(history)$ node. The node u is deleted from the history field and the settlement procedure starts again.

The procedure described above repeats until an appropriate node is found. In the improbable case of not finding any new node to host the service, there are two possibilities. The first is to cancel the migration. The second, when swap operations are desired, consist of swapping the service s with services hosted on the node where the exploration packet were where the settlement phase was started.

4.4.1.4 Example

In the Figure 4.7(a), a scenario with 9 nodes is presented. In this scenario, node g has a requester (r_1) of the service s_1 that is located at node v. At the same time, requesters r_2 and r_3 ,located at nodes b and c respectively, are accessing the same service. Let's assume that the pheromone related to the connection $S_{r_1}^{s_1}$ in the node h is $p_{S_{r_1}^{s_1}}^{h} = 0.3$, the pheromone of the connection $S_{r_2}^{s_1}$ in the nodes j, u, k is $p_{S_{r_2}^{s_1}}^{j,u,k} = 0.2$ and the pheromone of the connection $S_{r_3}^{s_1}$ is $p_{S_{r_3}^{s_1}}^{j,u,l} = 0.2$. After deciding to start the migration process, the exploration packet is launched on node v. According equation 4.3, force attracting the exploration packet to the node h is $b_{v,h}^{s_1} = \frac{0.3}{0.3+0.2+0.2} = 0.428$ whereas the force attracting to node j is $b_{v,j}^{s_1} = \frac{0.2+0.2}{0.3+0.2+0.2} = 0.571$. This means that the first step of exploration phase is to send the packet to the node j. Before that happens, the virtual pheromone field receive $b_{v,h}^{s_1} = 0.428$.

In the node j, the same analyses is made in order to determine the next hop. But instead of using the real pheromone levels on node v in the equation, the potential pheromone value carried by the exploration package is used. Now the packet is sent to the node u.

In this node, the situation is different. The force attracting the exploration packet to the node j is $b_{u,j}^{s_1} = \frac{0.3}{0.3+0.2+0.2} = 0.428$ whereas the force attracting to node l or k is $b_{u,l}^{s_1} = b_{u,k}^{s_1} = \frac{0.2}{0.3+0.2+0.2} = 0.285$. Therefore, the selected next hop of the exploration package is the node j. As node j has been already visited, the heuristic goes to the settlement phase.

The settlement phase is illustrated in the Figure 4.7(b). The first step is to check whether there is enough resource for the service in the actual position of the exploration package. In the example,

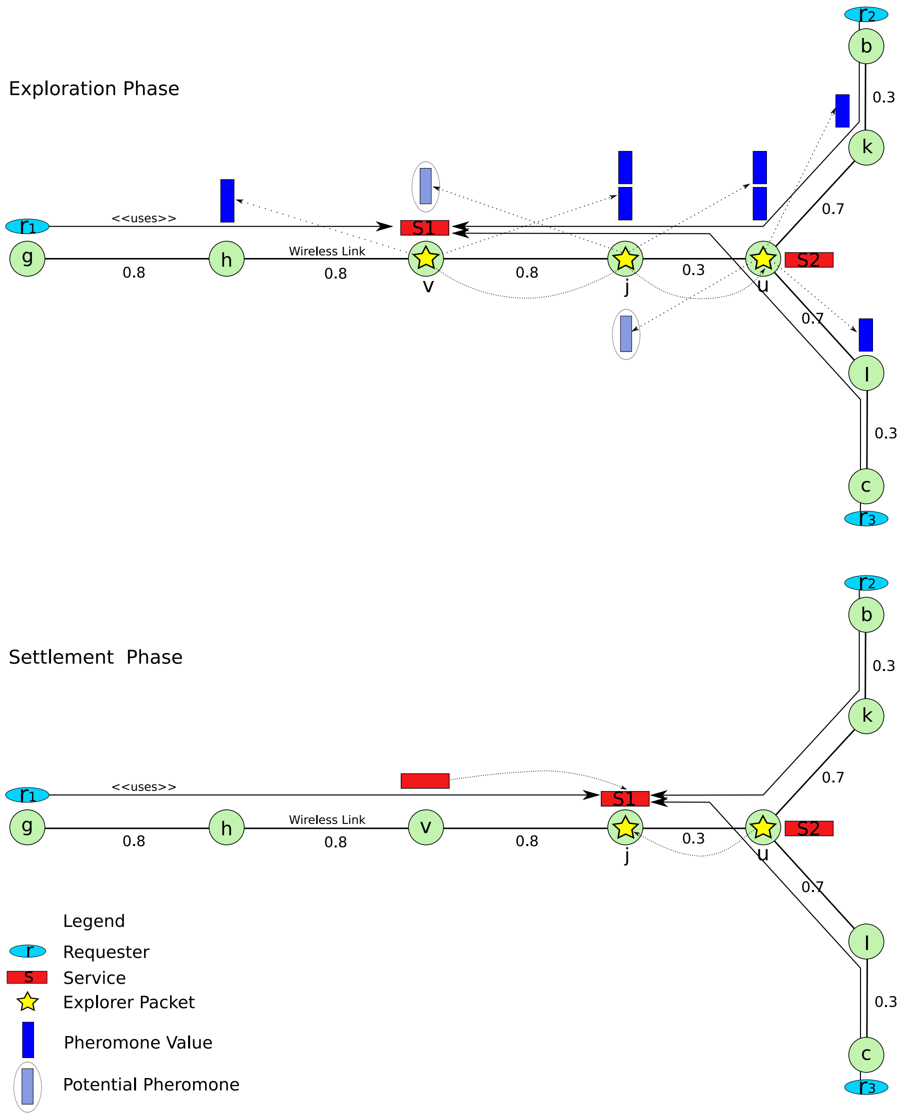


Figure 4.7: Example of the algorithm for a scenario with 9 nodes.

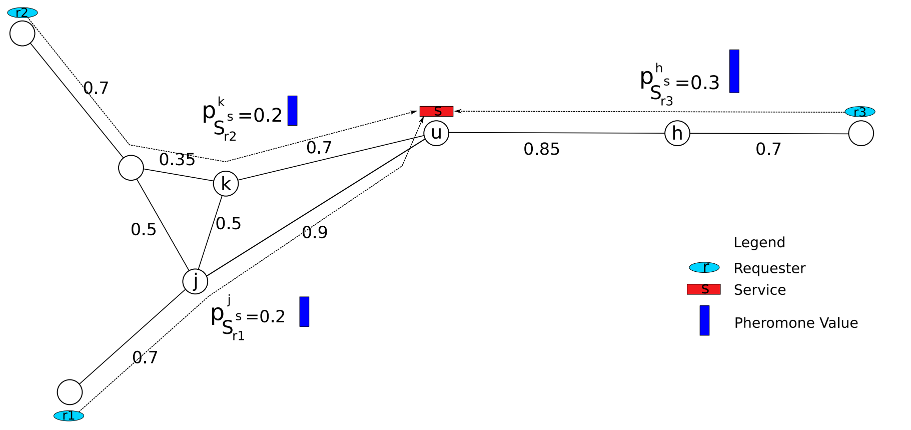


Figure 4.8: Instance of the problem that will result in a wrong migration decision due greedy behavior

this is not the case, because we suppose that each node may host at most one service and node u has already s_2. Therefore, a new candidate must be searched.

The neighborhood of u is analyzed using the virtual distance to define the priority order. Better links have priority. The node j has enough resource for the service s_1. Because it is also the last node visited, it is promptly selected for the migration. A message is sent to node s_1 to trigger the migration of s to node j.

4.4.2 Extended Heuristic

In this section, an identified problem caused by the greedy nature of the presented algorithm is described and a improved heuristic that tries to overcome some adversarial situations is proposed. It is also inspired by the nature.

For the purpose of simplification, we will assume for the following example that allowed_h=1, i.e., just one hop migrations are allowed. Nevertheless, the presented shortcoming of the heuristic is present for arbitrary values of this parameter.

The presented basic heuristic has a behavior that leads to suboptimal solutions when the following situation happens. More than one nearly located requesters use the same service, but due the used routing algorithm, the requests are routed through different paths. An example of such situation is depicted in Fig. 4.8. This situation can only occur if there are more than two requesters using the same service. It is more likely to occur when the service is located in a node-dense area of the network.

In the Fig. 4.8, the requesters r_1, r_2 and r_3 are acessing the service s in the node u. The total communication cost C can be calculated using the eq. 4.1.

In order to calculate the communication cost, we assume that the average bandwidth utilization is proportional to the pheromone deposited in a node inside the flow path. Thus, the total communication cost is 1.135, calculated using the equation 4.1.

Now, we analyze the migration that would be decided by the basic heuristic. As the pheromone value of the node h is higher than the value deposited in nodes j and k (separately), the exploration packet is send to node h. Let's suppose that allowed_h=1, the service would migrate to node h. The total communication cost of the system is in this case 1.22.

This result shows that the heuristic, in such adverse situation, selects the wrong node to migrate to, increasing the total communication cost of the system. This happens because of the lack of information over not directly-connected parts of the network (each node has just the *local* view of the system, i.e., just neighbors information is available). The main idea of the improvement is to migrate the service not to the neighbor with the biggest amount of requests (highest flow), like presented in the previous section, but to the neighbor whose flow, in some part, is crossing nodes near to other flows requesting the same service. If the defined metric (virtual distance) has (geographical) norm properties, this will be equivalent to migrate the service to the geographical *direction* from where the highest amount of requests is coming. Two flows related to the requesters r_1 and r_2 (see Fig. 4.8) are transversing neighboring nodes in their path to s, thus, they should attract the service instead of r_3.

We will define that such flows transversing neighboring nodes are called correlated flows. The definition follows later on.

In addition, the new migration heuristic isn't just based on the pheromone level to drive the settlement of the services, but also on a "potential goodness" of each node to receive highly loaded services and the energy level of the nodes. The "potential goodness" η_{vs} measures how appropriate is the node v to receive service s, i.e., whether the node is central in the network and the service s is a highly required one. If the complete network topology would be known by each node, the centrality could be measured by the sum of the distance to every other node. The idea of the potential goodness is that services with high request rates are coupled with high probability to locations with good connections to others. Just using this rule, it is possible to obtain good (but not optimal) placement of the services in the network [19].

In the same way as the basic heuristic, the location policy of the extended is divided in two phases: exploration phase, where a exploration packet is used for examining possible candidates for hosting the service, and the settlement phase, where a good candidate among the explored nodes and their direct vicinity is selected to host the service s.

In the next section, we will describe formally what are correlated flows. Subsequently, the structure of the exploration packet will be presented. Following, the two phases of our extended heuristic are described.

4.4.2.1 Correlated Flows

The concept of correlated flows is important in the extended heuristic. It will be defined here.

Let $ngh(v),\ v \in V$ be the set of all neighbors of node v, and $nghn(v),\ v \in V$ be the set of all neighbors of node v, including node v.

Definition 4.4.1. *The flows $S^s_{r_1}$,$S^s_{r_2}$,..,$S^s_{r_n}$, coming from different requesters and requesting the same service s (hosted in the node u), are* ***correlated flows*** *iff it exist an multiset $O = \{v_1, v_2, .., v_n\}$, where the n^{th} element of the multiset is one node selected from the path realized by the flow $S^s_{r_n}$ (i.e. v_n is selected from the path of the flow $S^s_{r_n}$), where the following holds: $\forall g, h \in O, \exists j_1, j_2, .., j_m \in O | j_1 \in nghn(g), j_2 \in nghn(j_1), ..., j_m \in nghn(j_{m-1}), h \in nghn(j_m)$ and $u \notin ngh(g) \cup ngh(h)$.*

Where $ngh(v)$ denotes the neighbors of the node v and $nghn(v)$ denotes $ngh(v) \cup v$. In the following items, the characteristics of the correlated flows are once again explained:

- The correlated flows are originating from different requesters and are requesting the same service
- They use different pathways to achieve the service
- Two flows are (*direct by*) correlated if they have at least one node in the pathway that is direct neighbor of a node in the pathway of the other flow

- The correlations are transitive, i.e., if $S^s_{r_1}$ is correlated with $S^s_{r_2}$ and $S^s_{r_2}$ is correlated with $S^s_{r_3}$, $S^s_{r_1}$ is also correlated with $S^s_{r_3}$

4.4.2.2 Exploration Packet

The following new fields are necessary in the exploration packet (in addition to those described in the section 4.4.1.1):

followed requesters: A list of requesters whose flows to s are being followed by the exploration packet (in the first round). We will call this set $FR \subset M$.

correlated requesters: A list of all requesters that have correlated flows to some member of the FR set. We will call this set $CR \subset M$.

correlated potential pheromone: List of pheromone values induced by the correlated requesters. The variable $Ph_r, r \in CR$ returns the correlated potential pheromone of a correlated flow.

non-correlated potential pheromone: This is just a new name for the potential pheromone field of the basic heuristic. It include all pheromone (in the first round) that don't contribute to the selection of the next hop of the exploration packet. In the equation, we call this field as nc_pot_pher. Note that differently from the previous field, this is not a list of pheromone but a field that store the sum of all non-correlated potential pheromone.

In Figure 4.9, the followed requesters, correlated requesters (and potential pheromone) can be seen. In 4.9(a), the initial state of the system is shown. There are tree requesters (r_1, r_2, r_3) using the service s. The service decides to start the migration process and launches the exploration packet. The first selected node is j (the next section will clarify how this decision is made). The packet is sent to node j. This situation can be seen in the Figure 4.9(b). From this example, we will describe the meaning of the new fields. The followed requesters are the requesters that cause the decision of migration of the exploration packet to the node j. They are responsible for the flow that the exploration packet is following. In the example, the requester r_1 is followed by the exploration packet. But for that decision (going to node j), the pheromone of the flow coming from r_1 was summed with the flow $S^s_{r_2}$, because it is a correlated flow (due the link between nodes a and c). This means that the identifier of requester r_2 should be stored in the correlated requesters field. As it can be seen in (b), we assume that the correlated requesters have a shortcut to the service s if the service will migrate to the actual position of the exploration packet. Potentially, they will generate pheromone that increase the tendency of migration in the direction of the followed requesters, and not back as the non-correlated potential pheromone do.

The fields described here are initialized with zero and filled with the real values just before the exploration packet leave its first node (the node hosting the service s).

4.4.2.3 Pheromone and Correlated Flows Table

In addition to the already presented pheromone table P_v that stores the rate of requests that are crossing the node v, there is a second table F_v that stores the information about flows occurring in the neighborhood of v (correlated flows). $F_v(S^s_r) : M \times S \rightarrow \{0, 1\}$ return 1, iff some direct neighbor of the node v is routing a request from the requester r to the service s.

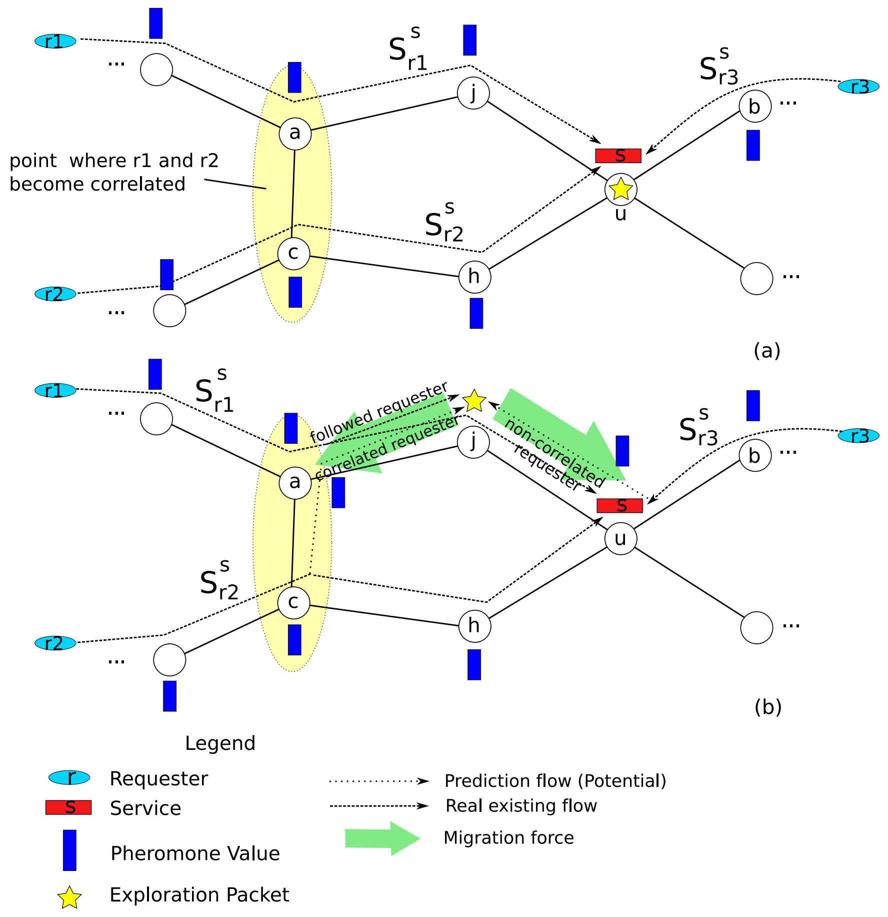


Figure 4.9: Example depicting the concepts of followed requesters, correlated requesters and potential pheromone.

The idea is that neighboring communications (like the $S^s_{r_1}$ and $S^s_{r_2}$ in the figure 4.8 and also in the figure 4.9) can be recognized as being originated in the same network "direction" and traveling to the service s.

The table F_v is filled without the necessity of any direct exchange of messages between the node v and the neighbors (just using passive listening, snooping). Each node just hears passively the communication originated in neighboring nodes to fill the table. In the Figure 4.9(a), the nodes a and c are neighbors, therefore the communication flows $S^s_{r_1}$ and $S^s_{r_2}$ are considered correlated, i.e., $F_a(S^s_{r_2}) = 1$ and $F_c(S^s_{r_1}) = 1$.

There is a constraint when filling the table: if the node v has a directed connection to the node u where the service s is located, it ignores all the neighboring communication going to to the service s (i.e., for $\forall r \in M$, $F_v(S^s_r) = 0$). This avoids the problem that near the sink (service s), all nodes can hear each other, resulting on a false interpretation that all requests are coming from a similar network "direction".

Each request r to the service s now carries the information collected in the nodes about which requests to the service s are occurring in neighboring nodes (i.e., while being relayed to the service s, the F_v information is added to a special field of the request packet). $F(S^s_{r_1}, S^s_{r_2}) : M \times M \times S \rightarrow \{0,1\}$ return 1 iff r_1 and r_2 are neighboring requests (correlated flows) regarding service s.

4.4.2.4 Exploration Phase

The exploration phase from the extended heuristic is similar to the basic one, nevertheless, here neighboring flows are recognized as correlated and used in order to drive the exploration packet. The first thing that the node u must do upon receiving the exploration packet is again to verify whether allowed_h is 0. In this case, the settlement phase starts. If in some neighboring node, a flow coming from a requester present in the correlated requesters is found and the node receiving the exploration packet is not a direct neighbor of the node containing the service s, the exploration phase also ends and the settlement starts. The fact of encountering a correlated flow in some direct neighbor highlight that this position (node u) is the point causing the correlation. This is showed, for example, in the figure 4.9.

Otherwise, the next destination of the packet is defined. In the original heuristic, the "force" attracting the exploration packet from node u to node d ($b^s_{u,d}$, see eq. 4.3) does not take into account the requests coming from near areas of the network. In this version of the heuristic, $b^s_{u,d}$ is calculated taking in account the pheromone values and correlated flows information (eq. 4.7 and 4.8). Here also the non-correlated potential pheromone and the correlated potential pheromone are used.

If we are calculating the force of a node that has not been already visited (i.e. $d \neq l$, where $l = last(history)$):

$$b^s_{u,d} = \frac{\overbrace{\sum_{x \in M} p^d_{S^s_x}}^{\text{flows using d}} + \overbrace{\sum_{x \in M} \sum_{z \in M} \sum_{g \in Ngh_v - \{d,l\}} p^g_{S^s_z} \cdot \lceil p^d_{S^s_x} \rceil \cdot F(S^s_z, S^s_x)}^{\text{correlated flows}} + \overbrace{\sum_{x \in FR} \sum_{y \in CR} \lceil p^d_{S^s_x} \rceil \cdot F(S^s_x, S^s_y) \cdot Ph_y}^{\text{potentially correlated flows}}}{normalizer} \tag{4.7}$$

And the attraction force for the last visited node is given by:

$$b^s_{u,d} = \frac{nc\_pot\_pher}{normalizer} \tag{4.8}$$

where:

$$normalizer = \sum_{y\in(Ngh_v-\{l\})} [\sum_{x\in M} p^{y}_{S^s_x} + \sum_{x\in M}\sum_{z\in M}\sum_{g\in Ngh_v-\{d,l\}} p^{g}_{S^s_z}\cdot\lceil p^{y}_{S^s_x}\rceil\cdot F(S^s_z,S^s_x)] + \\ + \sum_{x\in ISF}\sum_{y\in ICF}\lceil p^{d}_{S^s_x}\rceil\cdot F(S^s_x,S^s_y)\cdot Ph_y + nc\_pot\_pher$$

The first term of the eq 4.7 is the same of the eq. 4.3, that means, the sum of all requests coming to service s through node d. The second term of the numerator is the sum of the pheromone generated by correlated flows of the flows present in the node d. As already explained, the function F tests whether S^s_z and S^s_x are correlated flows, and the ceiling $\lceil p^{d}_{S^s_x}\rceil$ checks whether the connection S^s_x exists in the node d (i.e. $p^{d}_{S^s_x} > 0$). The denominator normalizes $b^s_{u,d}$ $(0 \leq b^s_{u,d} \leq 1)$. The third term accounts the correlated potential pheromone already explained and shown in Figure 4.9. As already said, we assume that the correlated potential pheromone will act in the same direction of the followed flow, i.e., should be summed to it.

In the eq. 4.8, we calculate the force in the direction back to the node currently hosting s, i.e., the force attracting the exploration packet back to the last visited node ($last(history) = l$). We use here the non-correlated potential pheromone because we assume that the flows that are not correlated to followed flows will act with a contrary force (see Figure 4.9).

Again the main idea here is to forecast the situation that a service will encounter if it would migrate to the position of the exploration packet.

The next hop of the exploration packet is again selected using the equation 4.4. Let's call the selected node j. If the selected node is the last node visited (i.e., $j = l$), the heuristic starts the settlement phase (see next section). Otherwise, the allowed_h field is decremented, and the current node is inserted in the history field. Moreover, if the exploration packet is in the same host as the service (is the first exploration phase being executed), the fields of the exploration packet presented in the section 4.4.2.2 are updated:

- The field followed requesters receives all the requesters whose pheromone generated by the corresponding flow contribute to select j as winning node, i.e., all requesters using s that have positive pheromone in the node j. $FR = \{m \in M | p^{j}_{S^s_m} > 0\}$. In the example shown in the Figure 4.9, the requester r_1 would be added to this field.

- All correlated requesters of the flows added in the previous field are included in the correlated requesters. This means that $CR = \{x \in M | \exists m \in FR, F(S^s_m, S^s_x) = 1$ and $\exists h \in (Ngh_u - \{j\}), p^{h}_{S^s_x} > 0\}$. In the example, the r_2 is included in this field.

- For each requester in the correlated requesters ($r \in CR$), the correlated potential pheromone is updated by:

$$Ph_r = p^{b}_{S^s_r} \tag{4.9}$$

- Finally, the non-correlated potential pheromone is also update, here instead of individual pheromone value, just a summarized value (sum) is stored.

$$nc\_pot\_pher = \overbrace{\sum_{\{h\in Ngh_u|h\neq j\}}\sum_{\{x\in M\}} p^{h}_{S^s_x}}^{\text{flows that don't use } j} - \overbrace{\sum_{\{k\in Ngh_u|k\neq j\}}\sum_{\{x\in FR\}}\sum_{\{y\in M\}} p^{k}_{S^s_y}\cdot F(S^s_x,S^s_y)}^{\text{correlated flows}} \tag{4.10}$$

Finally, the exploration packet is forwarded to the node j and the exploration phase starts again.

4.4.2.5 Settlement Phase

Again after the exploration phase, where possible hosts candidates for the service s are appraised, the settlement phase is responsible for selection of the de facto new host of service s. For this section, let's call u the actual node of the exploration packet.

In the basic algorithm, the main concern of the settlement phase was about the amount of free resources of each candidate. In this extension, besides the amount of free resources, a "potential goodness" of each node to receive highly loaded services and the energy level of each node are also addressed in order to select an appropriate node to the service. The "potential goodness" η_{vs} measures how appropriate is the node v to receive service s, i.e., whether the node is central in the network and the service s is a highly required one. If the complete network would be known by each node, the centrality could be measured by the sum of the distance to every other node. The idea of the potential goodness is that services with high amount of communication are coupled, with higher probability, to locations with good connections to others. Just using this rule, it is possible to obtain good (but not optimal) placement of the services in the network [19].

Like in the basic heuristic, each node has just local information. This means that the "potential goodness" cannot rely on global knowledge. Although without global knowledge about the network topology we cannot determine exactly whether a node is central in the network, we use local information to approximate it. We define η_{vs}, where $v \in V$, $s \in S$ and $0 \leq \eta_{vs} \leq 1$:

$$\eta_{vs} = [1 - \sum_{g \in Ngh_v} \frac{D(v,g)}{(|Ngh_v|)^{\delta}}] \cdot h(s) \tag{4.11}$$

Where $Ngh_v, v \in V$ the set of neighbors of v and $\delta \geq 1$ gives the importance of the number of neighbors. $h(s) : S \rightarrow [0,1]$ returns the current request load (how much traffic) that service s is currently serving. This can be measured by means of how much pheromone concerning the service s the neighbors of the hosting node have. A $h(h) = 0$ means that there is no pheromone in the neighboring nodes and 1 means a maximum pheromone sum was reached).

The energy of the node is modeled by E_v and $0 \leq E_v \leq 1$, where 1 means full battery and 0 depleted.

Now, we calculate the so called settlement fitness of a node v to receive the service s ($\sum_{i=1}^{3} k_i = 1$):

$$sf_{vs}^{u} = k_1 \cdot \eta_{vs} + k_2 \cdot E_v + k_3 \cdot [1 - w(u,v)] \tag{4.12}$$

Where u is the node where the exploration packet is currently located. The third term penalizes nodes that are away from the existing flow, as we will describe next.

In order to gather the settlement fitness when exploring the network, a new field is necessary in the exploration packet: best settlement fitness (or bsf). It stores the best settlement fitness found up to the moment. For each node that the exploration packet visits (in the exploration phase), it calculates the settlement fitness of the node and also calculate this fitness of good connected vicinity nodes (with the link metric are less than a given threshold γ). The third term in the equation 4.12 penalizes the links according their virtual distance, i.e., nearer vicinity nodes are privileged. The node actually visited by the exploration packet is benefited largely (because $w(u,u) = 0$). Just nodes with enough resources are considered ($e(s) \leq r(u) - \sum_{\{m \in M | q(m)=u\}} e(m)$). The calculated values are compared with the existing best settlement fitness and if it is necessary, the field is updated.

After these definitions, we will describe the procedure of the settlement phase.

The following procedure is executed:

- The settlement fitness of the node u and the good connected neighborhood ($\{k \in Ngh_u | w(u,k) \leq \gamma\}$) is calculated. Just nodes with enough resource are selected ($e(s) \leq r(u) - \sum_{\{m \in M | q(m)=u\}} e(m)$). Let's call the highest settlement fitness as sf_{win}.

- The best settlement fitness is compared to sf_{win}. If $sf_{win} > \rho bsf$, where $\rho \leq 1$ is the accepted difference, the winning node is automatically selected. In this case, the settlement phase is terminated and the node u sends a message to the service s in order to trigger the migration process.

- Otherwise, the exploration package is sent back to the $last(history)$ node. The node is then deleted from the history field and the settlement procedure, here described, is started again. This is repeated until an appropriate node is found. In the case of not finding any new host, the migration may be canceled otherwise the swap operation may be used as in the basic heuristic.

4.4.2.6 Example

In the Figure 4.10, an example of the service migration is showed. In (a), the initial situation, with tree requesters (r_1, r_2 and r_3) and a provider (service s) is depicted. The exploration packet has been launched and resides in the node u. The heuristic is in the exploration phase. Using the equations 4.7 and 4.4, the next hop is selected. The node j has been chosen, because $S^s_{r_1}$ and $S^s_{r_2}$ are correlated flows, therefore the pheromone of the both flows are used in order to take the decision. We have here $b^s_{u,j} = 0.4$ and $b^s_{u,b} = 0.3$. The fields followed requesters, correlated requesters, correlated potential pheromone, non-correlated potential pheromone are updated based on the rules described in section 4.4.2.4. For this example, followed requesters receives r_1, correlated requesters receives r_2, correlated potential pheromone receives 0.2 and non-correlated potential pheromone 0.3. Finally, the exploration packet is sent to node j.

In (b), the next situation is described. The exploration packet is in the node j, the settlement fitness is calculated and the best settlement fitness field is updated. The next hop of the exploration packet is calculated using equations 4.7, 4.8 and 4.4. It is important to remark that now two pheromone values are accounted in the node a, the real pheromone generated by the flow $S^s_{r_1}$ and a potential correlated pheromone. This correlated pheromone is used to forecast the situation in the node (in this case j) if the service would be placed there. It is read from the field correlated potential pheromone. In the node u, the pheromone that acts is the non-correlated potential pheromone, read from this field in the exploration packet. Here we have again $b^s_{j,a} = 0.4$ and $b^s_{j,u} = 0.3$. The exploration packet is sent therefore to node a.

This next situation is depicted in Figure 4.10(c). The exploration packet is now in the node a. The settlement fitness of the nodes are calculated and compared with the best settlement fitness. Node a has the higher fitness and the value is updated in the exploration packet. As node a has a correlated flow in a neighboring node (node c), the exploration phase is ended and the heuristic goes to the settlement phase.

In the settlement phase, the settlement fitness from node u and from the good connected neighborhood is calculated. Just nodes with enough resource are selected (in this example, let's assume that a and c have enough resources). The higher settlement fitness is compared to the best settlement fitness. In the example, the both are the same, therefore, node a is selected to host the service s. The service s is informed about its new destination and the migration can start.

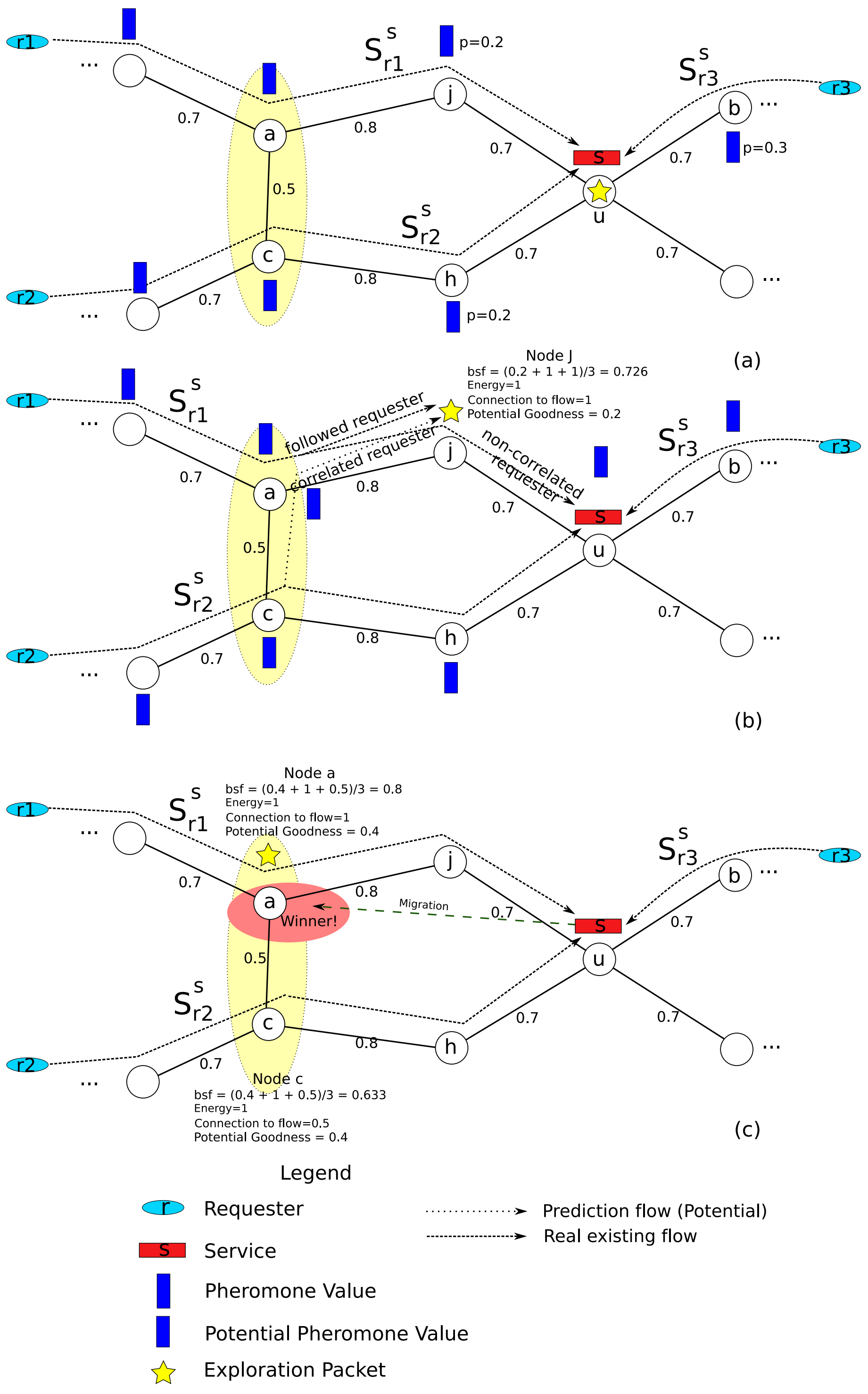


Figure 4.10: Example of service migration using the extended heuristic.

4.5 Discussion

In this chapter, we present two heuristics that aim to reduce the communication overhead among the tasks and mobile services in an ad hoc wireless network or sensor network. Many existing load sharing/balancing protocols have different objectives, between them we can stress the minimization of the make span of a task set. Just in some proposals the communication delays are taken in account. Differently from these algorithms, our main focus here is to reduce the communication cost. This cost is calculated using the amount of communication being realized between two entities and the distance of this communication (for this, the link metric is used). Moreover, our heuristic (the basic and extended versions) is developed to provide dynamic self-optimization during the run time of the system. If the topology of the network changes, there is an automatic response from the service distribution heuristic.

As already presented in the section 4.2.3, other WSN middleware/virtual machines have also pursued this objective. Nevertheless, as presented, our approach is more flexible (not targeting just one special set of applications) and produce better placements.

In order to produce a self-organizing system, the algorithm is completely distributed and each service can be seen as an agent with local information and local rules to decide about the migration. We start from the idea coming from reactive agents: intelligent behavior (in our case, the search for a global reduction of the communication overhead) emerges from the interaction of simple behavior distributed over the agents [134]. Our services (or agents) recognize the current environment state (pheromone levels in the neighborhood) and start the discovery process in a network direction, guided by this local view. The communication is done by means of stigmergy: the exchanged message between the requesters and the services leave information in the communication path (the pheromone values). Stigmergy means that the communication is done using the environment instead direct exchange. Because the heuristic deals with movement of components along paths in the network, the approach of marking paths and communication through this mark instead of exchanging explicity messages has an inherent advantage. The pheromone stores indirectly statistical information about the communication patterns without the need of explicit control messages.

The agents, in our system, have an egoistic behavior: each one tries to optimize its local utility function by means of how much its communications are costing, without taking in account the goal of the other services in the system. We aim, with this approach, to find a good placement of the services avoiding the complexity and overhead of an approach with a more global view and global control about the system's objective. Moreover, due to this independence between agents, the system is robust against failures and topology changes. Nevertheless, as usual for such sort of algorithm, we are finding suboptimal solutions.

Another important point to be highlighted here is that although some similar metaphors are used (like pheromone value, ants and stigmergy), the basic and extended version of the heuristic developed here are not modifications of the well known ant colony optimization algorithms [43, 19]. These are based on a distributed autocatalytic process and may be used to solve classical optimization problems. Like genetic algorithm and other meta-heuristics, they have been originally developed to run in a centralized system (but may be distributed). Our algorithms, in other hand, are running in a distributed-fashion among the nodes of the sensor network and are developed just for solving the presented problem. Moreover, they do not use the autocatalytic effect in the same way as the described in the ant colony optimization. Autocatalysis plays a central role in our dynamic clustering approach, which is presented in the next chapter.

In the basic and extended version of the presented heuristic, we are dealing with migrations of complete services. Those migrations have advantage when we want to control the number of copies of a service inside a given cluster or long distances exist between the service and the providers. We combine this long distance migration with an approach that tries to better distribute the load of the services by migrating just one context to another service. When the service is migrating in the direction of a group of providers, the context belonging to other providers being penalized by the migration and placed far from the service are marked as available for this local context-only migration. This algorithm will not be presented here and can be found in [151].

The disadvantage of such context migration is that the common state area of the service must be the same for both source and target services. Moreover, if new services are always created, a large overhead of controlling several incomplete instances of the same service is generated.

Another important point to remark is that the heuristic presented here is dependent of the underlying routing algorithm. A single-path routing algorithm is desirable and, for a better performance, the virtual distance should be used as metric instead of some traditional approaches that use the number of hops.

A common situation that may occur in dynamic scheduling protocols and may be also a concern to our heuristic is an instable behavior. For example, when one module communicate in burst mode, carrying a lot of communication during certain periods and almost none in other periods, unnecessary migrations may happen. To avoid that, the intensity of pheromone values layed by the communication packets in the network and the evaporation of the pheromone must be correctly adjusted, in order to trigger migrations just when a intensive communication holds a significant period of time.

Using the heuristic presented in this chapter, NanoOS can provide a transparent service placement relieving the application's designer. He can concentrate in the program's logic and create complex applications without concerning about the physical position of the services. This system level placement results in a smaller energy utilization of the sensor network and adapts the system as well as applications to the actual network topology and communication needs.

Chapter 5

Self-Organizing Cluster Construction

5.1 Introduction

In general, there are two heuristic design approaches for management of ad hoc networks at different levels (e.g. topology control, network layer, application). The first method is to have in all nodes the knowledge of the (entire) network and let they manage themselves. This circumvents the need for more advanced organization. Nevertheless, this generates the overhead in terms of communication and memory at each node. Each node must, for example, maintain routes to the other nodes in the network. In large networks, the number of messages needed to maintain routing tables may cause congestion in the network and depletes the energy of the nodes. Ultimately, the need of individual self-management will generate a huge exchange of messages and overhead.

The second approach is to identify a subset of nodes within the network and vest them with the extra responsibility of being a leader (clusterhead) of certain nodes in their proximity. The clusterheads are normally responsible for managing communications between nodes in their own neighborhood as well as routing information to other clusterheads in other neighborhoods [4]. This creates a hierarchy in the network. Clustering in large-scale networks was proposed as a means of achieving scalability through a hierarchical approach [123]. Some examples of clustering benefits can be found at the medium access layer, where clustering helps to increase system capacity due to the promotion of the spatial reuse of the wireless channel, and at the network layer, where it helps to reduce the size of routing tables. Wireless ad hoc networks benefit a great deal from clustering.

In this chapter, we present the state of the art of clustering in ad hoc networks, and after this, two new heuristics to organize an ad hoc network into clusters. Differently from the previous approaches, our proposal addresses the problem of partitioning the nodes of the network in multi-hop groups with a guaranteed minimum amount of resources q (or budget) in each one of them. This kind of clustering is useful in various scenarios. In our case, the clustering heuristic is used in the development of an efficient service distribution in our OS.

The idea is to group a complete instance of the OS and application services inside a single cluster. This brings a reduction of the organization overhead, since the discovery process will be locally constrained (within one cluster) and the pheromone tables used in the service distribution must only store pheromone values for services used by the application inside the cluster. Furthermore, a simple but efficient service discovery based on a central broker per cluster can be easily implemented. Moreover, a topology control of the network can be easily realized based on the hierarchy created by the clustering. The clustering brings additional advantage: the applications can implement algorithms based on this created hierarchy that help them scale.

Constraining an instance of the OS inside a cluster facilitates the maintenance of the consistency of the OS, because the dependencies among the different modules are constrained within the cluster's nodes. Moreover, in the worst case, for any distributed algorithm, a node may keep state information about all other nodes in the cluster and not the complete network. This is true even if a central control paradigm is used.

5.2 State of the Art - Clustering in Ad hoc Networks

In this section, a literature overview of clustering algorithms developed for ad hoc networks is presented.

The idea of clustering is to partition the nodes of a graph in subsets in a way that the union of the subsets contains all nodes of the graph. For each subset (or cluster), some conditions should hold.

Given a graph $G = (V,E)$ representing a communication network, where vertexes are the nodes and edges the communication links. The clustering process construct subsets of nodes $V_i, i = 1,..,n$ where $\cup_{i=1,..,n} V_i = V$, such that each subset V_i induces a connected sub-graph of G. These vertex subsets are clusters. Ideally, the size of the clusters falls in a desired range. Moreover, for several approaches, a special vertex in each cluster is elected to represent the cluster and it is called clusterhead [32].

According to [69], the following design factors differentiate the various approaches:

Clusterheads: The partition of the graph V into clusters does not require considerations about the internal structure of clusters. However, it is common to define a node that will assume the leader role in the cluster.

Neighboring Clusterheads: It can be defined that clusterheads may or may not be direct neighbors. When not, the clusterhead set forms an *independent set*: a subset $C \subset V$ such that $\forall c_1, c_2 \in C : (c_1, c_2) \notin E$. Normally, the approaches try to calculate the *maximum independent set*, which contains the maximum number of nodes. The *maximum independent set* is also a *dominating set*[1]. Determining the *maximum independent set* is an NP complete problem.

Overlap of clusters: Clusters may overlap when it is allowed that member nodes participate in more than one cluster.

Maximal Diameter: Although normally clusters have diameter of two (when constructed by the *independent set*), it is also possible to have multi-hop clusters with larger diameters.

Hierarchy of Clusters: Either a two-level or multi-level hierarchy is used. In the multi-level hierarchy, each cluster is considered a node in a recursive cluster construction method.

Rotating clusterheads: It is possible to have clusters with fixed clusterheads during the life time of the system or rotating clusterheads that reassign the clusterhead role to another node periodically.

Another aspect that should be considered is the communication among the clusters. *Gateways*, nodes that are adjacent to two clusterheads, can be used. In case that clusterheads are separated by more then one node, the so called *distributed gateways* can be used.

[1]A *dominating set* is a subset of nodes $D \in V$ where each node in V either is a member of D or is a direct neighbor of a node in D.

We divide the different approaches of obtaining a clustered network in three main groups: the *Maximum independent set* approaches, where the objective is to find clusters where all members are at most one hop away from the clusterhead and there are no neighboring clusterheads, the dominance only approaches, where neighboring clusterheads are allowed, and the multihop clustering approaches, where diverse multihop objectives are pursued.

5.2.1 Maximum Independent Set Approaches

There are several clustering algorithms that aim to find the *Maximum independent set* (MIS) of a network modeled as an undirected graph. This is often combined with the dominance property, which leads to the following clustering properties that should be satisfied:

Independence No two clusterheads can be neighbors. This property assures that the set of clusterheads will be scattered, i.e., they will not be grouped in a small part of the network.

Dominance Every ordinary node has at least one clusterhead as direct neighbor.

Several heuristics have been proposed to find clusters based on the maximum independent set. The existing clustering algorithms differ on the criterium for the selection of the clusterhead [13]. For example, the highest-ID and lowest-ID heuristics use the unique identifier (*id*) to select the clusterhead. The choice can be also the degree of the nodes (number of neighbors), as in the node-degree heuristic. Others combine different parameters in a metric (e.g. VDBP heuristic).

Many of the heuristics are working in a similar manner. Each node with the highest (lowest) metric in the nearby vicinity is selected locally as clusterhead (being included in the independent set). This information is broadcasted and the neighborhood joins the forming cluster. The new members also inform their neighbors about the member status (broadcating this status). This un-locks other nodes that have lower a metric to eventually become clusterhead (i.e., the new elections are restricted to nodes that are neither members nor clusterheads).

In the next section, we will present several algorithms that are based on independent sets.

5.2.1.1 Identifier-based Clustering

In this section, we will present the heuristics that use a fixed identifier to elect the set of clusterheads.

Highest-ID ranking In [6, 47], the authors consider the problem of organizing a set of mobile, radio-equipped nodes into a connected network. They argue that a reliable structure should be acquired and maintained in the face of arbitrary topological changes due to motion or failure, and this structure should be achieved without a central controller. For that, a self-starting distributed algorithm that maintains a connected architecture is presented. The algorithm is based on a cluster construction based on node *id*.

The two logical stages of the algorithm are the formation of the clusters and the linking of them. Here we will concentrate on the formation stage.

Each node has a `connectivity` matrix with binary entries. $(k, j) = 1$ means that packets sent by k are received by j.

The algorithm is based on a *TDMA* scheme where the control messages have a fixed schedule. In the TDMA frames, node i transmits information in the slot i. The two steps (based on two communication rounds or frames) of the algorithm are:

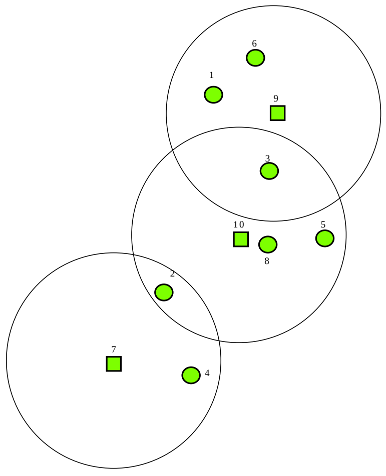


Figure 5.1: Example showing cluster formation using the method described in [6].

1. Each node broadcasts its own identity and the identity of the already heard nodes.

2. Each node broadcasts the complete information about which nodes can be reached by its broadcasts (the *i* row of the `connectivity` matrix) plus the status of the node (undecided, clusterhead, member).

For deciding the status, each node checks the connectivity row. If there is no neighbor with higher *id* number, the node becomes clusterhead. The algorithm is a distributed version of the very simple centralized procedure: start with the highest numbered node, say, *N* and declare it clusterhead. Draw a circle around that node with radius equal to the range of communication. If some node is outside the circle, then select the higher *id* among the nodes outside the circle and restart the process until all nodes are inside at least some circle [47]. Figure 5.1 shows an example of a small network clustered using this process.

The described procedure has four keys elements: knowledge of the neighbors of each node, a rule to select the clusterhead from a set of candidates, knowledge of the sequence in which clusters are to be formed, and knowledge of each node's own clusterhead.

In [7], the same authors describe the same method with minor differences.

Lowest-ID heuristic Another approach called *Lowest-ID Cluster* is presented in [53]. The algorithm works in a similar way. Periodically, each node broadcasts the list of nodes that it can hear (including itself). A node which only hears nodes with ID higher than itself is a “clusterhead”. The lowest-ID node heard by a node is its clusterhead, unless it gave up its role. The other nodes are members. The difference to the previous algorithm is that the election is not realized in a fixed, determined TDMA order where the nodes *id* must be related to the transmission TDMA slot in a increasing order. This means, it is easier to implement the *Lowest-ID Cluster* in different architectures due to its independence on a specific MAC protocol.

It is important to highlight that both algorithms here have a similar structure: the *id* are exchanged within the neighborhood. Each node decides whether it should become clusterhead based on those received *ids*. When a node become clusterhead, it announces that. The decision of becoming clusterhead is taken based on the lowest *id* among the nonmember nodes.

At receiving a clusterhead announcement, the neighboring nodes become members of the cluster and announce that also. This unlock lower *id* nodes (because the clusterhead decision is done just among nonmembers), that can now become clusterhead.

The cluster formation resulting from the application of the *Lowest-ID Cluster* algorithm has the independence and domination properties. Moreover, the algorithm is also suitable for networks where the nodes move, causing re-election of clusterheads and also changes in the members-clusterheads assignment.

5.2.1.2 Node’s Degree Ranking

In this section, we will present some heuristics that use the node degree in order to rank the nodes for the clusterhead selection.

Highest Degree Clustering In [53], an additional clustering approach is presented. Instead of selecting the clusterheads based on a pre-assigned rank of each node, the election relies on the degree of the node in the graph. The heuristic is called *Highest-Connectivity Cluster Algorithm* and has the following steps:

1. Each node broadcasts the list of nodes that it can hear (including itself)
2. A node is elected clusterhead if it is the most connected node of all its “not clustered” neighbor nodes
3. A node which has already elected another node as its clusterhead gives up its role as clusterhead and becomes a member of the cluster.

Like in the previous section, the formation resulted by the application of this algorithm has the independence and domination proprieties.

VDCA Heuristic The Variable Degree Based Clustering Algorithm [85] uses different rules based on the node’s degree to assign priorities to the nodes for the clusterhead selection.

The different rules proposed in the algorithm are:

Maximum degree: The algorithm is similar to the highest degree clustering.

Degree two priority: Nodes with degree two have the highest priority. Under this rule, clusters with three nodes will be build first.

Average Degree Priority: The average degree of the network has the highest priority.

The idea of not using the highest degree is to reduce the variance of the clusters size and to control the number of clusters. The authors argue that reduced variance results in a smaller lower bound of the total size of the routing table. Moreover, they aim to control the number of clusters in order to keep it near the optimal number for reducing the size of the routing table.

A difference from the other algorithms is that each node keeps the complete topology of its voting area, which can be one or two hops. Messages are exchanged in order to enable each node to construct its topology map. Further, the algorithm was developed to enable multiple hierarchy levels.

5.2.1.3 Combined Metric Clustering

In this section, the clustering algorithms that use a combined metric to rank the nodes in order to select the clusterhead are presented.

VDBP Heuristic The Virtual Dynamic Backbone Protocol [73] (VDBP) is a heuristic that constructs and maintain clusters that serve as backbone. In the first phase of the algorithm, a dominating and independent set is constructed. The difference from the other heuristics here presented is that the algorithm uses mobility information in the clusterhead election.

The idea is the selection of relative stable nodes to be part of the clusterhead set. Moreover, the number of clusterheads should be also kept small. The clusterhead election is made based on the following information: normalized link failure frequency, number of nonmembers in the neighborhood, node identifier (to break the ties). The nodes are rated based on this information.

The normalized link failure frequency is used to estimate the relative mobility pattern of a node. Higher link failures indicate a higher mobility. The number of nonmembers in the neighborhood is used to check how connected the node is. Nodes with plenty nonmembers in the vicinity have higher priority to become clusterhead.

Every node sends this information periodically in two types of messages: the Hello message and the Global Broadcast Messages (GBM). The first is frequently sent and has a reachability of 1 hop. The GBM has a much larger inter-message period and is forwarded by the clusterheads, which put their information inside before broadcasting.

When a node powers on, it starts sending the both messages. A timer for checking whether the node should become clusterhead is set. At the same time, it collects information from the other nodes. When the timer expires, the node checks whether it has the higher rate in the neighborhood. If positive, it declares itself clusterhead. When not, it starts the timer again.

Upon receiving a Hello or GBM message from a direct neighbor that is clusterhead, a nonmember node becomes member of the cluster.

MOBIC Heuristic The lowest mobility clustering algorithm (MOBIC) [14] uses a metric for ranking the nodes that is exclusively based on the mobility of the nodes.

As usual for this class of heuristics, beacons are sent between neighboring nodes to advertise their presence. But different from the other heuristic, the received power levels (RSSI) of the beacon packets are used to calculate the relative mobility of the node. Before sending the next beacon, each node computes an aggregate relative mobility metric. This value is sent with each beacon packet.

When a node has the lowest aggregate relative mobility among all its neighbors, it assumes the clusterhead status. Vicinity nodes are attached to the clusterhead with the lowest aggregate relative mobility.

An additional enhancement to the other approaches is the introduction of a delay before re-clustering the network when the topology changes. This is done to avoid incidental contacts between passing clusterheads to trigger a re-clustering process.

WCA Heuristic In this section, we will present the Weighted Clustering Algorithm (WCA). The clustering scheme tries to preserve its structure as much as possible when nodes are moving or the topology is slowly changing.

The combined metric tries to measure how appropriate a node is to be the clusterhead. It takes into account the following parameters:

Ideal number of members δ: It states how many members are desired in each cluster. The *degree-difference* of the node v, $\Delta_v = |d_v - \delta|$ returns the difference of the requested number of members to the current number of neighbors (given by $d_v = |Ngb(v)|$, $Ngb(v)$ is the set of neighbors of v).

Battery power: The clusterheads have an extra energy consumption. It is not desired that nodes almost depleted assume the clusterhead role. P_v is the cumulative time during which a node v acts as a clusterhead and has used extra battery power. This parameter acts for the rotation of clusterheads in the heuristic.

Mobility: The election of nodes that do not move very quickly is desirable. The mobility is measured by the average speed of the node up to the current time T, $M_v = \frac{1}{T}\sum_{t=1}^{T}\sqrt{((X_t - X_{t-1})^2 + (Y_t - Y_{t-1})^2}$

Distance to neighbors: A clusterhead can communicate better when its neighbors have a smaller distance from it. The distance to neighbors is calculated using: $D_v = \sum_{v' \in N(v)} dist(v, v')$.

The combined weight W_v for each node v is evaluated by:

$$W_v = w_1\Delta_v + w_2D_v + w_3M_v + w_4P_v$$

w_1, w_2, w_3, w_4 are the *weighing factors*.

The node with the smallest W_v is chosen as clusterhead in a very similar fashion to the other algorithms. In order to cope with mobility, the clusterhead election is invoked multiple times, however as rarely as possible. It is not invoked if the relative distance between the clusterhead and the nodes does not change significantly.

Nevertheless, due to the dynamic nature of the system, the nodes tend to move in different directions, disorganizing the stability of the network. The system has to update itself from time to time. In WCA, all nodes continuously monitor their signal strength received from the clusterhead. If the signal between a cluster member and the clusterhead gets weak, the member informs the clusterhead that it is no longer able to attach itself to that clusterhead. Then, the clusterhead tries to hand over the node to a neighboring cluster. If the node goes to a region not covered by any clusterhead, the clusterhead selection algorithm is again invoked.

Additional characteristics of WCA are: the presence of a methodology to balance the load (amount of members) among the clusterheads and the assumption the nodes have two power modes (radio) for short and long distance communication. The short range mode is used for communication between members and clusterheads, and the long range between clusterheads.

DCA and DMAC Heuristics The Distributed Clustering Algorithm (DCA) and the Distributed Mobility-Adaptive Clustering (DMAC) were proposed in [12] and presents a generalization of the greedy dominating independent set heuristics.

A common model presented in several previous algorithms is the greedy search for a *Maximum Weight Independent Set* (MWIS) in a graph, where non-negative weights are associated with the nodes. These weights are the degree of the node in the *Highest-Connectivity Cluster Algorithm* and the node's *id* in the "lowest *id* first" approach.

The objective of the *Maximum Weight Independent Set* problem is to find an independent set of nodes where the sum of weights is as big as possible. The MIS problem is a special case of the MWIS problem, and as MIS, it is an NP-hard problem.

The main idea of the generalization is that with the appropriate selection of weights, the *preference* to have a given node as clusterhead can be expressed. The authors propose the centralized *Generalized Clustering Algorithm* (GCA), that is a generalization of the previous clusterhead selection algorithms. The procedure is showed in the Algorithm 1.

Algorithm 1 The *Generalized Clustering Algorithm*

```
{input: $G = (V,E)$: network, $w$: weights; output: $\{C_i\}_{i \in I \subset V}$;}
$i \leftarrow 0$
while $V \neq \emptyset$ do
    $i \leftarrow i+1$
    {Pick the node with the lowest ID among those with maximum weight}
    $v \leftarrow min\{u \in V : w_u = max\{w_z : z \in V\}\}$
    {$Neigh(v)$ returns the set of neighbors of node $v$}
    $C_i \leftarrow \{v\} \cup Neigh(v)$
    $V \leftarrow V \setminus C_i$
end while
```

The author defined the *quality* of a clustering algorithm as measure of how the algorithm performs compared to the theoretical optimum and showed a theoretical nontrivial lower bound. This lower bound depends on global network parameters. Moreover, it is shown that the greedy algorithms are the best that can be done in polynomial time, given that $P \neq NP$.

Two distributed heuristics are presented by the same authors in [11]. The *Distributed Clustering Algorithm* (DCA) is suitable for clustering "quasi-static" networks whereas the *Distributed Mobility-Adaptive Clustering* (DMAC) can deal with mobility. Both algorithms are message-driven, i.e., except for the initial routine, a specific procedure will be executed at a node depending on reception of the corresponding message.

The fact that DCA uses a generic *weight* associated with each node to measure how desirable it is to the clusterhead position makes DCA a generalization of the previous algorithms. The *weight* drives the clusterhead choice.

In DCA, there are two types of messages: *CH*(v), used by the node v to make its neighbors aware that it is going to be clusterhead, and *JOIN*(v,u). Every node starts the execution running at the same time the procedure *Init*. Only the nodes with the higher weight among their neighbors will send a *CH* message. The other nodes will wait in order to receive these messages.

- *On receiving CH*(u). The node v receiving this message will check whether some other node in the neighborhood with higher weight may send a *CH* message (in other words, no message was already received from that neighbor, neither *CH* nor *JOIN*). The node v will select the neighbor with the higher weight among them. After joining the cluster, the *JOIN* message will be sent.

- *On receiving JOIN*(u,t). The node v checks whether v is a clusterhead and u wants to join its cluster ($t = v$). When all neighbors with smaller weight have joined some cluster, the algorithm is ended.

As already said, DMAC is also a message-driven heuristic. Different from DCA, it is not assumed that during the clustering process the nodes of the network do not move. Instead of just reacting upon reception of messages from other nodes, it also reacts in the case of link failure (possibly caused by movement) or in the presence of a new link. The algorithm is similar to DCA, nevertheless a new clusterhead election may be activated as response to the *New_Link* and *Link_Failure* events.

5.2.2 Dominance Only Approaches

In the literature, there are also one hop clustering algorithms that do not aim at fulfilling the independence property (nevertheless, the dominance is satisfied). This means that clusterheads can be neighbors but every ordinary node has at least one clusterhead as direct neighbor.

5.2.2.1 LEACH Heuristic

The *Low-Energy Adaptive Clustering Hierarchy* (LEACH) [57, 59] is a clustering-based protocol that minimizes the energy dissipation in sensor networks. It has a randomized rotation of clusterheads. Such kinds of approach change dynamically the clusterheads in order to avoid overburden of one node. This may happen because normally the clusterheads have additional responsibility in the network (e.g. to organize the set of nodes). This means that the battery of the clusterheads have a tendency to be faster depleted.

In the LEACH architecture, the clusterhead has the task to coordinate the sleep time of the other nodes, to receive the sensor data from the members of the cluster, to perform data fusion, and to send the result to the base station.

The LEACH algorithm is probabilistic; sensors elect themselves to be local clusterheads at any given time with a certain probability. These clusterhead nodes broadcast their status to the other sensors in the network. Each sensor node determines to which cluster it wants to belong to by choosing the clusterhead that requires the minimum communication energy. The decision of becoming clusterhead depends on the amount of energy left at the node. There is no extra negotiation among the nodes.

To assure that the network, with high probability, will have enough clusterheads to cover it and at the same time does not overestimate this number, the optimal number of clusters is determined a priory.

In each round of the algorithm, a node v chooses a random number $r \in [0,1]$. If $r < T(v)$, the node becomes clusterhead, where $T(v)$ is the threshold to become clusterhead and is given by:

$$T(n) = \begin{cases} \frac{P}{1-P\cdot(r\cdot mod\frac{1}{P})} & \text{if} \quad n \in G \\ 0 & \text{otherwise} \end{cases}$$

Where P is the desired percentage of clusterheads, r is the current round, and G is the set of nodes that have not been clusterhead in the last $\frac{1}{P}$ rounds.

5.2.2.2 GDMAC and MACA Heuristics

In [10], a generalization of the Distributed and Mobility-Adaptive Clustering (DMAC, see [11]) is presented (called GDMAC). A very similar version of the algorithm, called *Mobility-Adaptive Clustering Algorithm* (MACA) is presented in the [13]. Although the MACA algorithm has about the same functionality as the GDMAC, the comparison with the DMAC is not done in this paper.

The idea of the heuristic is to overcome some limitations of DMAC, but keeping its desirable properties. As in DMAC, the nodes here can move during the cluster set up and decide by themselves about their own role based on their current one-hop neighbors. Nevertheless, the independence property of DMAC is relaxed, and now a degree of independence can be selected, i.e., the number of clusterheads that are allowed to be neighbors. Moreover, a new weight-based criterion that allows the nodes to decide whether to change their role depending on the current condition is defined.

The heuristic runs continuously on each node of the network, and the decision about the role of a node has to obey the following constraints:

- Ordinary (members) nodes are affiliated with only one clusterhead.
- For an ordinary (member) u, there is no clusterhead v such that $w_v > w_{clusterhead} + h$, where clusterhead is the current clusterhead of u. w is the current weight of the nodes serving as clusterhead, and $h \in \mathbb{R}$ is a parameter. This constraint says that no member can be affiliated with a clusterhead whose w is much (h) below another neighbor that is also clusterhead.
- A clusterhead cannot have more than k neighboring clusterheads (degree of independence).

Like DMAC (and DCA), the heuristic is message-driven. There are five types of messages: *CH*(v), used by the node v to make its neighbors aware that it is going to be clusterhead, *JOIN*(u,v), used to inform the neighbors that node u is joining the clusterhead v, *RESIGN*(w), to force nodes with weight less than w to leave the clusterhead condition (because the number of clusterheads in the neighborhood is greater then k), *Link_failure*, and *New_link*.

The execution of the algorithm is similar to DCA (and DMAC), but k neighboring clusterheads are tolerated before one of them has to withdraw its condition. Moreover, a member just changes its clusterhead if there is another clusterhead in the vicinity that has a considerably better weight. These two measures result in a drastic reduction of message exchange during the maintenance phase of a comparable DMAC algorithm.

5.2.2.3 The Zonal Clustering

The zonal clustering [33] is based on the dominating set property, but employs following enhancement: In order to allow an easily communication among the clusterheads, facilitating the routing of messages among clusters, the concept of weakly-connected dominating set is used.

Given a graph $G = (V,E)$, the dominating set $S \subseteq V$, the subgraph weakly induced by S is denoted by $< S >_w = (Ngh(S) \cup S, E \cap (Neg(S) \times S)$. $< S >_w$ includes the vertices in S and all of their neighbors as the vertex set. The edges are all of G that are incident to S. A vertex subset S is a *weakly-connected dominating set*, if S is dominating and $< S >_w$ is connected.

The zonal clustering algorithm, a zone size control parameter x controls the size of each zone of the graph. A zone is a connected subgraph of the input network with not more than $2 \cdot x$ vertices. Each zone has a root vertex. The zonal construction algorithm has two levels: intrazonal and interzonal.

The intrazonal level is resposible to construct a weakly-connected dominating set inside the zone. In the interzonal level, the root of a zone adds additional vertices to its weakly-connected dominating set to guarantee that the union of the dominating sets for the individual zones is a weakly-connected dominating set for the whole network [32].

There are other algorithms that even desire a higher connectivity of the clusterheads. The connected dominating set S is a dominating set where there exists a path among any two vertices in this dominating set, and this path is also included in S. The idea is that this dominating set forms a backbone for routing the messages in the network. A survey about such algorithms (among others) can be find in [32].

5.2.3 Multihop Clustering

In this section, the proposed approaches go beyond the search for the maximum independent set of a graph or a dominant set. Instead of just finding clusters with members that are one hop away from the respective clusterhead, the different proposals presented in this section comprise finding multihop clusters with different construction objectives.

In this section, we will first present approaches that aim to create clusters with low diameter. They try to decompose a graph to connected elements with a maximum diameter. Further, the Max-Min D-Cluster Formation [4], which aims at constructing clusters where any node within the cluster is at most *d* hops away from the clusterhead, will be presented. The difference here is that the diameter is not dependent of the number of nodes in the system like in the first approach, where low diameter means $O(log\ n)$. Nevertheless, big clusters may be formed.

Subsequently, other multihop cluster heuristics that pursue other objectives are introduced. For example, the *Expanding Ring* and *Budged Approach* try to divide the ad hoc network into a set of clusters whose sizes are close to a given bound. Beyond this is the *Upper and under bound approach*: an inferior and a superior size limits are given, and the problem is to divide the network into clusters that match the given boundaries.

5.2.3.1 Low Diameter Network Decompositions

In [87], the problem of finding a low-diameter network decomposition was studied in algorithmic graph theory. The decomposition of a graph $G = (V,E)$ is the partition of the vertex set into subsets (called clusters). The idea is to decompose the network into connected clusters, each with a small diameter.

A fast algorithm for low diameter network decomposition was presented in [5].The problem is that the approach considers low diameter as $O(log\ n)$, and this does not bound the cluster size [76].

Because dense networks have nodes with a very high degree, decomposing such networks may result in very large clusters. Although the algorithms presented are very fast and could be applied usefully in ad hoc networks, the fact that the size of the cluster is dependent of the network density is not a very good property.

5.2.3.2 Max-Min D-Cluster Formation

In [4], the issue of constructing the d-hop dominating set in an ad hoc network is addressed. The publication presents a proof of the NP-completeness of the problem for unit disk graphs. In addition, an heuristic to construct a good clustering solution is given.

Given a desired maximum number of hops *d* (from the clusterhead), the heuristic runs for $2d$ rounds of information exchange. During the execution, two arrays are maintained by each node: a WINNER and a SENDER array. The WINNER array stores the *id* of the nodes that wins some round of the algorithm. How the winner in a round is determined will be described later. The SENDER array stores the node that sends the winner *id*.

The algorithm is composed of three phases. They are as follows:

***Floodmax*:** This phase consist of *d* rounds. In a round, each node locally broadcasts its current WINNER value to all of its 1-hop neighbors (in the first round, each node takes its own *id* as current WINNER). After all neighboring nodes have been heard from in this single round, the node chooses the largest value among its own WINNER value and the values received. This is the new WINNER. This new value is stored in the WINNER array, and the node that sent it is stored in the SENDER array. This phase is used by the nodes to propagate the largest node *id*s.

***Floodmin*:** This phase also consists of *d* rounds. It is the same as *Floodmax*, but each node selects the smallest value as its new WINNER. The purpose of this phase is to let the nodes with smaller *id*s reclaim some of their territory.

Selection of Clusterhead: In this phase, each node will locally select the clusterhead based on the WINNER array collected in the previous phase. For that, there are tree rules:

1. Each node checks whether it has received its own node id in the second phase. If it has, it becomes clusterhead and skips the other rules.
2. The nodes look for node pairs, i.e., node *id*s that appear both in the first phase and in the second. From those nodes, it selects the smallest *id* to be its clusterhead.
3. Elect the maximum node *id* in the first phase as the clusterhead for this node.

The characteristics of the heuristic are that it can find good solutions with relative low communication ($O(d)$) and generalizes the dominating set problem.

5.2.3.3 ADBP Heuristic

The adaptive dynamic backbone protocol (ADBP) [67] is a clustering algorithm used to construct a backbone in a WSN. The idea is to have multihop clusters where the clusterheads form a backbone to route the packets in the network. Here, the diameter of the clusters is dynamically adjusted: instead of a fixed distance to the clusterhead, as presented in the Max-Min D-Cluster Formation, the allowed distance to the clusterhead is adapted to the current network conditions.

In networks with low topology changes, large clusters are allowed. This comes from the assumption that the cluster will stay stable. In networks with higher dynamics, the size of the cluster is reduced in order to avoid frequent cluster reconstruction.

The clusterhead election has some similarities to the rating system used in the maximum independent set approaches. The node with the highest rate will become clusterhead. Nevertheless, at the beginning, all nodes are clusterheads. The first step is to exchange beacons in order to discover nodes in the vicinity. Among other information, the beacons contain the distance from the clusterhead, degree of the node and the accumulated error rate (a kind of link metric) to the clusterhead.

Every node checks by itself, based on the neighboring information, whether its rate is higher than the neighboring. The rate is a linear combination of the distance to the clusterhead, degree and accumulated link metric (path to the clusterhead). If some neighbor has a higher rate, the node decides to leave its position as clusterhead and assume the winning node as parent in a cluster tree (clusterhead is the root). Nevertheless, there are two constraints in this clusterhead selection: a hop limit constraint, and a accumulated link metric constraint.

In the case of mobility, if a node detects that its parent has moved out of range, it will do the same thing when the parent violates the constraints: try to find a new parent with higher rate or become again clusterhead.

5.2.3.4 Expanding Ring Algorithm

In [104], an algorithm for bounded size clustering based on an expanding ring search is presented. The algorithm relies on a sequence of rounds. In each round, a variable indicating the maximum hop limit is incremented. The initiator (clusterhead) sends this limit in the beginning of each round. This message is repeated by the receivers after decrementing the maximum hop until it becomes zero. At the end of the round, the clusterhead knows the total number and *id*s of the nodes added in the last layer. After some rounds, eventually, the size bound will be exceeded. When this happens, the clusterhead sends a message containing a list of arbitrary chosen nodes (from the last layer) that should be dropped from the cluster (in order to achieve the bound). This message is simply flooded inside the cluster.

5.2.3.5 Rapid and Persistent Clustering

In [75, 77], two clustering methods aiming at producing clusters with a maximum determined size (i.e. number of member nodes) are presented. The algorithms are more efficient than the expanding ring. The cluster sizes produced should be as close as possible to the specified bound (which we will call here B) in order to limit the total number of clusters. Nevertheless, the bound should not exceeded.

Two algorithms are presented: the *Rapid* and the *Persistent* ones. Both approaches rely on allocating growth budgets to neighbors. This significantly reduces the number of messages exchanged because it allows the cluster to grow based on local decisions rather than involving the initiator at each round.

Both of the algorithms produce clusters of bounded size. The *Rapid* heuristic uses less messages than the *Persistent* one. Nevertheless, it has a poor worst-case analytical performance. The *Persistent* heuristic persistently tries to produce a cluster of the specified bound if possible. Simulations show that this algorithm performs well on average when building a single cluster. The proposed algorithms do not violate the cluster size bound at any time. They generally produce flat rather than deep clusters. Flat clusters means that the format of the cluster resembles a circle, whereas deep clusters have less connections and the shape of a line. This is advantageous because flat clusters lead to smaller end-to-end delays.

The *Rapid* Clustering Algorithm In this algorithm, the initiator starts with a budget of B, then it counts itself and therefore has $B-1$ nodes missing to accomplish the requested cluster size. The initiator distributes the current budget ($B-1$) among its neighbors by sending a message to each one of them. If there are more neighbors than the budget, a subset is arbitrarily selected. When receiving the message, each neighbor counts itself and redistributes the partial budget to its neighboring nodes (except to the respective parent). This process is repeated until the complete budget is exhausted. An example of the running algorithm is depicted in Figure 5.2.

In the example, the cluster bound is 8 nodes, and the result is a cluster of size 6. The initiator (node “A”) allocates a budget of 3 to node “B”. As node “B” is just a leaf, it can only contribute with 1 to the cluster. Nodes that receive a message send acknowledgment to their parents in two situations: the budget is exhausted or they have received acknowledgments from all their children. When the initiator receives messages from all neighbors that it sent a budget to, the algorithm terminates. When the acknowledgments carry extra information like hop count, the clusterhead can compute the size and depth of the cluster.

The *Persistent* Clustering Algorithm Although the *Rapid* algorithm has a low message complexity per cluster (maximum $2 \cdot (B-1)$ messages), it can construct clusters that are very small when compared to the desired size. It is possible to see this in the example showed in Figure 5.2. The *Persistent* algorithm uses more messages, but it improves the worst-case behavior.

As in the *Rapid* version, the *Persistent* algorithm has an initiator that distributes $B-1$ (budget) among its neighbors. All nodes receiving the message count themselves and distribute the remaining budget among their neighbors (except the parent) until the budget is exhausted. The difference of this algorithm is that, when receiving the acknowledgments of the children, each node does not send immediately an acknowledgment to the parent. First it compares the size of its subtrees and compares it to the budget allocated to it. If there is a residual budget, the node distributes it among its neighbors

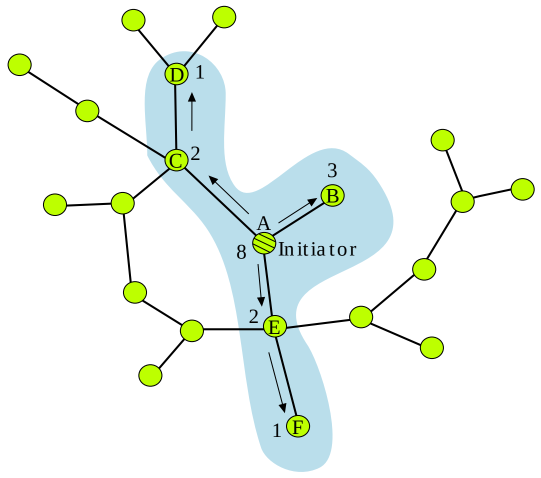


Figure 5.2: Example of execution of the *Rapid* algorithm [75].

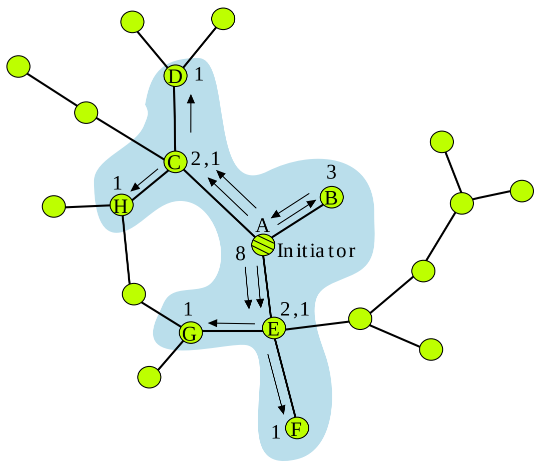


Figure 5.3: Example of execution of the *Persistent* algorithm [75].

that either did not receive any budget previously or met all previously allocated budgets. When the budget is met or no further growing is possible, it returns an acknowledgment to its parent.

When the initiator realizes that the budget was met or no further growth is possible (e.g., no more nodes are connected to the cluster), the heuristic terminates. An example of execution of the algorithm is shown in Figure 5.3. As in the previous example, node “B” cannot allocate the requested budget. Node “A” then realizes that the subtree “B” has consumed just 1 from the budget and re-allocates it among its children (nodes “C” and “E”).

When possible, the *Persistent* algorithm always produces the cluster with the specified size. Wherever this is not possible, it attempts to build the largest possible cluster.

Network decomposition The *Rapid* and *Persistent* algorithms can produce a single cluster of bounded size. To perform a network decomposition in clusters of bounded size, a systematic way of electing the initiators (clusterheads) that will start the decomposition process must be used.

In [77], the following method to elect the clusterheads is presented. Each node that comes up waits for clustering messages from the neighborhood. When a timeout (previously configured) occurs, the node becomes an initiator (or in our terminology, a clusterhead) and invokes one of the two clustering algorithms (in fact, the authors argue that this method can also be used in the *Expanding Ring* algorithm). This process is repeated in several (random) places in the network until the complete network is clusterized.

In order to reduce the complete network decomposition time and at the same time to shrink the number of initiators active at the same time, the paper presents a proposal of initialization of the timeout in order to achieve a good trade-off between the two presented aspects. If the times when the initiators become active are set too far apart, the total time of the network decomposition will be large. In the other hand, when several initiators are concurrently active in a neighborhood locality, some initiators will produce clusters of size smaller than the specified bound.

Some problems of both algorithms are the assumption that the network is static and the fact that the heuristics do not attempt to rank the links in order to select the best connected nodes to form the cluster. Moreover, the initiators are also randomly chosen, in contrast to our two heuristics presented in the next section, where clusterheads are carefully selected based on their fitness for the clusterhead role.

5.2.3.6 Upper and Lower Bound Approach

In [8], a clustering scheme to create a hierarchical control structure is presented. The exact clustering problem is formally described in the publication . Given the graph $G = (V,E)$ and a positive integer $1 \leq q \leq |V|$, the problem is to find the clusters $V_1, V_2, .., V_n$ with the following conditions:

1. All nodes should be included in at least one cluster ($\cup_{i=1}^{n} V_i = V$).

2. Each cluster should be connected ($G[V_i]$, and the subgraph induced by V_i is connected.

3. All clusters should have a minimum (called q) and a maximum size constraint $2 \cdot q$ ($q \leq |V_i| \leq 2q$).

4. Any two clusters should have small overlap ($V_i \cap V_j \approx O(1)$).

5. A vertex should belong to a constant number of clusters ($S(v) \approx O(1)$, $S(v) = \{V_i | v \in V_i\}$).

6. Clusters should be stable across node mobility.

An important remark is that in this approach, as opposed to our basic clustering problem, a small overlap is allowed.

In their paper, the authors analyze how the allowed topologies may influence the feasibility of the desired properties described above. For example, for a complete graph, the condition number five can be violated, e.g. in a star graph, for $k \geq 2$, the center must be included in all clusters, i.e., $S(v) = O(\frac{n}{q})$.

For the rest of the paper, the authors just analyze the properties for graphs with a special topology, modeled by the *Disk Graphs Model* [35, 66]. A *Disk Graph* is a communication model that consists of a value $R \geq 0$ and a graph $G = (V,E)$ embedded in an euclidean plane. The edges are defined as follows: for each two vertices in G, iff the distance between them is less or equal than R, there is a $\{u,v\}$ in E. If $R = 1$, the model is called *Unit Disk Graph*. The model considers that with omnidirectional antenna and fixed power, transmitted packages can be received successfully just inside a given circle. This is not really true in practical cases, but the *Disk Graph* provides a simple model for theoretical analysis.

With an *Unit Disk Graph*, dense star topologies are not allowed, which avoids the problem exposed in the previous paragraph. It is proven in the paper that even with *Unit Disk Graph*, the requirement four could be violated for certain cases. In order to avoid this, the third constraint is altered to:

3a. $\forall i, |V_i| \leq 2q$

3b. $\forall i$ except one, $|V_i| \geq q$, i.e., only one cluster smaller than q is allowed.

With this new constraint, the algorithm described in the work is able to meet all constraints for *Disk Graph* models.

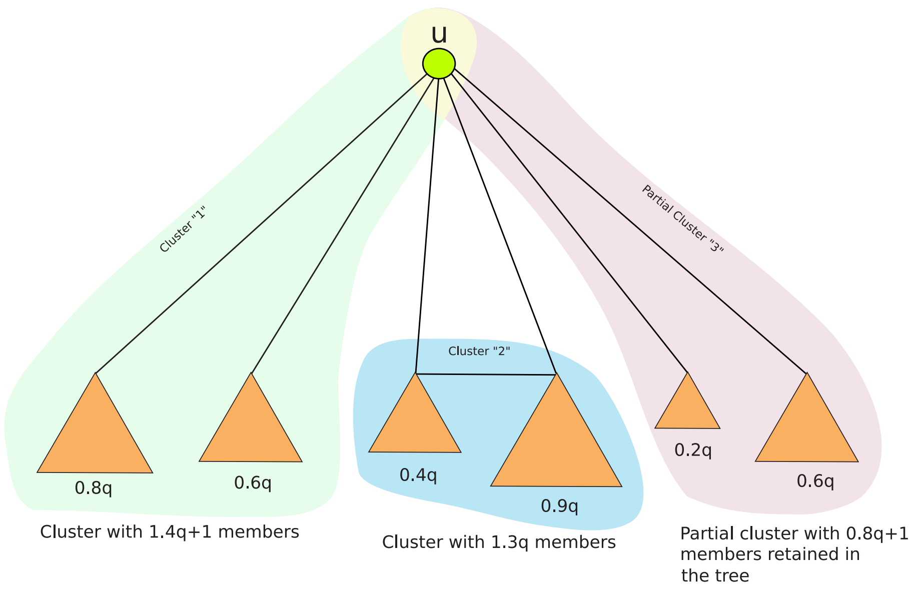

u
Cluster "1"
Cluster "2"
Partial Cluster "3"
0.8q
0.6q
0.4q
0.9q
0.2q
0.6q
Cluster with 1.4q+1 members
Cluster with 1.3q members
Partial cluster with 0.8q+1 members retained in the tree

Figure 5.4: An example of execution of the upper lower bound algorithm.

Clustering Algorithm The first step of the algorithm is to find a rooted spanning tree of the graph G using Breadth-First-Search tree in order to bound the diameter of the tree. Let T be this tree and $T(u)$ be the subtree rooted by the vertex u. $C(u) = \{u_1, u_2, .., u_l\}$, denotes the children of vertex u in the tree.

Now, a node u where $T(u) \geq q$ and for $i = 1, 2, .., l$, $T(u_i) < q$ is selected. The idea is now to form clusters from the subtrees of u. The algorithm selects successively subtrees until the size reaches w, where $q-1 < w < 2q-2$. It is important to remark that, at the end, the last subtrees whose sum do not achieve the needed size do not form a cluster.

All nodes belonging to some cluster are then deleted from the tree T and the process is started again. In practice, the tree T is created and then post-order transversed to find the node u ($T(u) \geq q$, $i = 1, 2, .., l$, $T(u_i) < q$). Then, subtrees are selected taking into consideration the connection between the $C(u)$, because wherever the subtrees are connected through $C(u)$, the node u does not need to be included in the cluster. For subtrees starting at children from u that aren't connected, the node u must be included in order to assure the second constraint. It is easy to see that the clusters will have at most one common node (u in the case).

An example of execution of the algorithm is depicted in Figure 5.4. In this step, the node u holds the conditions $T(u) \geq q$, $i = 1, 2, .., l$, $T(u_i) < q$. Then, two clusters are formed: "cluster 1" has size $1.4q + 1$ because it must include the node u, since it is not known whether the subtrees are connected. The "cluster 2" is formed by two subtrees whose roots are connected by a link. Therefore, the node u is not necessary in this cluster. The "cluster 3" is not complete because the minimum amount of nodes q has not being reached. Therefore, just clusters one and two are deleted, and the algorithm goes ahead.

5.2.4 Other Approaches

There are also other clustering approaches that do not fit in any other presented category. For example, in [74], a method that clusters are formed without clusterheads is presented. A *clique* in graph $G =$

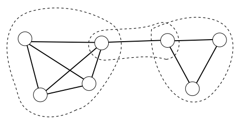


Figure 5.5: Example of output of the clique-based clustering.

(V,E) is a subset S of V, whose induced subgraph is complete. The clustering algorithm decomposes the network in the maximal cliques as clusters [32]. Overlapping clusters are allowed and nodes that are members of more than one cluster are called boundary nodes. They are responsible for the communication among the different clusters. Figure 5.5 shows an example of a clique-based clustering.

5.2.5 Discussion

The maximum independent set and the dominating only approaches concern the division of the network in one hop clusters. This is different from our approach, where multi-hop clusters are allowed. Moreover, there are few approaches that take into account the link quality when constructing the cluster. The WCA heuristic makes a very simple link assessment. In our approach, a much more elaborated link metric is used. The MOBIC heuristic uses a simplified link metric too, but for the purpose of mobility assessment. Stable nodes are desired as clusterheads. The used link metric just uses the received signal streght indication. The VDBP clustering construction method uses the link failure rate, but again, just for guessing the mobility pattern of the nodes.

Another interesting aspect presented in *WCA* is the combined metric weighting used to select the clusterhead. In our algorithms, we use also combined metric weighting. Nevertheless, differently from this approach, we use metrics for clusterhead and members election. Moreover, the links are also rated with a much more elaborated link metric. Our metrics use a larger number of parameters than the ones presented here.

In the approaches based on independent and/or dominating set, the membership selection does not rate the possible members with a fitness metric as our approach does.

In the state of the art, we presented also multi-hop clustering strategies pursuing different objectives. The Max-Min D-Cluster Formation aims at finding clusters with a maximum number of hops d from the clusterhead. There is no distinction of the link quality when selecting cluster members. Moreover, differently from our approach, the size of the cluster is uncontrolled. Dense network areas result in bigger clusters than sparse ones. In the ADBP heuristic, variable cluster diameters are allowed. The diameter of the cluster is controlled by the mobility of the network: when small topology changes are detected, the clusters are larger than when larger topology changes are observed. The idea is that a very high cluster reconstruction overhead is necessary when the network is experiencing extensive topology changes and the clusters are large. Nevertheless, our approach tries to construct clusters with a minimum amount of resources, different from this approach. Moreover, this approach uses also a simplistic link metric for the clusterhead election.

A little bit more in the direction of this work are the *Rapid* and *Persistent* algorithms. They have an objective bound B and try to produce clusters achieving this bound. However, the bound is just given in number of nodes and there is no way to differentiate nodes. Moreover, the clusters are always smaller or equal to the given size (bound). In our approach, all clusters have at least a specified amount of resources (as can be seen in the next section).

In the *Rapid* and the *Persistent* algorithms, the clusterheads are elected in a completely random

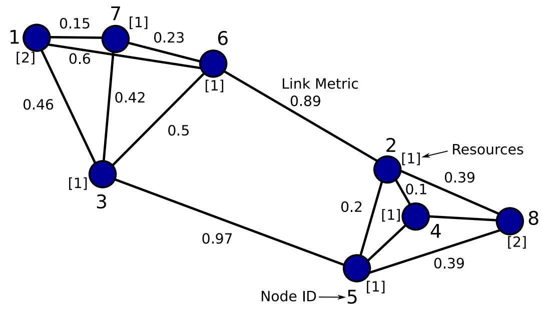


Figure 5.6: Example of ad hoc network model with weighted links and nodes

fashion, which leads to the selection of nodes that are not very suitable for the role. In our approaches we use the opposite approach: strongly connected nodes with plenty energy have a higher probability to be selected as clusterheads. Another difference is related to the links: the *Rapid* and the *Persistent* heuristics do not attempt to rank the member candidates' (concerning, for example the links) in order to select the best connected nodes to form the cluster.

The upper and lower bound approach try to keep the amount of nodes in the clusters inside a specified interval. But different from our approach, overlaps are allowed. Moreover, the link quality is also not relevant to the heuristic.

Another very important difference between all existing approaches and the one presented in this work is the fact that we try to minimize the communication overhead among all nodes inside a cluster. For that, as it will be presented in the next section, we use the smallest distance between each pair of nodes inside the clusters for the objective function. This distance is calculated by means of our combined link metric.

A final comment is also important: besides the different approaches presented in this state of the art, there are several theoretical approaches of graph clustering and partitioning. Besides Section 5.2.3.1 that goes a little bit in this direction, we have concentrated on more practical approaches in this survey, i.e. approaches for mobile ad hoc networks.

5.3 Problem Definition

In this section, a formal definition of our exact clustering problem is described. Moreover, a proof of the NP-hardness of the problem is also given.

We call our problem *minimum intracommunication-cost clustering*.

The ad hoc network is modeled by an undirected graph $G = (V, E)$, where V is the set of wireless nodes and an edge $\{u, v\} \in E$ if and only if a communication link is established between node $u \in V$ and $v \in V$. The two nodes in this case are neighbors. Each node $v \in V$ has an unique identifier (ID_v).

For each link, a weighing function assigns a positive weight. $w : E \longrightarrow \mathbb{R}^+$. This weight measures the quality of a wireless link (for details see the *virtual distance* concept). We define for each edge not in the graph ($\{u, v\} \notin V$), $w(u, v) = \infty$.

For each node, an additional weighing function r is responsible for characterizing the amount of resources available in the node. $r : E \longrightarrow \mathbb{R}^+$. This models the resource capacity of the node.

An example of a simple network with link and node weights corresponding to the link quality and resource availability is shown in Figure 5.6.

The clustering process partitions the nodes into *clusters*, each one with a *clusterhead* and possibly some *ordinary nodes*. As presented in the related work section, there are several different types of clustering strategies pursuing different objectives.

In our problem, the objective is to get multihop clusters with enough resources for the OS and application processing. Moreover, the minimization of the intra-cluster communication cost is desired.

This optimization problem is modeled as following:

Input: A graph with weighted nodes and links (G,w,r) and a resource requirement $q \in \mathbb{R}^+$, where the sum of all node weights in each cluster must be greater or equal to q

Constraints: For every input instance (G,w,r,q), $\mathscr{M}(G,w,r,q) = \{C_1,C_2,..,C_k | C_k$ is the k^{th} cluster configuration$\}$, where the following properties hold
$C_k = \left\{c_{k1}, c_{k2}, .., c_{k(nk)}\right\}$ is the k^{th} possible cluster configuration of the graph, where $k = \{1,2,..,n\}$ (n is the number of possible configurations, nk is the number of clusters in the k^{th} configuration, $nk = |C_k|$)
$c_{ki} = \left\{v^1_{ki}, v^2_{ki}, .., v^{|c_{ki}|}_{ki}\right\} \in Pot(V)$ is the i^{th} cluster of the k^{th} configuration, where v^j_{ki} is the j^{th} element of the cluster c_{ki}
For each configuration C_k, $k = 1,2,..,n$, the following properties must hold:

1. $\bigcup_{i=1,2,..,nk} c_{ki} = V$ (cluster definition constraint)
2. $\bigcap_{i=1,2,..,nk} c_{ki} = \emptyset$ (no overlapping constraint)
3. Let $P(u,v) = \left\{p^{(u,v)}_1, p^{(u,v)}_2, .., p^{(u,v)}_m\right\}$ be the set of all possible paths between nodes u and v. $p^{(u,v)}_h \in Pot(E)$ is the h^{th} possible path where:
$p^{(u,v)}_h = \left\{\{u,x^h_1\}, \{x^h_1,x^h_2\}, .., \{x^h_{g-1},x^h_g\}, \{x^h_g,v\}\right\}$, $x^h_f \in V$, $f = 1,2,..,g$, $g \in \mathbb{N}$
For each $\{u,v\} \in E \wedge u,v \in c_{ki}$, $i = 1,2,...,nk$, $\exists p^{(u,v)}_h \in P(u,v) | x^h_f \in c_{ki}$ for $f = 1,2,..,g$. (Connectivity constraint)
4. $\sum_{j=1}^{|c_{ki}|} r(v^j_{k_i}) \geq q$, for each $i = 1,2,...,nk$ (minimum amount of resources per cluster)

Costs: For every cluster configuration $C_k = \{c_{k1}, c_{k2}, .., c_{k(nk)}\} \in \mathscr{M}(G,w,r,q)$, the cost is given by:

$$cost(C_k,(G,w,r,q)) = \sum_{i=1}^{nk} \sum_{u,v \in c_{ki}} \frac{1}{2} \cdot D_{c_{ki}}(u,v) \cdot \big(\alpha \cdot r(u) + (1-\alpha)\big) \tag{5.1}$$

Where $D(u,v)$ is the virtual distance between $u,v \in V$. $D_{c_{ki}}(u,v)$ is the virtual distance between u,v using just edges that are inside the cluster c_{ki}. Note that $\forall v,u \in c_{ki}, D_{c_{ki}}(u,v) = D(u,v)$ iff the cluster c_{ki} is a convex cluster, i.e., the global shortest path between any two nodes in the clustering must use just links inside the cluster. $\alpha \in [0,1]$ controls how much the amount of resources influences the distance metric. For $\alpha = 0$, just the distances between cluster members enter into the metric; $\alpha = 1$ means that nodes with n times more resources have an n times stronger influence.
Now, we define how the virtual distance is calculated. First, we introduce the cost of a path:

$$PCost(p^{(u,v)}_h) = w(u,x^h_1) + \sum_{f=1}^{g-1} w(x^h_f, x^h_{f+1}) + w(x^h_g, v)$$

The virtual distance between u and v is the cost of the shortest path:

$$D(u,v) = PCost(p^{(u,v)}_h), \text{ where } PCost(p^{(u,v)}_h) = \min_b \left(PCost(p^{(u,v)}_b)\right), \text{ for } b = 1,2,..,m \tag{5.2}$$

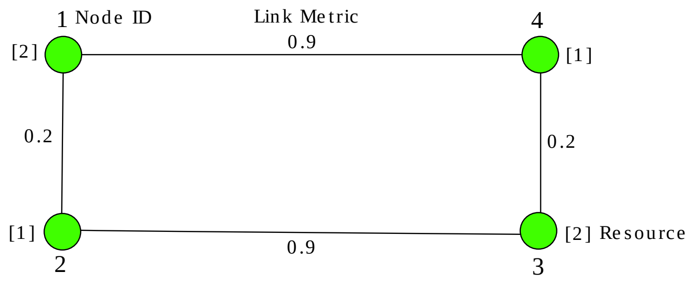


Figure 5.7: Simple network clustering example

The virtual distance using just nodes inside the cluster is defined by:

$$D_{c_{ki}}(u,v) = PCost(p_h^{(u,v)}), \text{ where } p_h^{(u,v)} \in P(u,v) | x_f^h \in c_{ki} \text{ and } PCost(p_h^{(u,v)}) = \min_b \left(PCost(p_b^{(u,v)}) \right), \text{ for } b = 1,2,..,m$$

Goal: *Minimum*, i.e. $min_k\{cost\,(C_k,(G,w,r))\,,\text{ for } k=1,2,..,n\}$

To better clarify the definitions an example is presented. Consider the graph $G = (V,E)$ with $V = \{v_1,v_2,v_3,v_4\}$, $ID_{v_d} = d$ and $E = \{\{v_1,v_2\},\{v_2,v_3\},\{v_3,v_4\},\{v_4,v_1\}\}$. G is shown in Figure 5.7. The function $w : E \rightarrow \mathbb{R}^+$ rated the links and in this case is defined as:

$$w(e) = \begin{cases} 0.2 & \text{if} \quad e = \{v_1,v_2\} \\ 0.2 & \text{if} \quad e = \{v_3,v_4\} \\ 0.9 & \text{if} \quad e = \{v_1,v_4\} \\ 0.9 & \text{if} \quad e = \{v_2,v_3\} \end{cases}$$

where $e \in E$.

The resource function ($r : V \rightarrow \mathbb{R}^+$) returns the resource availability of each node and in the example is defined as:

$$r(v) = \begin{cases} 2 & \text{if} \quad v = v_1 \\ 1 & \text{if} \quad v = v_2 \\ 2 & \text{if} \quad v = v_3 \\ 1 & \text{if} \quad v = v_4 \end{cases}$$

where $v \in V$

Our objective in this example is to find the clustering with the objective of minimizing the intra-cluster communication. The problem input is (G,w,r,q) where $q = 3$, i.e. we want at least 3 resource units in each cluster. For this problem, we use $\alpha = 0$.

The **set of feasible solutions** for our input $\mathscr{M}(G,w,r,q)$ is:

$$\mathscr{M}(G,w,r,q) = \{C_1,C_2,C_3\} = \left\{\{\{v_1,v_4\},\{v_2,v_3\}\},\{\{v_1,v_2\},\{v_3,v_4\}\},\{\{v_1,v_2,v_3,v_4\}\}\right\}$$

This set of solutions is depicted in Figure 5.8.

The following clusters, for example, are not part of the valid solutions:

$$\left\{\{\{v_1\},\{v_2,v_3,v_4\}\},\{\{v_1,v_3\},\{v_2,v_4\}\},\{\{v_1,v_2,v_3\},\{v_2,v_3,v_4\}\},..\right\} \nsubseteq \mathscr{M}(G,w,r,q)$$

because they violate some of the constraints, e.g.:

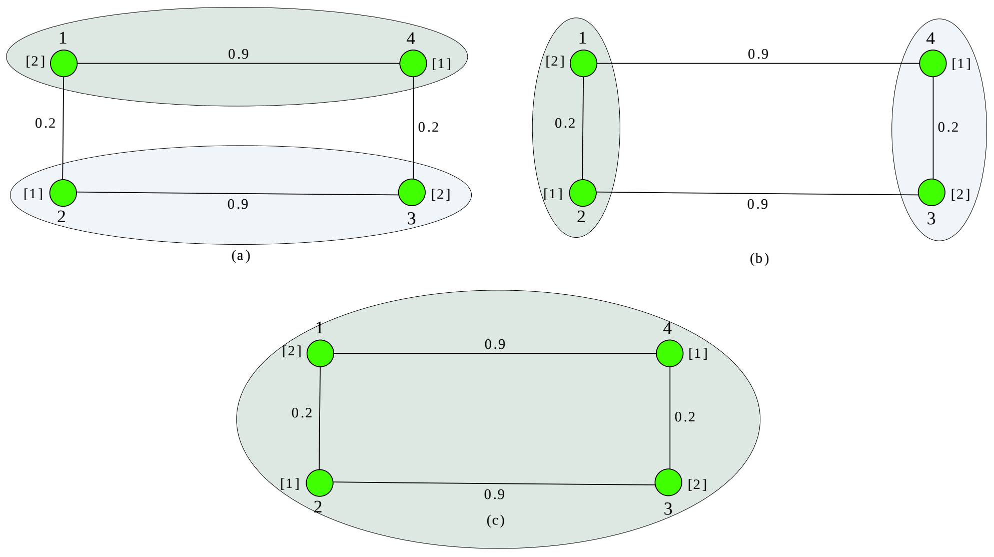


Figure 5.8: **Set of feasible solutions** $\mathscr{M}(G,w,r,q)$ for the input (G,w,r,q).

- $\{\{v_1\},\{v_2,v_3,v_4\}\}$: The minimum amount of resources per cluster $q=3$ (rule 3) is violated because $\sum_{j=1}^{|c_{11}|=1} r(v_{1_1}^j)=2$, note that $(v_{1_1}^1=v_1)$.
- $\{\{v_1,v_3\},\{v_2,v_4\}\}$: The same problem as above
- $\{\{v_1,v_2,v_3\},\{v_2,v_3,v_4\}\}$: The rule 2 is violated, because $\{v_1,v_2,v_3\}\bigcap\{v_2,v_3,v_4\}\neq\emptyset$

The costs of the valid solutions are:

$$
\begin{aligned}
&cost(C_1,(G,w,r,q)) = \textstyle\sum_{i=1}^2\sum_{u,v\in c_{1i}}\frac{1}{2}\cdot D_{c_{1i}}(u,v) = D_{c_{11}}(v_1,v_4)+D_{c_{12}}(v_2,v_3) = 1.8\\
&cost(C_2,(G,w,r,q)) = \textstyle\sum_{i=1}^2\sum_{u,v\in c_{2i}}\frac{1}{2}\cdot D_{c_{2i}}(u,v) = D_{c_{21}}(v_1,v_2)+D_{c_{22}}(v_3,v_4) = 0.4\\
&cost(C_3,(G,w,r,q)) = \textstyle\sum_{i=1}^2\sum_{u,v\in c_{3i}}\frac{1}{2}\cdot D_{c_{3i}}(u,v) = D_{c_{31}}(v_1,v_2)+D_{c_{31}}(v_1,v_3)+\\
&+D_{c_{31}}(v_1,v_4)+D_{c_{31}}(v_2,v_3)+D_{c_{31}}(v_2,v_4)+D_{c_{31}}(v_3,v_4) = 4.4
\end{aligned}
$$

Therefore, the *minimum* (i.e. $min_k\{cost(C_k,(G,w,r))$ for $k=1,2,..,n\}$) is C_2, that is showed in Figure 5.8b.

The optimal solution for the network depicted in Figure 5.6 is presented in Figure 5.9.

5.3.1 Problem Properties

Let us analyze some properties of our problem. When the constraint number three is not considered, a property of the optimal solution of the clustering problem is that, for some q and some $r: r\rightarrow]0,q[$, the smallest possible cluster size is obviously q and the biggest possible size in the worst case is $2q-\varepsilon$, $\varepsilon>0$. This is because whenever a cluster c_i achieves a size bigger than $2q$, it can be divided in clusters c_{i_a} and c_{i_b}, saving the cost of the paths among nodes belonging to c_{i_a} and nodes belonging to c_{i_b}.

If we consider again the constraint number three, the biggest possible size in the worst case turn to be the complete network (with cost: $\sum_{u,v\in V} D(u,v)$). An example where $\sum_{v\in V} r(v)\gg q$ and even so the complete network should be a cluster is shown in Figure 5.10.

It is important to remark that the *minimum intracommunication-cost clustering* is an NP-hard problem (even for unit-disk graphs).

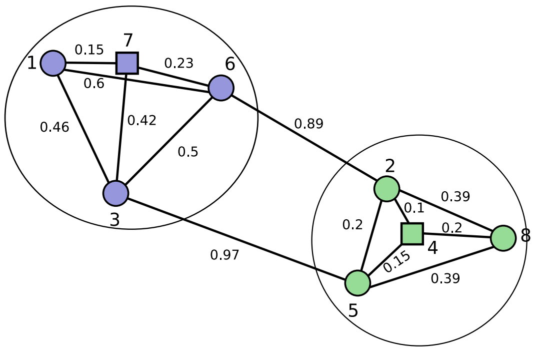


Figure 5.9: The resulting clustering of the network presented in Figure 5.6.

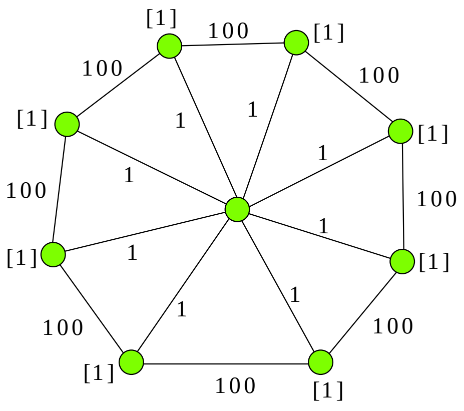


Figure 5.10: An example of network where for $q > 1$ the optimal solution is one cluster (complete network).

Proof. For the proof, we will use the *partition problem*.

Partition Problem. Given a multiset of positive integers, $M = \{i_1, i_2, .., i_n\}$, $n \in \mathbb{N}$, the problem is whether a subset of the multiset exist ($s \subset M$) where, for $k, l \in M$: $\sum_{k \in s} k = \sum_{l \notin s} l$

This means it is possible to divide the multiset M in two groups with the same sum. It is known that the *Partition Problem* is NP-complete.

We will now reduce the *Partition Problem* to our clustering problem (*Partition Problem* $\leq_p$ *minimum intracommunication-cost clustering*).

The first step is to change our clustering from an optimization problem to a decision problem. This is done by altering the goal:

Goal: Is there a solution with the total communication cost less than d? Formally, for $k = 1, 2, .., n$, $\exists cost(C_k, (G, w, r)) < d$?

For each instance M of the *Partition Problem*, a *Minimum intracommunication-cost clustering* problem instance (G, w, r, q) with the following characteristics will be constructed:

- The graph $G = (V, E)$ is a complete graph, where $V = \{v_1, v_2, .., v_j\}$ ($|V| = j$) and $E = \{e_1, e_2, .., e_{\binom{j}{2}}\}$.
- $|V| = |M|$, i.e., the number of vertices is the same of the number of positive integers in M.
- The function $w : E \longrightarrow \mathbb{R}^+$ is defined as $w(e \in E) = 1$, i.e., all edges have unitary weight.
- The resource function $r : E \longrightarrow \mathbb{R}^+$ is defined as $w(v_g \in E) = i_g \in M$, i.e., each vertice becomes the weight of an element of the multiset M.
- As minimum resource request for each cluster, we select $q = \frac{1}{2} \cdot \sum_{g=1}^{n} i_g$
- And finally, we select $\alpha = 0$.

The decision question is now, for $k = 1, 2, .., n$:

$$\exists cost(C_k, (G, w, r)) < \binom{j}{2} = |E|?$$

If a clustering solution exists, then a partition in M exist. The clustering solution is composed by two clusters c_1 and c_2, where for any $y \in s$ and for any $z \in (M - s)$, $y \in c_1$, $z \in c_2$ and $c_1 \cap c_2 = \emptyset$.

$\square$

In realistic environments, a constraint that may appear is that the amount of resources of a node must fall in a specific range, i.e., $\forall v \in V, r(v) \in [L_{inf}, L_{sup}], L_{inf}, L_{sup} \in \mathbb{R}^+, 0 < L_{inf} \leqslant L_{sup}$. Even in this case, the problem is still NP-hard.

Proof. The proof here is very similar to the previous one. The *Partition Problem* is also used. We will now reduce the *Partition Problem* to our clustering problem (*PartitionProblem* $\leq_p$ *constrained minimum intracommunication-cost clustering*).

For each instance M of the *Partition Problem*, a *Constrained minimum intracommunication-cost Clustering*) problem instance (G, w, r, q) with the following characteristics will be constructed:

- $|V| = \sum_{h=1}^{n} i_h$, i.e., the number of vertices is the same of the sum of all positive integers in M.
- The resource function $r : E \longrightarrow \mathbb{R}^+$ is defined as $w(v \in E) = L_{inf}$, i.e., each vertice becomes the weight of the inferior allowed limit.

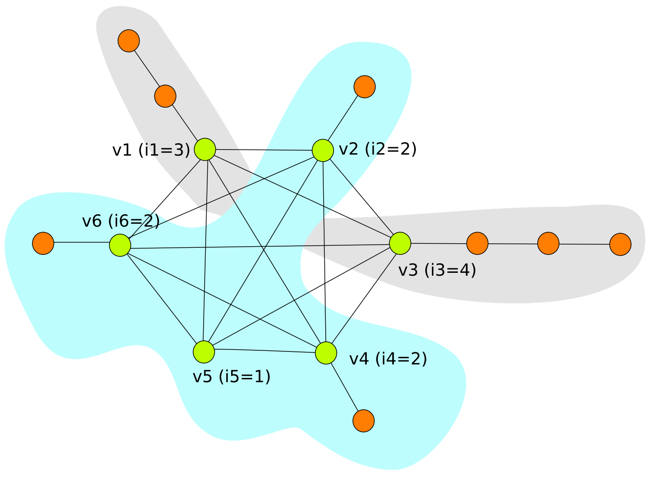


Figure 5.11: Example of graph construction. Notice that if a partition of M exists, the solution of the problem will be two connected clusters with the same number of nodes ($q = 7$ nodes) as can be seen in Figure.

- The graph $G = (V,E)$ where $V = \{v_1, v_2, .., v_j\}$ ($|V| = j$) is constructed as following:
  - $C = \{v_1, v_2, .., v_m\}$ (where m is the number of integers in the problem M) forms a clique in the graph
  - The rest of the nodes are added to the vertices according to the value of the integer i_g $1 \leqslant g \leqslant m$. Node v_g is connected to a chain of $i_g - 1$ nodes. An example of this graph construction for $M = \{3,2,4,2,1,2\}$ is depicted in Figure 5.11.
  - The function $I : C \rightarrow \mathbb{N}$ returns the correspondent integer of the problem M ($i_g = I(v_g)$).
- The function $w : E \rightarrow \mathbb{R}^+$ is defined as $w(\{u,r\}) = 1$ if $u \neq r$, $u \in C$ and $r \in C$, i.e., all the edges in the complete graph have unitary weight. For the other vertices in the graph (vertices forming the chain) $w(v) = 0$.
- As minimum resource request for each cluster, we select $q = \frac{L_{inf}}{2} \cdot \sum_{g=1}^{m} i_g$
- And finally, $\alpha = 0$

The decision question is now, for $k = 1,2,..,n$:

$$\exists cost(C_k, (G,w,r)) < \frac{1}{2} \sum_{u \in C} I(u) \cdot (j - I(u))?$$

$\square$

5.4 Division of Labor and Task Allocation in Social Insects

The cluster construction approach presented in this chapter is based on a particular kind of self-organization: the division of labor and task allocation in swarms of social insects, described in detail by Bonabeau et al. [20]. In social insects, different tasks are performed by specialized individuals. It

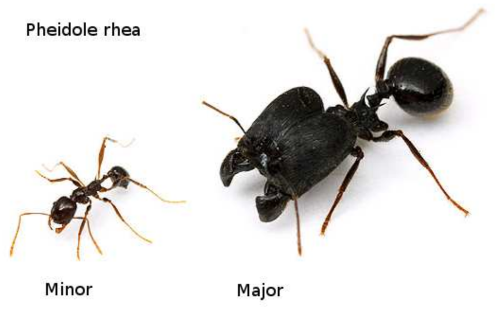


Figure 5.12: *Minor* and *Major* subcastes of the workers in the *Pheidole rhea* specie. Image source: [129]

is highly probable that specialized task performance is more efficient than sequential task execution of unspecialized individuals.

All the different types of social insects have division of labor. The most basic level of division of labor is the reproductive division: only a small part of the insects are involved in reproduction tasks. Nevertheless, often a further division of labor exists.

The approach of cluster construction presented in this thesis is based on a sub-form of division of labor called *worker polymorphism*, i.e., the workers have different morphologies. Each of the different morphological castes tends to perform a different task in the colony.

The division of labor is normally not rigid, it exhibits plasticity [105]. Changes in the environment or in the internal structure of the colony make adjustments in the allocation of tasks necessary, which is possible due to the plasticity of individual workers.

An experiment made by Wilson [130] in the ant species from the *Pheidole* genus examined the worker polymorphism. There are two morphological subcastes in the workers: the minors, which are responsible for the quotidian tasks of the colony, and the majors, mainly responsible for seed milling, abdominal food storage, or defense, normally known as "*soldiers*". Figure 5.12 shows an example of a *major* and a *minor* of the specie *Pheidole rhea*.

In the experiment, Wilson altered the structure of colonies from the *Pheidole* genus. The *majors* exhibits elasticity, i.e. the behavior repertoire could be stretched back and forth in a predictable manner in response to perturbations. The perturbation inserted by Wilson in his experiment was to change the ratio between *majors* and *minors* in a colony. Reducing the amount of *minors*, the *majors* start to execute tasks that were almost exclusive for the *minors* in the normal situation. Wilson suggested that the colony as a whole exhibits resilience, the degree of response to alterations was determined by the elasticity of the individual ants. Therefore, the resilience of task allocation accomplished at colony level is linked with the elasticity of individual workers.

The demand response behavior demonstrated by Wilson was formally modeled by Bonabeau et al.[18]. In the model, individuals have a response threshold for every type of task. The task-associated stimulus controls the engagement of the individuals in a specific task. When the stimulus for a certain task rises, the probability that individuals will react to that stimulus and perform the task will increase. This probability of task engagement depends on the threshold of a certain individual with respect to the requested task.

Formally, the model used has the following components:

- Let s_a be the stimulus associated with the task a

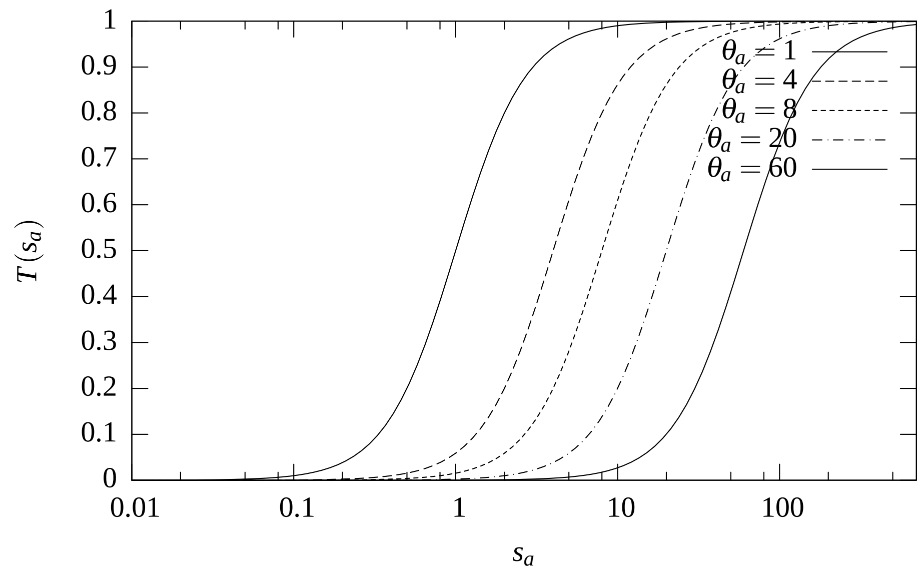


Figure 5.13: Some threshold response curves with different thresholds ($\theta = 1, 4, 8, 20, 60$).

- Let θ_a be the threshold of an individual associated with the task a. It determines the tendency of an individual to respond to a stimulus such that $s_a \ll \theta_a$ for low response probabilities and $s_a \gg \theta_a$ for high response probabilities.

The selected response function that brings a similar behavior to the one observed by Wilson is:

$$T_{\theta_a} = \frac{s_a^\beta}{s_a^\beta + \theta_a^\beta} \tag{5.3}$$

T_{θ_a} is the probability of performing the task a as a function of the stimulus intensity s_a and $\beta > 1$ determines the steepness of the threshold. Normally, in the experiments, the value $\beta = 2$ was used. Figure 5.13 shows the different response probabilities given some thresholds to perform the task a. In this figure, the meaning of θ could be easily recognized: it regulates the response of an individual to a stimulus. When $\theta_a = s_a$, the probability of performing the task a is exactly $\frac{1}{2}$.

Monte Carlo simulations showed that this function brings a similar behavior as the one observed in Wilson's experiment. In Figure 5.14 a schematic representation of hypothetical response curves for minors and majors in the polymorphic species of ants studied by Wilson are shown. The figure represents the threshold for typical minor jobs.

5.5 Heuristics Basic Concepts

Our approach for cluster construction is based on several examples of task allocation coming from nature (i.e., we developed a cluster construction based on division of labor in social insects). We aim at mapping these behaviors to a consistent heuristic that finds and maintains clusters with a maximum defined q during the runtime of an ad hoc (and sensor) network.

The possible "castes" (or roles) that a node can assume are:

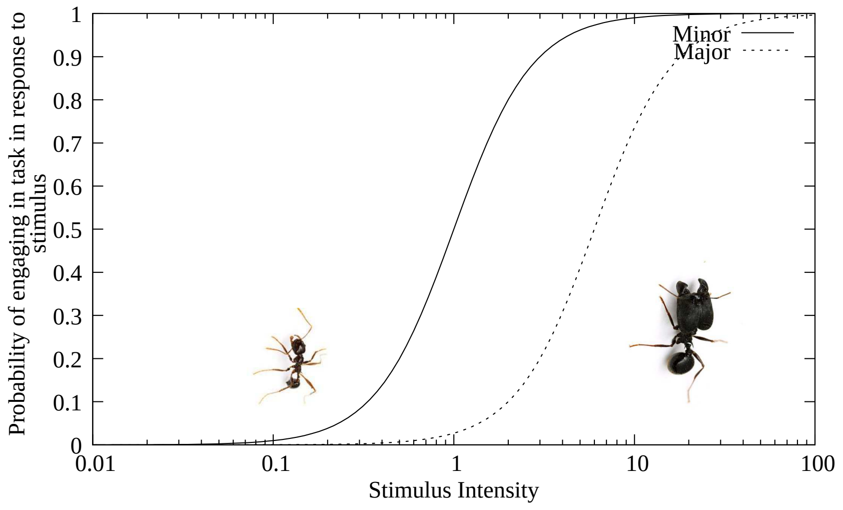


Figure 5.14: Hypothetical response curves for minors and majors [19].

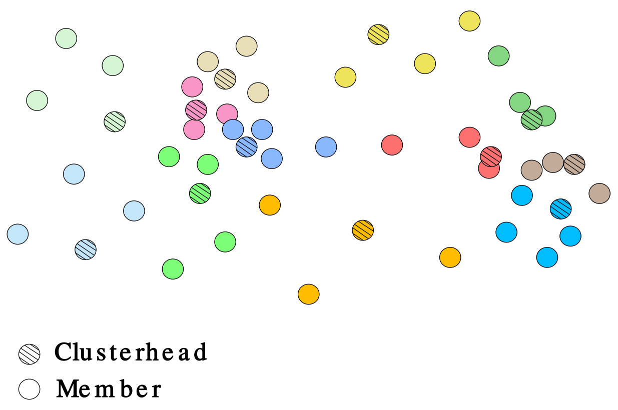


Figure 5.15: Example of a good solution of task allocation in an ad hoc network ($q = 4$).

Clusterhead (CH): The clusterhead nodes are the representatives of the clusters. The identification of the cluster is given by the clusterhead, moreover special tasks are assigned to the clusterhead. Once the clusterhead is not present in a cluster anymore, the cluster ends its existence.

Member (Me): The members of the cluster are the nodes that have decided which cluster they belong to.

Ordinary Node (Not member, Nm): Nodes that do not decide to enter into a cluster neither become clusterhead.

A network where all nodes are already clustered is shown in Figure 5.15.

We have developed two heuristics that implement the clustering in a given ad hoc networks. The first of them is designed for “quasi-static” networks, where the nodes do not move or move very slowly. The second one was developed for sensor networks with certain movement patterns.

5.5.1 General Ideas

The idea of the clustering heuristic is that each node has probabilistic tendencies to assume a determined role in the network. For example, nodes with good connectivity and plenty of energy are good candidates to be clusterheads and their clusters will probably have a small communication cost. In the same way, poorly connected nodes with low energy level are not good clusterhead candidates, and should stay as cluster members. This idea is derived from the division of labor of social insects. Instead of having just a certain number of fixed morphology agents (like the *majors* and *minors* in the *Pheidole* genus), we have here the complete spectrum of nodes: from nodes very capable of assuming the clusterhead role to nodes not suitable at all for this task. They all have a probability of assuming a determined function based on their fitness to the specific role and the actual *necessity* (stimulus) that a determined role has in the network. The fitness to assume a role is modeled as θ_{CH}, i.e., the threshold to become clusterhead and θ_{M_n}, i.e., the threshold to become member of the cluster $n \in \mathbb{N}$. The stimulus to become clusterhead is called s_{CH}.

5.6 Clustering “Quasi-Static” Ad hoc Networks

The heuristic presented in this section is responsible for finding a good clustering configuration in a network with low mobility. Wherever big changes occur in the network topology, the heuristic should be called again in order to re-define the network partition by re-electing clusterheads.

5.6.1 Clusterhead Selection

The main difference between this and the next heuristic is that here only the clusterheads are elected using a response function from ordinary node to clusterhead. In the initial state, all nodes of the network are ordinary nodes, i.e., there is no cluster structure in the network. The variable $state_v$ describes the actual state of a node v ($state_v \in \{CH, Me, Nm\}$) and c_i is the set of current members of cluster $i \in \mathbb{N}$. For simplification we define that the $clusterID = i$. Initially, for $i = 0, 1, .., n$, $c_i = \emptyset$. The response function of Equation 5.4 is responsible for the transition of a node $v \in V$ from ordinary (Nm) to clusterhead.

$$T_{\theta_{CH_v}}(s_{CH_v}) = \frac{s_{CH_v}^{\beta}}{s_{CH_v}^{\beta} + \theta_{CH_v}^{\beta}} \cdot \rho \tag{5.4}$$

Where θ_{CH_v} is the threshold of the node v to become clusterhead and s_{CH_v} is the stimulus of v to assume the clusterhead role. The parameter $\rho \in (0, 1]$ is used to control the speed at which the clusterhead selection happens. It will be explained later on, for now it can be ignored.

The threshold specifies how appropriate a node is to the role, a small θ_{CH_v} means that the node v is very suitable to be clusterhead. The definition of the threshold can be seen in Equation 5.5.

$$\theta_{CH_v} = k_1 \left(\frac{\sum_{u \in Ngb_{Nm}(v)} w(u,v)}{|Ngb_{Nm}(v)|} \right) + k_2(1 - E_v) + k_3 \left(1 - min\left(1, \frac{|Ngb_{Nm}(v)|}{Max\_Neighb}\right)\right) \tag{5.5}$$

Where $E_v \in (0, 1)$ describe the energy level of node v, where 1 means battery full and 0 depleted. Let $Ngb(v)$ be the set of nodes that are directly connected with v, i.e. $u \in Ngb(v)$ iff $\{u, v\} \in E$. A node u is in the set $Ngb_{Nm}(v)$ iff $u \in Ngb(v)$ and $state_u = Nm$. This means that $Ngb_{Nm}(v)$ is the set of neighbors of v that currently do not belong to any cluster.

The idea of this threshold function is that nodes with high energy level which are very connected (vertices with high degree) are good candidates to be clusterheads (having a small threshold). The energy is an important factor because clusterheads perform administrative (among other) tasks within the cluster and have a special status in the network. Good connectivity comes from the greedy assumption that starting a cluster from well connected nodes will result in a relative small clustering cost (given by eq. 5.1).

The stimulus function is given by:

$$s_{CH_v} = k_1 \frac{t_{elapsed}}{t_{required}} + k_2 \left(1 - \frac{|Ngb_{Me}(v)| + |Ngb_{CH}(v)|}{|Ngb(v)|} \right) \tag{5.6}$$

Where $t_{elapsed}$ is the elapsed time since the clustering heuristic has started and $t_{required}$ is the maximum running time of the algorithm. A node u is in the set $Ngb_{Me}(v)$ iff $u \in Ngb(v)$ and $state_u = Me$. Similarly, $u \in Ngb_{CH}(v)$ iff $u \in Ngb(v)$ and $state_u = CH$. With simple words, $Ngb_{Me}(v)$ is the set of neighbors of v that are members of some cluster and $Ngb_{CH}(v)$ is the set of neighboring nodes that are already clusterheads.

The underlying idea is that nodes that for a long time did not belong to any cluster and nodes without clusters in the vicinity should have a higher stimulus to become clusterheads.

With the transition function given by eq. 5.4, some nodes will spontaneously start to change the role to clusterhead based on the stimulus function. When a node decides to be clusterhead, it selects a random $ClusterID$.

Here we analyze briefly how the behavior of the nodes with different probabilities coming from the response function evolves. As already stated, the clusterhead test is executed in a periodic way. Let's $p_1, p_2, ..., p_n$ be the probability returned by the response function from nodes 1 to n ($p_1 = T_{\theta_{CH_1}}(s_{CH_1})$). We can model this behavior with a geometric distribution. This means, we calculate the probability distribution of the number X of *Bernoulli trials* needed to change the state from nonmember to clusterhead, supported on the set $\{1, 2, 3, ...\}$ (the trials). For example, for p_1, the probability of becoming clusterhead after k trials (testing rounds) is:

$$P(X = k) = (1 - p_1)^{(k-1)} p_1$$

In Figure 5.16, the cumulative distribution for different probabilities is shown. As expected, with high clusterhead response-function probabilities, a small number of rounds is enough to virtually ensure the clusterhead role assumption. Nodes with a higher threshold and/or smaller stimulus will have smaller probabilities, and this requires (with high probability) more time to decide to become clusterhead. If a more suitable node is in the neighborhood, it will (with a high probability) become clusterhead first and capture the node as member.

Another fact that can be derived from the cumulative distribution is that nodes with a very small role changing probability returned by the response function perhaps need a rather large number of rounds to become clusterhead. For very sparse areas of the network, this could be a normal situation. In order to accelerate that, we add the elapsed time in the stimulus function. Therefore, the probabilities are increasing with the time, which reduces the election time in sparse areas of the network.

A problem that arises from dense areas of the network is the fact that several nodes in some neighborhood can decide in very short time to be clusterhead, before a neighboring clusterhead has completed its cluster.

To understand the problem better, we define here the least potential area of influence and area of interference (extension of the sphere of influence/interference definitions made in the work [77]).

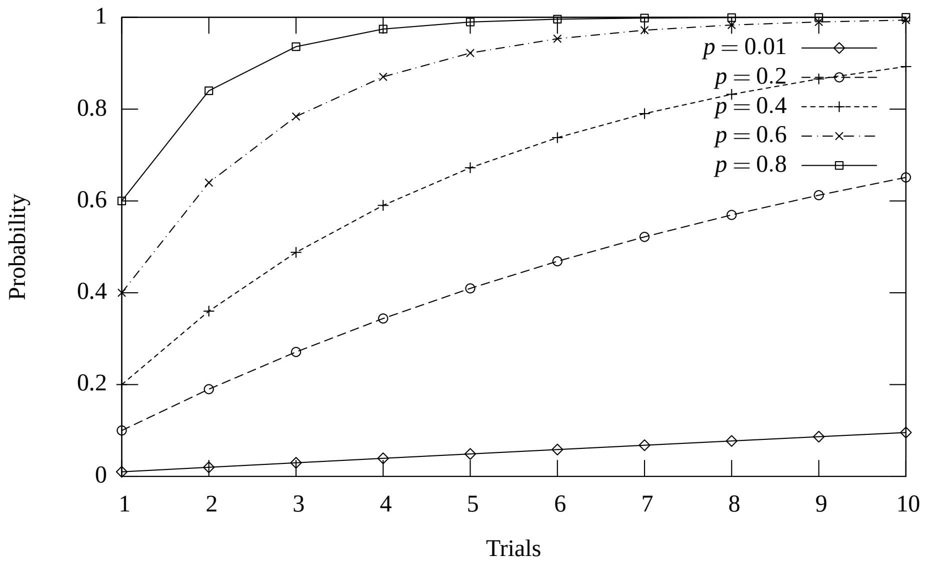


Figure 5.16: Cumulative of the geometric distribution for given response function probabilities

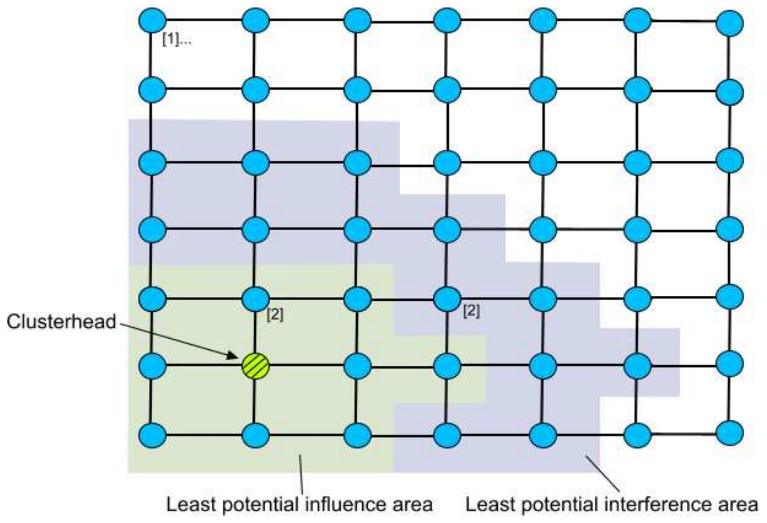

[1]...
[2]
[2]
Clusterhead
Least potential influence area
Least potential interference area

Figure 5.17: Least potential influence and interference areas for a grid network. $[x]$ near the node means that the node has x units of resource.

Definition 5.6.1. *A node u is in the least potential area of influence of clusterhead v ($u \in \Psi_v$), if a cluster constructed by v can contain u for a cluster with at least q resources and where no extra node is taken after reaching (or overpassing) the limit q.*

In Figure 5.17, an example of the least potential area of influence is shown.

Definition 5.6.2. *A node u is in the least potential area of interference of the clusterhead v ($j \in \Phi_v$), if there is a node j in the network whithin the least potential area of influence of both u and v.*

An example of a least potential area of interference is also shown in Figure 5.17. The least potential area of influence is the (minimal) search space from members of the cluster whose clusterhead is v. Two clusterheads inside the area of interference gathering members at the same time can potentially compete for the same node. If both are inside the area of influence, they can even block themselves to attract potential good members for the cluster.

To reduce the probability of occurrence of this situation, we introduce the parameter $\rho \in (0,1]$, which controls how fast the emergence of clusterheads occurs (see Equation 5.4). A smaller ρ means that the nodes will take more trials (rounds) to have a higher probability of becoming clusterhead. In general, bigger ρ values bring a faster network decomposition (complete network clustering time), whereas smaller ρ values bring a slower, but better quality (with less collisions) network decomposition. The parameter also depends on the size of the area of influence ($|\Phi|$). As $|\Phi|$ is correlated with the minimum amount of resources per cluster (q), with higher q, ρ should be smaller in order to avoid collisions. Moreover, the worst case clustering time is also an important factor influencing an optimal ρ. A longer cluster construction time brings higher probability of collisions because nodes in the neighborhood have a higher chance to select themselves to be clusterhead in parallel [2]. The topology of the network plays also an important role since dense (very connected) networks bring larger areas of influence.

We present here a short analysis of the worst case probability of a collision when a node v has fired (and becomes clusterhead). Our analysis is very pessimistic, and in practical applications a much less strict ρ can be used. Let t_c be the worst case cluster construction time (defined later on). Let τ be the time between two consecutive tests (trials) for clusterhead selection. We define r_c to be the number of trials happening during the cluster construction time ($r_c = \lceil \frac{t_c}{\tau} \rceil$). In a very adverse scenario, we suppose that the threshold to become clusterhead for all nodes is approaching zero, which means that the response function with a very small stimulus is already returning ρ because $\frac{s^{\beta}_{CH_v}}{s^{\beta}_{CH_v}+\theta^{\beta}_{CH_v}} = 1$, i.e., $T_{\theta_{CH_v}}(s_{CH_v}) = \rho$.

We will calculate the probability that no node fires during the period where collisions may happen. This period can be seen in Figure 5.18 and it is two equal to times t_c, because when v decides to become clusterhead, a potential inferencing node could already exist (started before v) or can start after v has initiated. We will call the number of clusterhead trials happening during this period r_{interf} which is defined as $r_{interf} = \lceil \frac{2 \cdot t_c}{\tau} \rceil$.

As the geometric distribution is memoryless, we can state the probability of no other node in the area of influence disturbing the cluster construction by clusterhead v by:

$$P\big(k > (r_{interf} \cdot |\Phi|)\big) = 1 - \sum_{k=1}^{r_{interf} \cdot |\Phi|} (1-\rho)^{(k-1)}\rho \tag{5.7}$$

[2]A node test whether it should become clusterhead until receiving a call for members request from an actual member of the cluster. If, at the end of the process, the node was not included in the cluster, it starts again to make the clusterhead test.

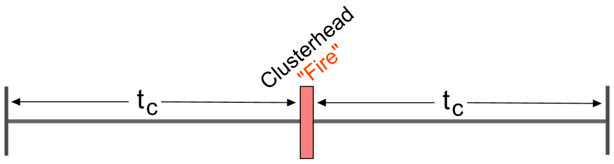


Figure 5.18: Disturbance prone period of time.

Now, given a cluster construction time (t_c), the time between two consecutive clusterhead tests (τ), and the influence area of the node, we can calculate the probability of collisions to a given ρ . We can also invert the equation and select a value of ρ appropriate for the desired collision probability.

In a more realistic environment, for most nodes, the threshold to become clusterhead will not be near zero for the majority of nodes, therefore a much higher ρ than the one calculated above could be used with only a unsubstantial penalty in the quality of the clustering.

5.6.2 Member Selection

5.6.2.1 Influencing Parameters

When a node decides to become clusterhead, it must select the appropriate members of the cluster. The following parameters influence the suitability of a node b to became member of the cluster:

1. The distance to the closest node already in the cluster. Nodes with a small distance to the cluster will bring a smaller cost than nodes with bigger distances. The distance of the node b to the cluster i is defined by: $D_i^b = min\{w(b,e)|e \in Ngb(b) \cap c_i\}$, i.e., the smallest vertex weight that is adjacent to node b and to a member of the cluster i. If a node d is not directly connected to a node which is already a cluster member, $D_i^d = \infty$.

2. The distance to the clusterhead. This parameter is responsible to shape the cluster, constraining its diameter. For clusters with the same amount of nodes but different forms, large diameters means higher cost (an example can be seen in Figure 5.19).

3. Connectivity to nonmembers. This parameter is important when a lot of resources are still missing in the cluster, i.e., the clustering process is in its initial phase. This is due to the fact that nodes with a good connectivity probably will have good candidates for the next membership selection. The connectivity of node $b \in V$ to nonmembers is given by $Cn_{Nm}^b = \sum_{e \in Ngb_{Nm}(b)}(1 - w(b,e))$, i.e., the sum of the "proximity" $(1 - w(b,e))$ of the set of the neighbors of b that have nonmember status. To understand the effect of this term, Figure 5.20 shows an example. Both nodes have the same distance to the cluster. However, node 1 has a very near nonmember node, and node 2 has a much more distant nonmember neighbor. In order to assure future good candidates, node 1 should be preferred for membership. Nevertheless, if the clustering process is in the end phase, we do not need additional members, which means that this parameter is not important in this case.

4. Connectivity to members of the cluster. Selecting nodes highly connected to other nodes that are already members of the cluster increases the probability of reducing the total cost of the cluster because there is a chance that the connections contribute to reduce the size of the paths through the nodes. The connectivity of node b to members of cluster i is given by $Cn_i^b = \sum_{e \in \{Ngb(b) \cap c_i\}}(1 - w(b,e))$, where c_i is the current set of members of the cluster $i \in \mathbb{N}$.

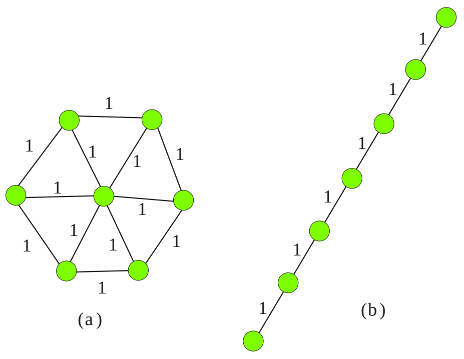


Figure 5.19: Diameter versus cluster cost in a cluster with 7 nodes. (a) Diameter is 2 and the cost is 63. (b) Diameter is 6 and the cost is 112

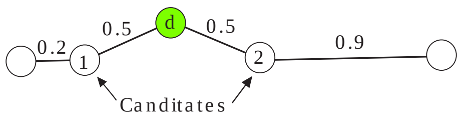


Figure 5.20: Example of two candidates with different neighborhood.

5. The resource availability of the node. As a general rule, nodes with higher resource availability will potentially reduce the cost of the cluster because they reduce the necessity of taking additional nodes. Nevertheless, to include nodes with plenty resources at the ending phase of the membership selection with high resource (perhaps more than the cluster needs) could increase the total cost due to the fact that the node may be better employed by another cluster.

These aspects will be explicitly or implicitly observed by the *Membership-Select* algorithm presented here.

5.6.2.2 *Membership-Select* Algorithm

For the membership selection, we have again a $state_v$ variable describing the actual state of a node v. Differently from the clusterhead selection, we have here an additional state: the deciding (Dd) state ($state_v \in \{CH, Me, Nm, Dd\}$).

Let Δq be the amount of resources still needed by the cluster in order to fulfill the requirement q.

The *Membership-Select* algorithm is an incremental process, i.e., at beginning the cluster has just the clusterhead (CH) node. During the clustering process, more and more nodes are added to the existing cluster until the cluster achieves an appropriate size ($\sum_{v \in c_i} r(v) \geq q$).

At the beginning of the clustering process of cluster i, just one node belongs to the cluster: the clusterhead, we will call it of node h_i ($h_i \in c_i, state_h = CH$).

When a node becomes part of the cluster (including the clusterhead), immediately a message is broadcasted to the neighboring nodes signalizing the new status and requesting for new members (`Call_Members` message). Each nonmember and deciding node d ($status_d \in \{Nm, Dd\}$) that receives this message changes its state to deciding $(status_d = Dd)$.

Deciding nodes are the potential new members of the cluster. Nevertheless, not all nodes are the best choice to be included in the cluster. In order to privilege nodes potentially contributing to a low global cluster cost (see eq. 5.1), each node b in the decision state estimates its own fitness value $0 \leq Fitness_i(b) \leq 1$. This value will be defined later. $Fitness_i(b)$ represents how suitable is the inclusion of node $b \in V$ in the cluster i.

At this point, the node b waits a delay whose duration is proportional to the $1 - Fitness_i(b)$ value. When the waiting time is elapsed, the node sends a `Membership_Request` message to the clusterhead, informing that it is willing to be included in the cluster. Now, the clusterhead, based on the Δq and the availability of resources of the candidate, can decide whether the node will be accepted as member. If accepted, the clusterhead includes the new node in a table with all members of the cluster. A message is sent back to the node confirming/refusing the entrance in the cluster. When receiving the response message, the requester changes its status accordingly ($state_b = Me$, if accepted, and $state_b = Nm$, if refused). If accepted, this new status is broadcasted immediately in a message calling for new members (`Call_Members`) to the neighborhood of b, starting the process again.

As already said, the decision of accepting/rejecting new members is done by the clusterhead node based on Δq and $r(b)$. If $r(b) \leq \Delta q$, the node b is promptly accepted by the clusterhead. When $r(b) > \Delta q$, before answering to a request, the clusterhead waits for an additional period in order to provide a chance to nodes with a not so good fitness request for membership. Let $R \subset V$ be the set of nodes requesting for membership during this period. The clusterhead will select the node $l \in R$, where $r(l) \geq \Delta q$ and for $\forall v \in R, ((r(v) - \Delta q) \geq (r(l) - \Delta q)) \vee r(v) < \Delta q$, i.e., the node that best fits to cover the requested resources. This node will receive the accept message while the others of R will receive a reject message.

When $\Delta q \leqslant 0$, i.e., the cluster is complete, all additional receiving requests will be rejected.

Before introducing additional aspects of the *Membership-Select* algorithm, a small example of the process described is presented. Consider the example depicted in Figure 5.21. We colored the nodes according to the $state$; white nodes are nonmembers, black nodes are members (or clusterhead, i.e. $status \in \{Me, CH\}$) of the cluster i being formed, and gray members are deciding nodes.

In Figure 5.21a, the initial condition is shown. The cluster has just one member: the clusterhead (node 1), selected by the transition function presented in Section 5.6.1. The clusterhead makes a broadcast of the `Call_Members` message transmitting its state (5.21b). At this point, all nodes that receive the message change to the deciding state. A timer is set based on the calculated fitness for each node. In Figure 5.21c, the programmed time of node 2 is already elapsed. The node now should ask for membership. A message is sent to the clusterhead asking to be included in the cluster. As the total resource request (q) is not satisfied by the current cluster size, the node 2 is included in the cluster. Now it also broadcasts a `Call_Members` message to the neighborhood (Figure 5.21d). When nodes 4 and 5 receive the broadcasted message, they start a timer that is related to the computed fitness ($1 - Fitness_i(4)$) and $1 - Fitness_i(5)$). Due to the fact that node 4 has already a timer, just the timer with the shortest deadline is kept. In Figure 5.21e, the programmed time of node 4 is elapsed. Similar to node 2, it requests for the permission to enter in the cluster (and it is included). Because the cluster is already complete, the node 4 does not broadcast a new `Call_Members` message.

Finally, the waiting time for nodes 5 and 3 has ended. They request to become members of the cluster to the clusterhead, but due to the fact that the cluster has enough resources, the permission to integrate the cluster is refused.

Now we will integrate the already presented heuristic hints (see Section 5.6.2.1) that should guide the member selection. The first point says that the heuristic should privilege nodes with a small distance to some of the nodes inside the actual cluster. In order to observe that, two aspects must be addressed:

1. Include the distance to the next cluster member in the fitness function. As it will be presented later in this section, an important parameter of the fitness function is the distance of a candidate

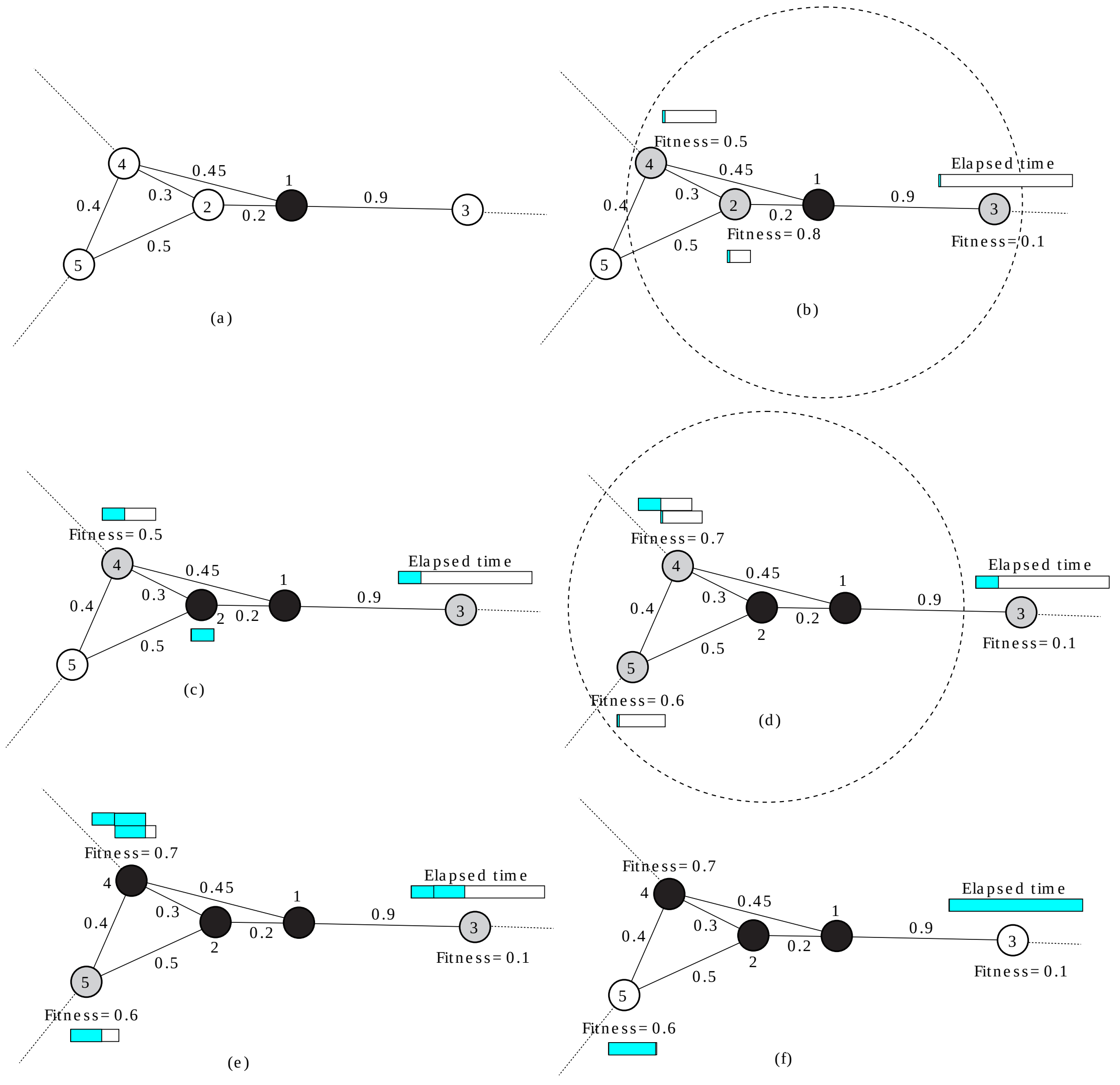


Figure 5.21: Example of member selection in a partial network. All nodes have unitary amount of resource ($r(b) = 1, b \in V$) and $q = 3$.

to the next member in the cluster. The distance is measured by our combined metric ("virtual distance"). Nodes with good connection to the already existing cluster have better fitness than nodes with just bad links to the cluster.

2. Using the example showed in Figure 5.21, we can highlight an implicit behavior of the heuristic that does not befit this metric. The nodes that are near the clusterhead started the timer very early during the members selection; nodes far away have the timer started later. This means that nodes far of the clusterhead, but at the same time with good connection to the cluster, are penalized in favor of nodes that perhaps do not have a good connection to the cluster, but are near to the clusterhead.

This aspect should be addressed together with the point number two in our influence parameters list: the distance to the clusterhead. This point is aided by the implicit behavior of algorithm. The two aspects are important to reduce the cluster cost. Nodes near to the cluster are suitable because the connection cost is smaller, nevertheless, to keep clusters with smaller diameter also helps to reduce the total cost.

The distance to the clusterhead is also addressed by two points:

1. Including the distance to the clusterhead in the fitness function.

2. Implicit behavior of the heuristic. To show that, we again use the example showed in Figure 5.21. The fact that nodes near to the clusterhead started the timer earlier implicitly helps to get small diameter clusters.

Analyzing this two different requirements, the following method was created in order to penalize the distance to the clusterhead and reward the distance to the cluster. We will now count the rounds that the algorithm has already executed. Using the example presented in Figure 5.21, (b) represents the first round of the algorithm and (d) the second. Each time that a new member was selected and makes a broadcast to the neighborhood, the variable $round_v, v \in V$ is incremented.

Now, instead of just using the fitness to calculate the waiting time, the round also plays an important role. We define the waiting time of a node v to request to be included in the cluster i as:

$$WaitingTime_i^v = k \cdot (1 - Fitness_i(v)) \cdot \frac{1}{\kappa\ round_v + (1-\kappa)}$$

Where $v \in V, \kappa \in [0,1], k \in \mathbb{R}+$ and $0 \leq Fitness_i(v)) \leq 1$.

Using this equation, for bigger rounds the time that should be waited is shortened. With the κ parameter, the amount of reward given to the distance to the cluster versus penalization of distance to clusterhead can be controlled.

Now we can present the Fitness function that takes into account all points presented in Section 5.6.2.1 (for $\sum_{j=0}^{5} k_j = 1$).

$$Fitness_i(v) = \tag{5.8}$$
$$= \begin{cases} k_1 \cdot (1 - D_i^v) + k_2 \cdot (1 - \ min\{\frac{D(v,Clusterhead_i)}{Max\_dist}, 1\}) + k_3 \cdot \min\{\frac{Cn_{Nm}^v}{Max\_connect}, 1\} + & \text{if} \quad r(v) < q \\ \quad + k_4 \cdot \min\{\frac{Cn_i^b}{Max\_connect}, 1\} + k_5 \frac{r(v)}{q}) & \\ 0 & \text{if} \quad r(v) \geqslant q \end{cases}$$

Where $k_1,..,k_5 \in [0,1]$ define how each of the terms influence the fitness metric. It is important to remark that $0 \leq Fitness_i(v) \leq 1$. For two nodes $v,u \in V$ and $Fitness_i(v) < Fitness_i(u)$ means that the node v is less suitable for the cluster i than node u.

Max_dist gives the minimal distance to the clusterhead that should be considered, and the maximum penalty *Max_connect* is the same for the connection measurements. An important remark is that for nodes with more resources than required, the fitness is always 0 because they should form a cluster with one member itself.

Now we will calculate the worst case construction time of a cluster (t_c). t_c parameter is used in the analysis of possible collisions during the clusterhead selection (Section 5.6.1). The worst case t_c happens when the members are selected in a line. For that, supposing that the members have the minimum resource L_{inf}, $\left\lceil \frac{q-L_{inf}}{L_{inf}} \right\rceil$ rounds are necessary to complete the cluster. We define t_c as:

$$t_c = \sum_{r=1}^{\left\lceil \frac{q-L_{inf}}{L_{inf}} \right\rceil} k \cdot \frac{1}{\kappa \cdot r + (1-\kappa)}$$

The expression is derived from the worst case *WaitingTime*, where the fitness of all nodes included in the cluster is equal to zero. Here the time of message transmissions are not taken in account. Since they are much smaller than the *WaitingTime*, they can be disconsidered in the t_c approximation.

5.6.3 Message Relay to Clusterhead

This is a complement of the presented algorithm. The idea is to create a spanning tree with the clusterhead as root, helping to select good paths when members communicate with the clusterhead.

Inside a cluster, good routes to the clusterhead are important because:

- The `Membership_Request` is sent by all nodes to the clusterhead
- Often, clusterheads perform other administrative tasks or even can be used to form a backbone for message relay (topology control)

For creating the spanning tree, each node stores the actual known virtual distance to the clusterhead (*D_CH*) and the corresponding link that is used to achieve it (*RelayNode_CH*). Every time a `Call_Members` message is transmitted, the current distance to the clusterhead is also transmitted. Then, every receiver d of the message sent by v can identify whether the current path is the shortest one. If not, the new route is assumed.

A further point that should discussed here is the inter-cluster communication. Nodes in the border of the cluster, i.e., that have one or more links to nodes belonging to other clusters are called gateways. Such nodes inform the clusterhead about which clusters they can arrive. The clusterhead may select, for each neighboring cluster, one of these gateways (a good connected one) to relay messages to the neighboring cluster. The spanning tree can be used to route the message from the clusterhead to selected the gateway (other routing algorithm may be also used).

5.6.4 Enforce Phase

Using the described algorithm, at the end of the clustering time, it may happen that some clusters could not find enough nonmembers to include in the cluster. For example, between several formed clusters, some nodes that haven’t been required by those clusters may remain. Together, they do not have enough resource to form a autonomous complete cluster.

Such incomplete clusters are disrupted by the clusterhead and the nodes enter in the nearest complete cluster using enforce membership requests. The enforce phase guarantees that all nodes will be included in some cluster in a finite time.

As a result of the clustering algorithm presented in this section, complete clusters have been built. In each cluster, a spanning tree with the clusterhead as root has been constructed.

5.7 Clustering Dynamic Ad hoc Networks

In dynamic ad hoc networks, the link quality and the network configuration change over time. New links are created, links are destroyed, new nodes appear, and nodes can fail. In this section, a heuristic that tries to find good clustering solutions and, at the same time, aiming at maintaining these clusters over the time, is presented.

5.7.1 General View

Like the first clustering algorithm presented, the clusterheads in this heuristic are elected using a threshold response function (see Section 5.6.1). Nevertheless, the other possible roles of the nodes are also selected using threshold response functions. As in the previous case, the variable $state_v$ describes the actual type (role) of a node v ($state_v \in \{Ch, Me, Nm\}$) and c_i is set of the current members of the cluster $i \in \mathbb{N}$, and, for simplification, we define the $clusterID = i$. Initially, for $i = 1, .., n$, we have $c_i = \emptyset$. In the initial state, all nodes of the network are ordinary nodes, i.e., there is no cluster structure in the network.

In the dynamic approach, we have the following response functions driving the changing of roles of the nodes in the system:

Clusterhead related functions:

Nonmember → Clusterhead: The response threshold function is called $T_{\theta_{ch_v}}$, $v \in V$. It returns the probability of a nonmember v to become clusterhead. Similar to the first clustering algorithm presented, this response function is responsible to model the emergence of clusterheads in areas of the ad hoc network where no clustering is already taking place.

The exchange from the clusterhead status to nonmember (this means the end of existence of a cluster) is controlled by a deterministic mechanism described later. Even so the exchange of the clusterheads inside a cluster (clusterhead rotation) when the energy reserve of the current clusterhead gets low.

Membership related functions:

Nonmember, Member of x → Member of y: This function (recruitment function, $T_{\theta_{recr_{v,i}}}$) models the entrance of a node (here called v) into the cluster with $ID = i$. The node v may be nonmember or a member of another cluster. When a cluster does not have the necessary resources to cover the required q, it “attracts” new nodes to join it. The response function to join a cluster ($T_{\theta_{recr_{v,i}}}$) gives the probability of node v to enter in the cluster i. It is possible for a cluster to “steal” nodes from a neighboring cluster. The threshold for this action (steal of nodes) is reduced when a node, for example, is changing the physical position to some place nearby the new cluster (therefore, leaving the cluster from which it was previously member).

Member → nonmember: This response function ($T_{\theta_{leave_v}}$) models the situation when a node abandons the cluster due to the fact that its connection to the cluster is very weak.

Before describing the response functions in detail, the general idea of the algorithm is described here. Nonmembers are "clusters" with no resource ($R_i = 0$). As the previous "quasi-static" version, the first task of the heuristic is to elect the clusterheads of the network. This is made using the response threshold function $T_{\theta_{ch_v}}$. A clusterhead is now an unitary cluster with some resource ($R_i = r(v)$, v is the clusterhead of cluster i). When a clusterhead is elected in some part of the network, it starts as consequence of the existing resource to attract new members. Nevertheless, instead of using the concept presented in the previous heuristic, here the response function ($T_{\theta_{recr_{v,i}}}$) is used to here recruit new cluster members through a positive feedback process.

In the case of the equation $T_{\theta_{recr_{v,i}}}$(attraction of new members to the cluster), the response threshold and stimulus now here the following meaning:

Threshold $\theta_{recr_{v,i}}$: Measures how connected is the node v to the cluster i.

Stimulus $s_{recr_{v,i}}$: Represents the volition of a cluster to attract new members. Here the positive feedback acts.

The idea is that a cluster incrementally grows until it achieves at least the requirement q. A growing cluster exercises an attraction "force" to the nodes that are in the vicinity. This attraction force is expressed by a higher stimulus s in the $T_{\theta_{recr_{v,i}}}$response function. This concept is presented in Figure 5.22.

As shown in Figure, an existing cluster exercise an attraction "force" to the neighboring nodes, independent of its type. For nodes that are already members of the cluster, the same force is responsible for the cohesion of a cluster. This is because the response function $T_{\theta_{leave_v}}$is the inverse of $T_{\theta_{recr_{v,i}}}$($T_{\theta_{leave_v}} = k_1 \cdot (1 - T_{\theta_{recr_{v,i}}})$).

Returning to the attraction of the nodes, the intensity of the force is expressed in the stimulus of the response function $T_{\theta_{recr_{v,i}}}$. The intensity of the force (and consequently the stimulus to enter into the cluster) is regulated by the amount of resources already in the cluster. We use here a positive/negative feedback mechanism typical for self-organizing systems.

5.7.2 Clusterhead Management

In this section we will describe how the clusterheads are elected and withdrawn in our cluster construction heuristic.

5.7.2.1 Clusterhead Election

As in the previous heuristic, the clusterheads are elected using a stimulus-response function. The function is the same as used in the "quasi" static heuristic (see eq. 5.4). The definition of the stimulus is also the same as used in the previous heuristic (eq. 5.6), i.e., the idea is that nodes which for a long time are not belonging to any cluster and nodes without clusters in the vicinity should have a higher stimulus to become clusterhead.

The threshold for assuming the clusterhead role is similar to the previously presented (eq. 5.5) with modification. The modification is the addition of a factor in the formula to take in consideration the stability of the node as part of the threshold, i.e., nodes with stable neighborhood are preferred to be clusterheads instead of nodes with constantly changing connections.

The modified version of the equation is given by Equation 5.9, for $\sum_{j=1}^{3} k_j = 1$.

$$\theta_{CH_v} = k_1 \cdot \left(1 - \frac{\sum_{u \in Ngb_{Nm}(v)} w(u,v)}{|Ngb_{Nm}(v)|}\right) + k_2 \cdot (1 - E_v) + k_3 \cdot \overline{\Delta W_v} \tag{5.9}$$

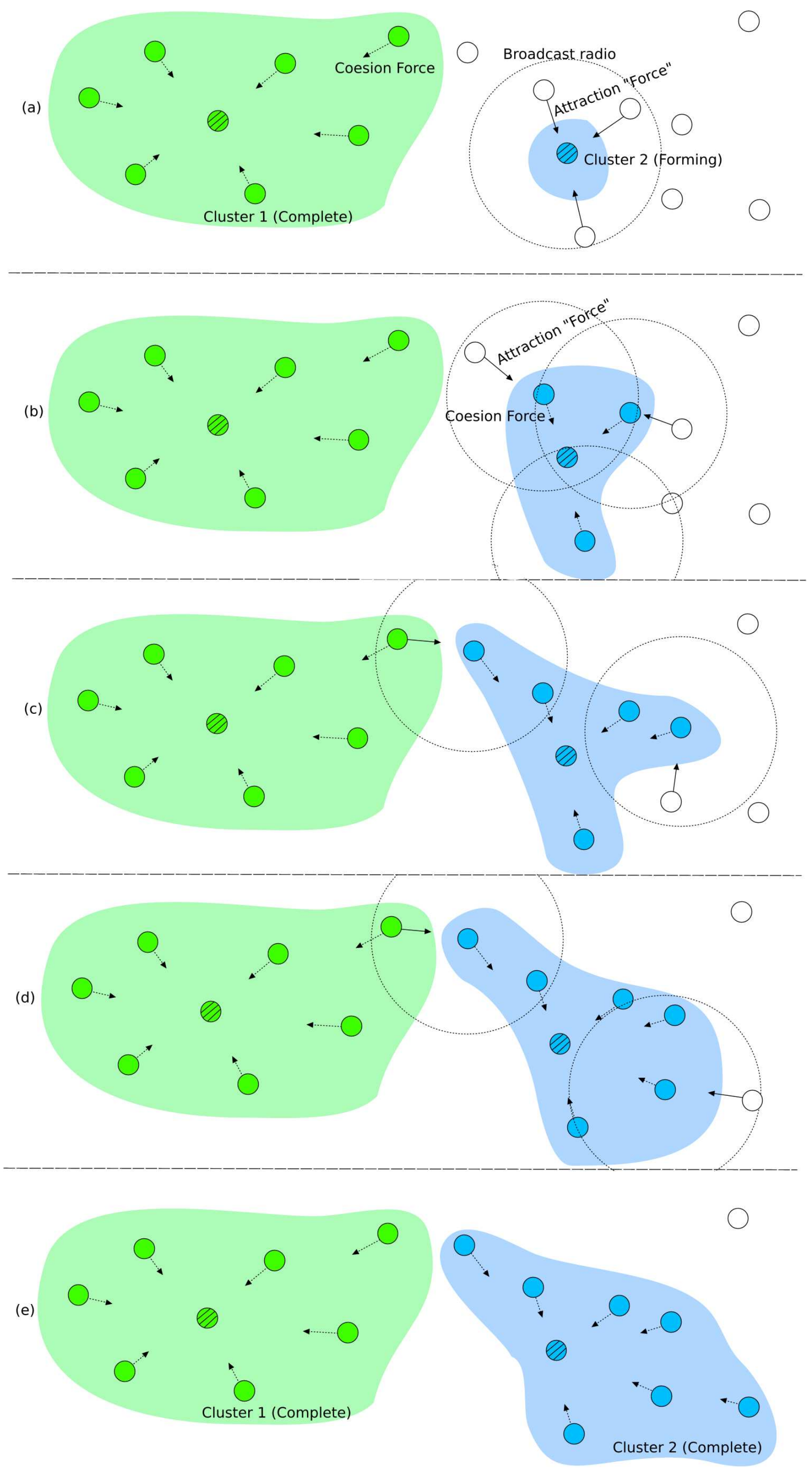


Figure 5.22: Example of execution of the heuristic for $q = 8$ and $\forall v \in V, r(v) = 1$.

As already described, $E_v \in (0,1)$ is the energy level of the node v, $Ngb(v)$ is the set of nodes that are directly connected with v, and $Ngb_{Nm}(v)$ is the set of neighbors of v that until now do not belong to any cluster.

The third term ($\overline{\Delta W_v}$) measures the stability of the node, i.e., how fast the neighboring of the node is changing. For that, we measure the variation of the link weight of the neighborhood over time. The function $\overline{\Delta w_t}(v,j)$ returns the mean of the variation of the link between $\{v,j\}$ until the time t. The definition can be seen in eq. 5.10. $Ngb_t(v)$ returns the neighbors of the node v on the time t, and $w_t(v,u)$ returns the link metric of the link $\{v,u\}$ in the instant of time t.

$$\overline{\Delta w_t}(v,j) = \frac{1}{2}\left(\overline{\Delta w_t}(v,j) + |w_t(v,j) - w_{t-1}(v,j)|\right) \tag{5.10}$$

For this function (Equation 5.10), we use $w_t(u,v) = 1$ for disconnected nodes (i.e. $\{u,v\} \notin E$) on the time t, (differently from the global definition). The reason is to keep the value of W_v inside the $[0,1]$ interval.

The initialization is done by $\overline{\Delta w_{t=0}}(v,j) = 0$.

Equation 5.11 gives the mean of all the variation of the links of v's recent neighborhood. t is in this case the current time ($t = now$).

$$\overline{\Delta W_v} = \frac{1}{|Ngb(v) \cup Ngb_{t-1}(v)|} \sum_{j \in (Ngb(v) \cup Ngb_{t-1}(v))} \overline{\Delta w_t}(v,j) \tag{5.11}$$

The idea of this threshold function is similar to the first heuristic, i.e., nodes with high energy level, strongly connected, and additionally with stable connections are good candidates to become clusterhead. The stability factor comes from the assumption that nodes with less movements in the past are good candidates for clusterhead because they have a higher probability of staying in the cluster, not migrating to remote regions of the network.

Similarly to the first heuristic, the transition function given by eq. 5.9 stimulates the spontaneous change of role from some nodes to clusterhead. When a node decides to became clusterhead, it selects a random *ClusterID*.

5.7.2.2 Clusterhead Withdrawal

This version of the heuristic is required to deal with dynamic networks, where the links are constantly changing. Moreover, other parameters may also change, e.g. the actual battery status of a given node. Therefore, it is necessary to design mechanisms that detect this dynamic and react upon than by means of cluster disruption (removing the clusterhead from the cluster) or clusterhead node exchange.

There are two types of withdrawal of the clusterhead:

Clusterhead Exchange: This happens when the energy level of the clusterhead is considerably below the neighborhood. This means that the clusterhead role has used a big amount of energy of the node due to the extra burden brought by the extra responsibilities of the role. Therefore, when this bypasses a certain threshold, the actual clusterhead selects a neighbor with higher energy level (and connection) to assume its role.

Cluster Disruption: This happens when a clusterhead, after a certain number of attempts, could not keep the requirement q of resources per cluster. In this case, the (incomplete) cluster will cease its existence and the current members are free to join other existing nodes.

Now we will briefly describe the both quoted mechanisms:

Clusterhead Exchange In the cluster construction (and maintenance) round, the neighboring nodes of the clusterhead v that are members of the cluster i ($Ngh(v) \cap c_i$) send their connections and energy rate. The clusterhead keeps this information in one table. For a neighboring node e ($e \in Ngb(v) \cap c_i = Ngb_{ci}(v)$), the connection rate is given by (for $k_1 + k_2 = 1$):

$$CR_i(e) = k_1 \cdot min\left(\frac{|Ngb_i(e)|}{Max\_Neigbor\_Cluster}, 1\right) + k_2 \cdot \left(1 - \frac{\sum_{u \in Ngb_i(e)} w(u,e)}{|Ngb_i(e)|}\right) \tag{5.12}$$

Where $Ngb_i(v)$ is the set of neighbors of v that belong to cluster i.

The energy rate ($E_e \in [0,1]$) is a measure of the current amount of energy of a node, where 0 means energy depleted.

For the decision of clusterhead exchange, the clusterhead calculates the average energy and connection rate among the neighbors belonging to the cluster. The neighboring average connection and energy rates are given by:

$$\overline{NgbCR_i(v)} = \frac{\sum_{e \in Ngb_i(v)} CR_i(e)}{|Ngb_i(v)|} \tag{5.13}$$

$$\overline{NgbER_i(v)} = \frac{\sum_{e \in Ngb_i(v)} E_e}{|Ngb_i(v)|} \tag{5.14}$$

The role of the clusterhead is transferred if the average energy and connection of the neighboring nodes is better by a factor $k \in [0,1]$. This means, if $CR_i(v) + E_v < k \cdot \left(\overline{NgbCR_i(v)} + \overline{NgbER_i(v)}\right)$, the clusterhead withdraws and assigns its role to the node with the higher sum of energy and connection rate using a message called *exchange_clusterhead*.

Cluster Disruption As already mentioned, the disruption is used to extinguish the existence of clusters without the necessary resources and without the possibility at short time to gather those resources. In Section 5.7.4.1, a mechanism responsible to gather new nodes (or release some nodes when the cluster is overcrowded) is presented. A round is executed every time when a clusterhead notices the lack or excess of resources in the cluster. The cluster disruption acts based on the execution of the rounds. When a lack of resource is noticed by the clusterhead (the mechanisms used by the clusterhead for this are discussed later), it starts a series of cluster construction rounds. A round counter ($round_i$) is initialized with zero. Every round within the cluster that achieves the minimum requested resource (q), the counter is again initialized.

The cluster disruption happens whenever the counter achieves a given limit $round_i \geq \zeta$. This means, after several tries, it was not possible to achieve a functional cluster. The message *cluster_disruption* is broadcasted to all current members. Upon receiving the message, they return their state to nonmember (Nm). Now they can be easily attracted by other clusters, or in the future a new clusterhead may emerge.

5.7.3 Member Selection

In order to elucidate further the member selection of our heuristic, we first introduce concepts of self-organization in biological systems.

5.7.3.1 Self-Organizing Systems

Organization is defined in [61] as a structure with function. Therefore a system can be called *organized* if it has certain *structure* and *functionality*. *Structure* means that the components are organized in a certain order. Function means that the system fulfills one purpose.

A system is *self-organized* if it is organized without any external or central dedicated control [61]. The individual entities interact with each other in a distributed peer-to-peer fashion. This interaction is normally local [101].

Self-organizing systems normally are composed by a large number of interacting components. [24] presents two basic modes of interaction among the components: positive and negative feedback.

The positive feedback generally promotes changes in a system. An example of positive feedback coming from the biology is the clustering of several species of birds. For example, herons and blackbirds nest in large colonies because such a behavior provides individuals with certain benefits, such as better detection of predators or facilities in finding food. This nesting is an example of self-organization driven by a positive feedback: birds tend to nest where other birds are already nesting.

The same behavior can be seen in male bluegill sunfish (*Lepomis macrochirus*). They follow the same behavioral rule "I nest where others nest". The nesting pattern appears in a large lake with an initial homogeneous structure due to the amplification of fluctuations: if the density of bluegills is sufficient, through a random process, several nesting sites occasionally will be close enough to provide a sufficiently attraction that stimulates even more bluegills to nest nearby. This random pattern of nest sites now became unstable and a cluster of nest sites will grow. A process, like this, with positive feedback is also called an autocatalytic process.

Now it is important to present the role of the negative feedback. Due to the amplifying nature of positive feedback, a potential destructive explosion may be easily reached. The negative feedback is responsible for controlling and shaping the system in a particular pattern. In the example of the bluegill sunfish, the negative feedback plays a role to avoid overcrowded colonies of fish. The fish have some limits in their behavioral tendency to nest where others nest. The behavioral rule has an autocatalytic as well as an antagonistic component: "I nest where others nest, unless the area is overcrowded". The negative feedback may also come from physical constraints like depletion of the building blocks.

Characteristics
Self-organizing systems present several typical characteristics. The most important are:

Dynamicity: The interactions about components characterizes self-organizing systems. This interaction is dynamic and the production and maintenance of structures is dependent on this interaction among the low-level components.

Emergence: Self-organizing systems usually posses emergent properties. This means that the system acquires qualitatively new properties that cannot be understood as the simple addition of their individual contributions, i.e., system-level properties arise unexpectedly from nonlinear interactions among the system components. An example of spontaneous emergence of patterns is the phenomenon of Bernard convection cells. An initially homogeneous layer of fluid becomes organized into a regular array of hexagonal cells of moving fluid. This can be considered an *attractor* of the system, i.e., under a particular set of initial conditions, and for particular parameter values, the system converges over the time to the attractor state. In the Bernard convection system, when the temperature gradient is low, one attractor is the random motion of fluid. When the gradient is increased (a parameter of the system is changed), a different attractor (the convention cells) appears. This is called a phase change.

Robustness: Self-organizing systems possess a high level of robustness against failures and damage. There is no single point of failure, and the system returns to a stable state after certain disruption [101].

Scalability: The system still works if the number of entities is very large.

An additional important aspect of self-organizing systems is the possibility of controlling the behavior by tunning the system's parameters. The tuning of these parameters can trigger the sudden emergence of novel behavior.

5.7.3.2 Aggregation Through Positive Feedback

As described above, in our clustering construction, positive feedback is used to control the stimulus of neighboring nodes to enter a determined cluster.

Nodes that are of the type *Nonmember* have a total available resource of zero. Once the node v becomes clusterhead of the cluster i, it has the amount of resources $R_i = r(v)$.

The positive feedback is performed by considering the attraction force (or stimulus in the response function) proportional to the amount of resources of the cluster i plus some bias, i.e.:

$$p(R_i) = k_1 + k_2 \cdot R_i \tag{5.15}$$

Equation 5.15 denotes the relation between the amount of resources and the "force" (that is reflected in the stimulus) to attract new nodes to the cluster. k_1 is the initial attraction or bias (even for clusters with minimal resources), and k_2 describes how fast the force increases with the amount of resources.

There is still a problem in this positive feedback: it has an explosive nature, i.e., if there will not be a negative feedback to control it, it tends to catch all nodes of the network in a single cluster. A small fluctuation in the size of a cluster gives it an advantage to catch more nodes from the neighborhood, fact that has as consequence even a larger force to attract more and more nodes. To avoid that, a negative feedback is used.

5.7.3.3 Creating Structure Through Negative Feedback

The negative feedback is responsible for "controling" the explosive nature of the positive feedback and to shape the emergent structures in the self-organizing process. In our case, we use equation 5.16 as negative feedback.

$$g(R_i) = 1 - \left(\frac{R_i}{k_1 \cdot q}\right)^{\beta} \tag{5.16}$$

In Figure 5.23, the positive and negative feedback functions can be seen. It is important to remark that the negative feedback in our case controls how much the positive take effect, i.e., the result stimulus is given by the multiplication of the feedbacks, a fact that is shown in Figure 5.24.

The β exponent controls how fast is the decreasing of the force once the cluster has enough resources, i.e., for $\beta = 1$ the force increases with the amount of resources till the requirement is fulfilled and decreases at the same speed with more resources than necessary. For higher β, the decrease curve is always faster. In Figures 5.23 and 5.24, a β of 5 was used.

Other different functions could be used for the positive and negative feedback.

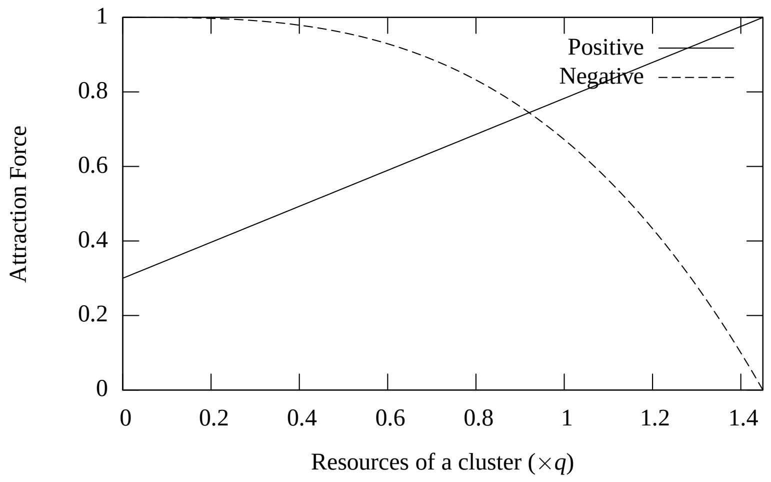


Figure 5.23: Examples of the positive and the negative feedback by the attraction force

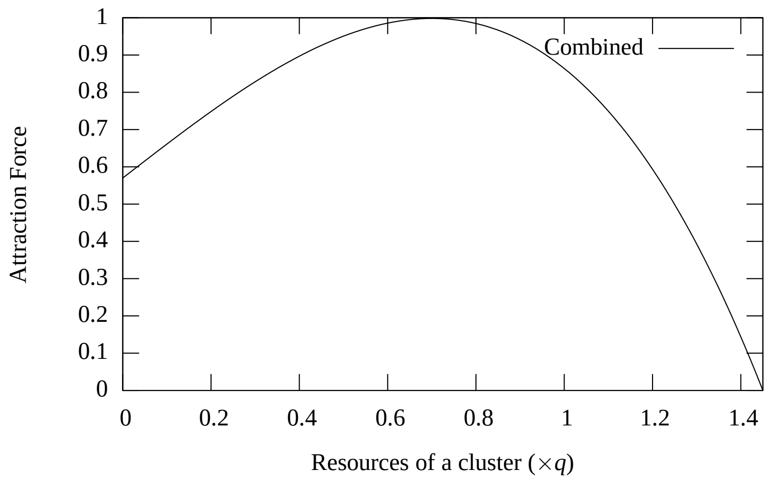


Figure 5.24: Resulting attraction force after combination of the positive and the negative feedback

5.7.3.4 Node Recruitment Response Function

As already presented, the recruitment of nodes is made using the response function $T_{\theta_{recr_{v,i}}}$. The stimulus-response function for the recruitment of new nodes to the cluster i is given by Equation 5.17:

$$T_{\theta_{recr_{v,i}}} = \frac{s_{recr_{v,i}}}{s_{recr_{v,i}} + \theta_{recr_{v,i}}} \tag{5.17}$$

Where the threshold and the stimulus have the following meaning:

Threshold $\theta_{recr_{v,i}}$**:** Measures how connected the node v is to the cluster i. This function is similar to the the fitness function presented in the “quasi” static cluster construction, with some modifications.

Stimulus $s_{recr_{v,i}}$**:** Represents the volition of a cluster to attract new members.

As already said, the threshold $\theta_{recr_{v,i}}$ is similar to the fitness function, but it has an inverse meaning: small values means high suitability of the node to be incorporated to (or continue to be part of) the cluster i. The stimulus-response function reacts to smaller stimulus when the threshold is also small.

The threshold in this function is based on the fitness function (Section 5.6.2.2) with some differences. In that function, the parameters presented in Section 5.6.2.1 are used to calculate the suitability of a certain node to enter a given cluster. In the threshold function here presented, a similar list of parameters is used. In order to avoid the repetition of the list presented in Section 5.6.2.1, we will just elucidate the differences:

1. The third parameter listed in the list (connectivity to nonmembers) is not used in the current threshold function. This parameter was important when a lot of resources were still missing in the cluster, i.e., the clustering process is at initial phase. This is due to the fact that nodes with a good connectivity probably will have good candidates for the next membership selection. Nevertheless, the same response function is used in our heuristic during the forming phase and the maintenance phase (in fact, the two phases use the same principles). Therefore, this item should not be used in the threshold function.

2. The poorest link to the clusterhead when considering all possible paths. This is a new parameter that is necessary in order to deal with the dynamics of the network. It tests whether bad links must be used to reach the clusterhead. The part of the cluster that v belongs to may be almost disconnected if in any possible path from node v to the clusterhead a very bad link must be used to reach it. This happens due to, for example, changes in the positions of the nodes. An example of this situation is depicted in Figure 5.25.

3. An additional parameter to avoid oscillations (entering in a cluster and leaving the same cluster several times) is also included.

The threshold function is defined by Equation 5.18 (where $\sum_{j=1}^{6} k_j = 1$).

$$\begin{aligned}\theta_{recr_{v,i}} = {} & k_1 \cdot D_i^v + k_2 \cdot \ min\left\{\frac{D(v, Clusterhead_i)}{Max\_dist}, 1\right\} + k_3 \cdot \min\left\{\frac{Cn_i^v}{Max\_connect}, 1\right\} + \\ & + k_4 \cdot \frac{r(v)}{q} + k_5 \cdot P_i^v + k_6 \cdot min\left\{\frac{PM_{i,v}}{Max\_past\_memb}, 1\right\}\end{aligned} \tag{5.18}$$

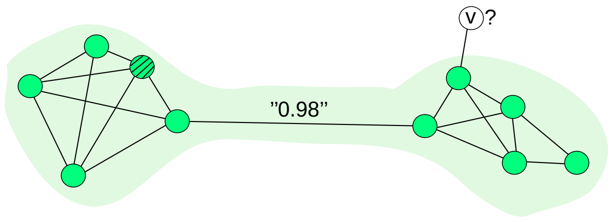


Figure 5.25: Example of cluster that has a part being disconnected. Should node v enter to the disconnecting cluster?

Where D_i^v, Cn_i^v, Max_dist and $Max\_connect$ have been defined in Section 5.6.2.1, and P_i^v is the poorest link from the path with the better links between v and the clusterhead of the cluster i.

$PM_{i,v}$ is a counter of number of past membership of the node v in the cluster i (i.e., $v \in c_i$, where c_i is the set of nodes of cluster i). Each determined time period, this counter is decremented (in order to forget very past events). This term makes that re-inclusion of previous members of the cluster are discouraged in a short range of time, to avoid oscillations.

Returning to the poorest link from the better path between v and the clusterhead. We will now define formally P_i^v.

Let $PoorestLinkMetric(p_h^{(u,v)})$returns the value of the poorest link in the h^{th} path between nodes u and v. This means that:

$$PoorestLinkMetric\left(p_h^{(u,v)}\right) = max\left(w\left(\{u,x_1^h\}\right), w\left(\{x_1^h,x_2^h\}\right), .., w\left(\{x_{g-1}^h,x_g^h\}\right), w\left(\{x_g^h,v\}\right)\right)$$

Then, P_i^v is:

$$P_i^v = min_{h=1,2,..,m}\left(PoorestLinkMetric\left(p_h^{(u_i,v)}\right)\right)$$

Where u_i is the clusterhead of the cluster i and m is the number of possible paths between the node v and the clusterhead.

The stimulus of a node to belong to the cluster i is given by the difference between the force coming from this cluster and a force coming from other neighboring clusters. If there is no neighboring clusters, i is the only force acting upon node v. Equation 5.19 defines the attraction stimulus of node v to cluster i (with $k_1 + k_2 = 1$).

$$s_{recr_{v,i}} = k_1 \cdot (p(R_i) \cdot g(R_i)) - k_2 \cdot max_{j \in \mathbb{N} | (c_j \cap Ngb(v)) \neq \emptyset}\,(p(R_j) \cdot g(R_j)) \tag{5.19}$$

5.7.3.5 Node Membership Withdrawal Response Function

The membership withdrawal can be done in two ways. Either a node has just died, and this will be noticed by the clusterhead in a complete round (presented in the next section), or it decided to leave the cluster using the response function $T_{\theta_{leave_v}}$.

Every time that the cluster round takes place (next section), the membership withdrawal response function is used by the cluster's members to test whether they should leave the current cluster. The main reason for a node to leave its cluster (without being attracted by another one) is topology change where the connection to the cluster gets lost.

The threshold of the response function is defined as:

$$\theta_{leave_{v,i}} = \begin{cases} k_1 \cdot (1 - w(v, parent_v)) & \text{if} \quad Child_v \neq \emptyset \\ k_2 \cdot \left(1 - \theta_{recr_{v,i}}\right) & \text{if} \quad Child_v = \emptyset \end{cases} \tag{5.20}$$

Where $parent_v$ is the current parent of the node v in the forming spanning tree (the spanning tree formation will be explained in the next section) and $Child_v$ is the set of children of the node v in the spanning tree. The parameters are $k_1 \geqslant 1, k_2 \geqslant 1$. This is because it must be rather difficult to a node to exit an existing cluster, in order to avoid constant instabilities. When a node has a child in the spanning tree, the only factor that should be considered in order to evaluate whether the node should leave the cluster is the connection to the parent. This is because other nodes are lying behind this node in the cluster, and if it becomes easily disconnected, the complete subtree can be disconnected from the cluster.

The stimulus to leave the cluster is the conversely from entering into the cluster:

$$s_{leave_v} = (1 - s_{recr_{v,i}}) \tag{5.21}$$

5.7.4 Cluster Construction Process

In this section, the basic skeleton of our heuristic is presented. Here we will just present the cluster construction process. In the next section, the maintenance of already existing clusters is described.

At the beginning, there is no cluster in the network. Every node tests periodically whether it should become clusterhead (using the response function $T_{\theta_{ch_v}}$). Every node also calculates its own threshold and stimulus to become clusterhead. This part of the algorithm is exactly the same as in the "quasi-static" heuristic.

When the node v decides to become clusterhead, a new cluster (we call it cluster i, $i = clusterID$) comes to existence. Initially, this cluster has the resource $R_i = r(v)$.

Now, it starts to broadcast to the neighborhood periodically its actual resource state (R_i). The message is called clusteringForward. The message is composed by the following fields:

clusterID: From which cluster the message belongs to.

clusterResource: Carries the actual resources (R_i) of the cluster.

mesID: An unique message ID.

parent: The parent of the sender in the spanning tree. During the cluster construction process, a spanning tree inside the cluster is formed with the clusterhead as root. It is used in the exchange of messages between the members and the clusterhead.

leaving: A boolean variable indicating whether the sender is leaving the cluster. When true, it forces the children to leave also the cluster.

generationLTime: The logical time of the original generation of the message.

requiredResource: The amount of resources that can be included in the cluster without consulting the root node (clusterhead) of the tree.

acceptedNodes: Nodes that have already been accepted in the cluster by the clusterhead.

completeRequest: This is a boolean field that is used to decide whether the request is a complete request or a partial one. These concepts will be explained in the next paragraphs.

clusterheadDistance: Gives the actual distance to the clusterhead.

As already said, there are two types of requests that can be issued by the clusterhead. The *partial* or the *complete* requests. In the *partial* requests, a message clusteringBackward is generated just in the parts of the cluster where some modification (nodes entering or leaving) is detected, whereas in the *complete* requests, all nodes have to send clusteringBackward messages either generating or forwarding messages coming from other nodes. The act of sending a clusteringForward message, waiting for its (re-)propagation, and receiving the responses (clusteringBackward) is called here cluster construction round. This round has the aim of propagating the actual resource availability of the cluster and, based on this new availability, gather the new and leaving members information to update the membership table.

Before describing in detail the cluster construction round, the format of the clusteringBackward message will be explained.

clusterID: To which cluster the message belongs to.

collectedResources: Carry the total collected resources of this branch of the tree.

originalID: The message ID of the clusteringForward message.

mesID: The ID of the current message.

generationLTime: The logical time of the original generation of the message.

enteringNodes: List of nodes that are entering into the cluster.

leavingNodes: List of members leaving the cluster.

membershipIntention: List of nodes that are willing to be member in the next round.

5.7.4.1 Cluster Construction Round

As already stated, a cluster construction round comprises a propagation of the clusteringForward started by the clusterhead and the return of the answers using the clusteringBackward message.

The whole process starts with the clusterhead sending the clusteringForward message to inform about the actual availability of resources in the clusters.

When hearing this message, a node u stores it temporarily. It waits a small time interval to check whether it will receive the same message coming from another path in the network. The same message can be heard multiple times due to different traveling paths. To check whether the same message have been already received, it uses the mesID field. From the received messages, it selects the one with the smallest clusterheadDistance to process and, if possible, a message where the leaving field is false. The smallest distance is selected in order to reduce the size of the generated spanning tree. If no message with negative leaving field is received, the node will be forced to leave the cluster because the possible parents in the spanning tree are leaving, therefore, it should try to select a parent that stays in the cluster. If after this selection, the same message is heard again, it just ignores it with one exception: if the field parent has the node's id, it stores the message in a table with the children (we call this set $Child_u$). This is because the message is coming from a node that is declaring itself as child of u.

If the message is new, the node checks whether the current logical time is greater than the message time. If yes, it ignores it. After checking these two fields, the currentTime of the node is upgraded by generationLTime. The message ID (mesID) is stored together with the sender of the message (call it node $parent_u$, it is the parent of the node u in the spanning tree). This is done in order to generate a spanning tree with the clusterhead as root.

The way of responding to the incoming message varies depending on the current status of the node u:

Node u is not a member of cluster i: The first action of the node is to determine whether it should enter in the cluster i. There are two ways of being able to enter cluster i. The first is if the node is in the acceptedNodes field of the message. This field is used by the clusterhead to respond to the membershipIntention request sent back to it in the previous round. If the node u is in this list, it has been accepted to be member of the cluster i. The second is to use the response function $T_{\theta_{recr_{v,i}}}$ to evaluate whether the node u wishes to enter into the cluster. If the test using the response function returns positive, the node checks whether it can enter into the cluster without sending a request through the membershipIntention list. The test simply checks whether its resource $R_i(u)$ is less than the requiredResource resource. If positive, it can enter in the cluster. In the negative case, the node ID will be added in the membershipIntention field of the clusteringBackward message. An important final observation is that if the parent is already leaving the cluster (the field leaving is true in the received message), the node will automatically avoid to enter into the cluster.

When a node is accepted to be a new member of the cluster, it changes its clusterID internal variable to the ID of the forming cluster (i). The node is now member of the cluster. The message is then changed and forwarded (broadcasted) further as described in the next section. After repeating the message, the node u listens to the network in order to determine which neighbors are reacting upon the received message. The neighbors repeating the message and with the field parent filled with the ID of u are the children in the forming spanning tree. The node from whom the original request came is the parent in the tree. A timer is set by node u in order to wait for clusteringBackward messages coming from the child neighbors. The field enteringNodes of the clusteringBackward message will be used to inform the clusterhead about the existence of the new member.

Similarly, when a node wants to add in the clusteringBackward message its ID in the membershipIntention field, it also forwards the clusteringForward message and sets a timer to wait for the clusteringBackward messages. When this message is returning to the clusterhead, the intention to be accepted in the cluster will be communicated through the membershipIntention field.

The forward process of the message clusteringForward will be explained in the next section.

If the node is not aiming to become member of the cluster, the action to be taken is related with the type of round: in a *complete* request, a message clusteringBackward is generated by the node and sent back to local parent in the tree. In the case of a *partial* request, the message is just ignored.

Node u is a member of cluster i: The node will test whether it should leave the cluster using the response function $T_{\theta_{leave_v}}$. If the test returns negative, the node just retransmits (forward) the message clusteringForward using the procedure described below. In the opposite case, the node deletes its internal variable clusterID, repeats the clusteringForward message (using the procedure described below), collects its child information in the forming tree, and set a timer to wait for the clusteringBackward message. In this message, the leaving node will inform its state in the leavingNodes field.

Broadcasting the Message clusteringForward As already explained, the nodes that are already in the cluster or entering should repeat the message clusteringForward (in the last case, just if the field requiredResource is greater than a specified threshold). The requiredResource field is updated by $\frac{requiredResource}{|Ngb(u)|}$, where $Ngb(u)$ is the set of neighbors of u . This is to divide the request for resource equality among the neighbors, giving then a chance to enter without asking the clusterhead for permission using the membershipIntentions message.

In addition, the field clusterheadDistance is also updated. In the case of a leaving node, this field is updated to infinity (because there is no more pathway to the clusterhead through this node). Moreover, the field leaving is set to true in this case.

The message is then broadcasted (repeated) to the neighbors.

Returning the clusteringBackward message There are two policies on when a node must respond to the clusteringForward message. Which one should be chosen one depends whether the request was a *complete* or a *partial* one. In the first case, all nodes that belong to the cluster must either generate a clusteringBackward message or forward a message coming from its child. Then the node must insert its ID and resource in the returning message. This is because *complete* rounds are used by the clusterhead to re-check the complete membership of the cluster.

In the second case (*partial* requests), just nodes that posses some information for the clusterhead are forced to generate clusteringBackward messages if a message of this type is not coming from the children to be forwarded.

Before generating the clusteringBackward message by itself, a node has a timer in order to wait for the corresponding message from the children. The node then makes a fusion of its own information with the messages coming from the children nodes and sends to its parent the message forward.

An example of a cluster construction round is shown in Figure 5.26. In (a), a *partial* round has been initiated by the clusterhead (node 1). It sends the clusteringForward message to the nodes in the vicinity. Upon receiving the message, the neighbors use the response function $T_{\theta_{leave_v}}$ to decide whether they should leave the current cluster. In this case, all the neighbors decide to stay in the current cluster and forward the message further updating the requiredResource field properly.

In figure 5.26 (b), nodes 4,5,11, and 14 received the clusteringForward message. Node 4 is already member of the cluster, and its action is similar to the described in the previous paragraph. Nodes 5,11, and 14 are not members of any cluster. Upon receiving the message clusteringForward, the nodes check whether their IDs are in the acceptedNodes list. Node 5 is already in this list, what means that it may enter in the cluster immediately. It then forwards the message further. The nodes 11 and 14 are not in the accetpedNodes. They use then the response function $T_{\theta_{recr_{v,i}}}$ to test whether they should pursue cluster membership. Since this is true in the example, the nodes now test whether they can be members without applying to the clusterhead, i.e., there is enough budget in the clusteringForward message. For nodes 11 and 14, this is the case in the presented example. They now change the internal clusterID to i and forward the message. They also start a timer, looking for the clusteringBackward answering message. This message will be used to inform the clusterhead about the new members and the success of the resource recruitment. The function of the timer is to avoid that each node generates

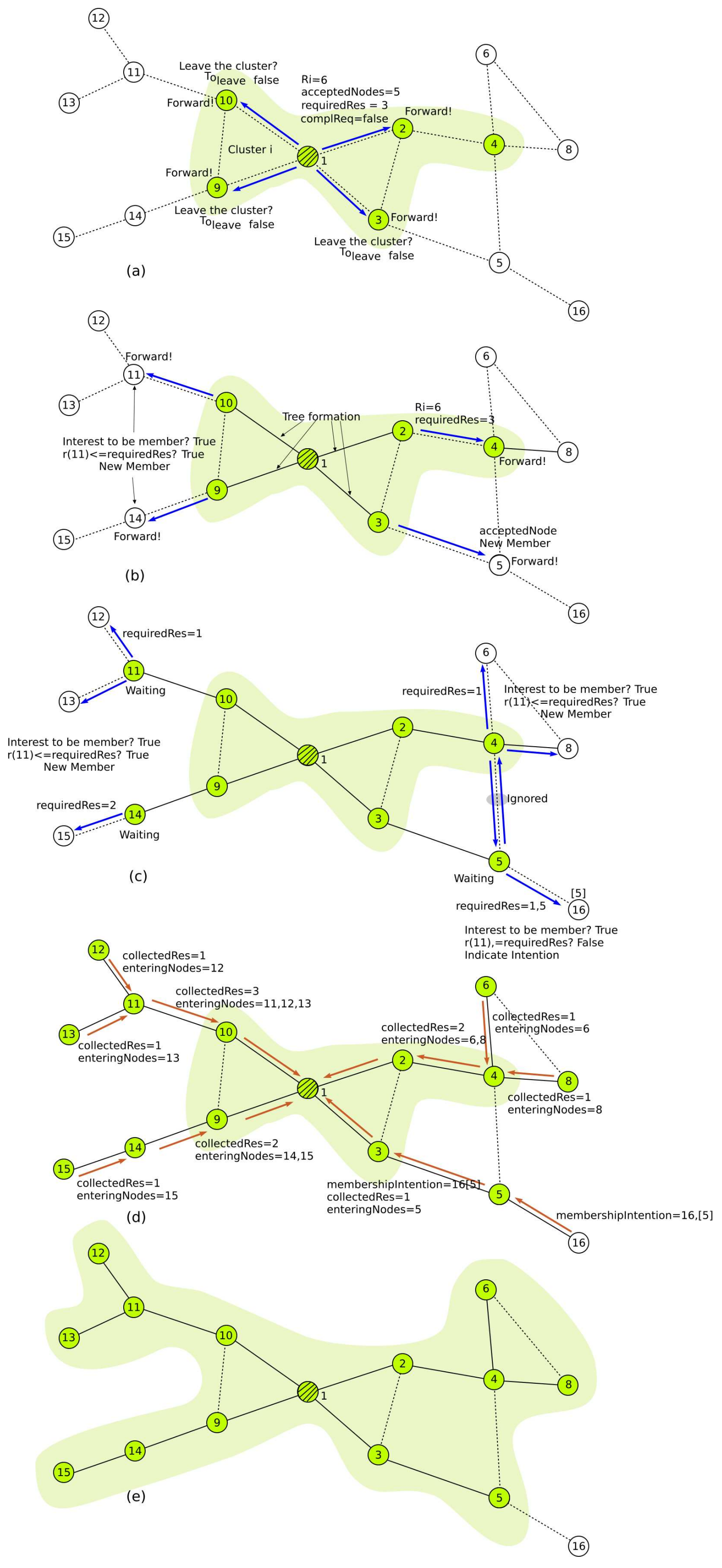


Figure 5.26: Example of a clustering round

its own clusteringBackward message, since the message should be generated just in the leaves of the tree. Nevertheless, due to the fact that in a *partial* round just parts of the tree where modifications are happening generate clusteringBackward messages, and also transmission errors can occur (even in a *complete* round), a corresponding clusteringBackward message will eventually never be received by a node. Therefore, the timer is important: whenever a node does not receive a clusteringBackward message after the timer elapses, it can generate its own new message.

Now the nodes 6, 8, 12, 13, 15, and 16 receive the message (Figure 5.26(c)). Those nodes are not yet members, and none of them is in the accetpedNodes list. Therefore, they test whether they should enter in the cluster using the response function $T_{\theta_{recr_{v,i}}}$. This test returns true for all of them. For the nodes 6, 8, 12, 13, and 15 there is enough budget, and they can thus be included in the cluster without sending a request. Node 16 must include in the message clusteringBackward a request to be member of the cluster (field membershipIntention). The nodes don't have any children to forward the message clusteringForward. Hence, they generate now a response message clusteringForward. In the case of a node not changing its status or willing to request membership, this node would not generate this response message in the case of a *partial* round.

Figure 5.26(d) shows the route of the clusteringBackward message. Due to the tree generation, the messages make the reverse path from the leaf to the root of the tree incorporating information that the intermediary nodes want to include to the clusterhead. During this phase, upon receiving a clusteringBackward, a node deletes its timer before retransmitting the message to its parent node.

In the previous example, the method to evaluate whether a node desires to enter in a cluster i was not presented in order to avoid excess of of details in the text. Now we will show one more example focusing on the influence of the attracting force that is controlled by a positive/negative feedback mechanism.

Consider the example depicted in Figure 5.27. As in the previous example, the figure shows the acquisition of members by a cluster under formation. In (a), the cluster possess resources of value 1 because just the clusterhead belongs to the cluster. The actual attracting force (stimulus $s_{recr_{v,i}}$,given by $k_1 \cdot (p(1) \cdot g(1)))$ is 4.7 units (for $k_1 = 1$). This is enough to attract vicinity nodes that have good connection to the cluster (in the example, nodes 2, 3, and 4).

Now the cluster has 4 members and $R_i = 4$. This brings to the cluster a higher potential to attract new members, and in the next cluster construction round (b) the nodes 5, 6, and 7 will be attracted. In (c), the actual situation of the cluster is shown. The stimulus to attract new members is considerably reduced. No new members will be attracted in this situation.

A main purpose of the positive/negative feedback mechanism is to control the competition among neighboring clusters. The feedback curves are designed in such a way that an already formed cluster may just loose some members till the q limit is achieved, because when this limit is achieved, the desire to attract new members is at maximum. In the same way, if there are two clusters under formation this method avoids that one cluster steals members of the other one, reaching the state where no cluster has fulfilled its requirement on resource. Figure 5.28 illustrates these two situations.

Figures 5.28(a) and (b) show a cluster under formation stealing some members of a complete one, but due to the positive/negative feedback, the cluster under formation with high probability can't attract a number of nodes that will disrupt the complete cluster. In (c) an example of two clusters under formation are shown. Due to the positive feedback, the cluster that has a small advantage due to a marginal difference in the number of nodes can attract a node of the other cluster, resulting in an even greater advantage. In the end of the process (d), instead of having two incomplete clusters, the cluster with the marginal advantage won and a complete cluster is the result.

The example shows how the feedback mechanism drives the cluster construction in a way that complete clusters are prioritized in detriment of incomplete ones.

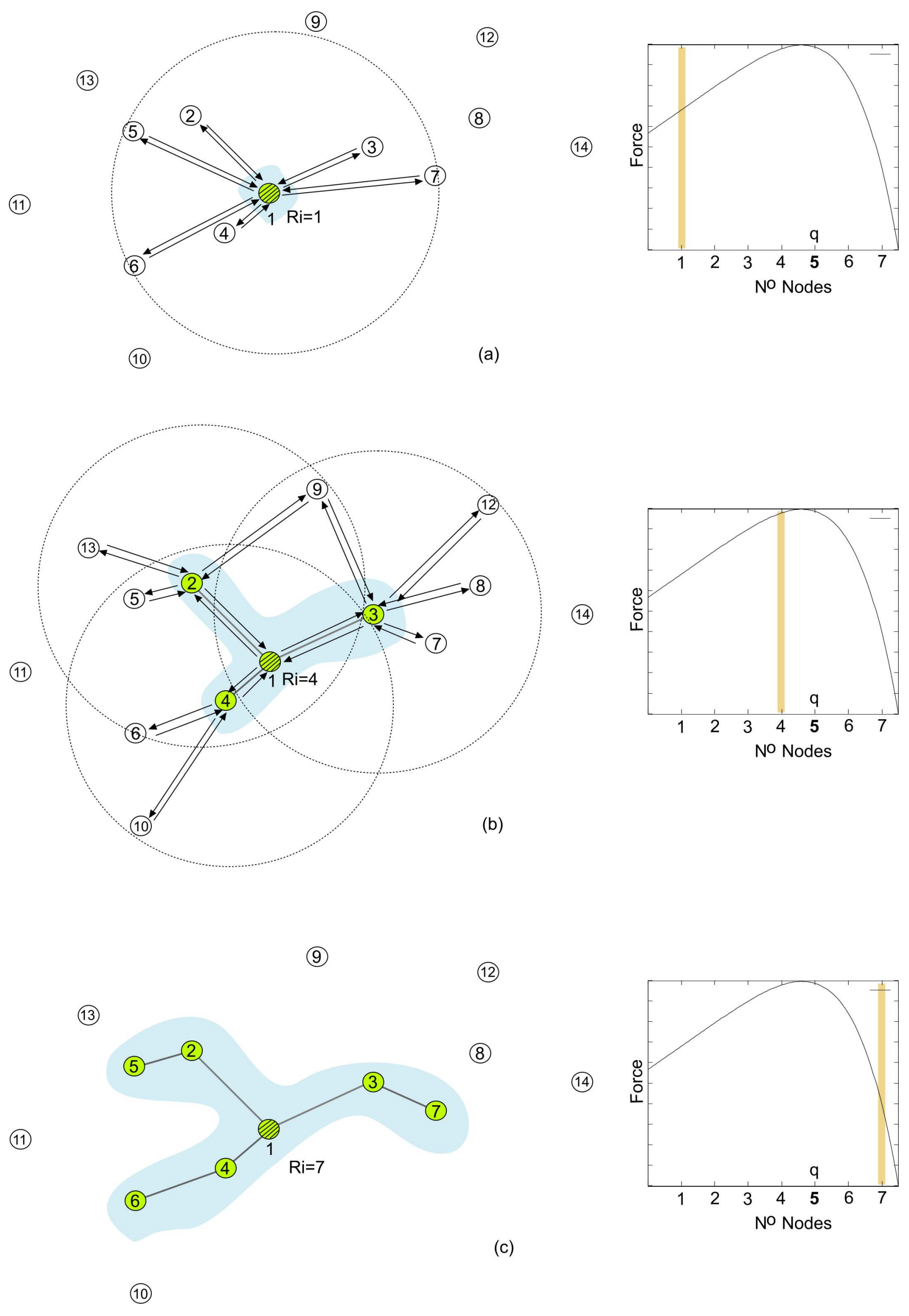


Figure 5.27: Example of cluster construction from the point of view of the positive/negative feedback mechanism.

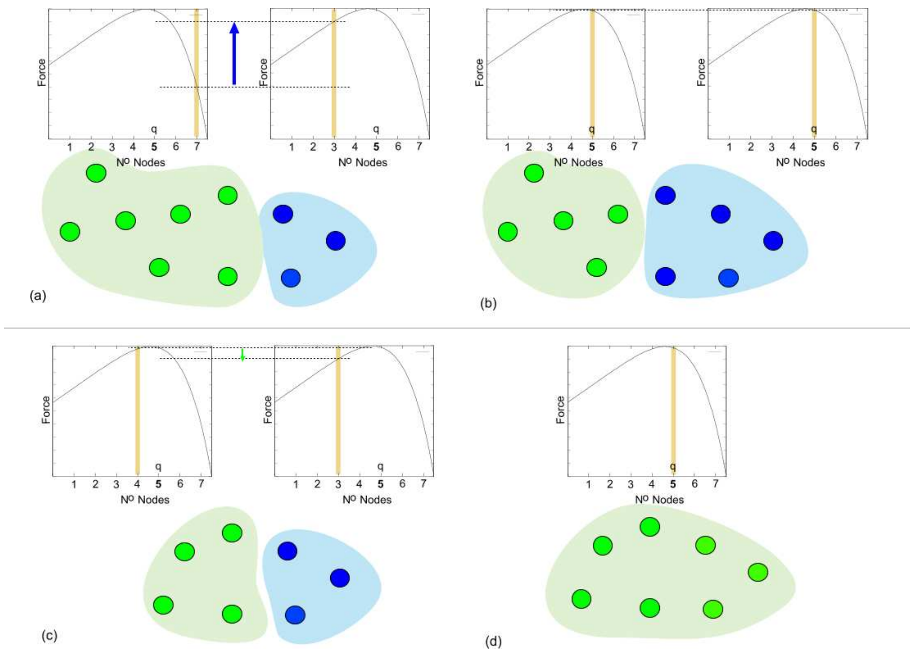


Figure 5.28: (a),(b): Example of a cluster under formation trying to steal nodes from a complete one. (c),(d): Example of two clusters under formation without enough resources for both.

5.7.5 Clustering Maintenance

After (at most) ζ cluster construction rounds ($round_i \leq \zeta$), either a complete cluster has been formed or the cluster has been disrupted due to the lack of resources in the area. Successfully constructed clusters enter in the clustering maintenance phase. It consists of detecting changes in the cluster and react upon them. The reaction is also performed using the cluster construction round.

The detection of changes in the cluster topology is responsible of all members of the cluster. The following mechanisms are used to detect changes:

Explicit Subscription/Unsubscription of members: Upon deciding to enter or leave an existing cluster (by means of the response functions), a node must inform this to the clusterhead. During the cluster construction round, this is done by the clusteringBackward message using the fields enteringNodes and leavingNodes. If, for any reason, a node must enter or leave the cluster when no cluster round is occurring, special messages are used to inform this to the clusterhead.

A reason for entering in the cluster by other means than the presented round appears when nodes try to form a cluster, but due to the lack of resources, the cluster formation is not successful. In this situation, the nodes use the enterCluster message to impose their entrance to any selected neighboring cluster. Enforced members are not considered in the clusterResource field of the clusteringForward message. Nevertheless, they should respond to this message.

When a node notices an abnormality, that can be some processing error or lack of energy, it can spontaneously leave the cluster using the message leaveCluster to inform the clusterhead about its decision. Upon receiving leaveCluster, the clusterhead checks whether the cluster still has enough resources ($R_i \geq q$). If not, a bunch of *partial* cluster construction rounds are used to fill the cluster again. Here just a *partial* request is necessary because only the replacement of the leaving node is necessary, if no other modification of the cluster has been noticed.

Perceived link metric (topology) changes: As already explained, each node of the network has a table with link metrics to the neighboring nodes. All packages received by the radio are used to update the link metric (even if the packet is not addressed to the observing node). If a node v perceives significant changes (over a certain threshold) in its neighborhood on nodes belonging to the same cluster, it informs the clusterhead with the message topologyChange. A significant change means a large change in the link metric or even the disappearance of a node of the neighborhood. Upon receiving the message, the clusterhead starts a bunch of *complete* cluster construction rounds in order to gather again the information about the membership of the cluster and eventually replaces a disappearing node. In this case a *complete* request is necessary due to the fact that links have changed and perhaps some nodes have lost a direct connection to the cluster (or even have some failure), being impossible to inform the cluster about this situation using the leavingNodes field of the clusteringBackward message.

5.7.6 Integrating Reference Point Group Mobility Model

Several mobility models have been developed in order to simulate dynamic topologies in ad hoc networks (for an in-depth survey, see [25] and [111, 142]). In order to develop a protocol for dynamic ad hoc networks, the mobility model adopted heavily influences the results due to the different behavior nature of the various models. There are two major types of mobility models: *traces* and *synthetic* models.

Traces use specific topology patterns observed in real-life. The *synthetic* models aim at realistically representing the movements without the necessity of the costly tracing process. In this work, we restrict our consideration to *synthetic* models.

Synthetic models are divided into two categories: *entity mobility models* and *group mobility models*. In the *entity mobility models*, nodes have individual pathways without influence from other nodes. The most common model in this category is the *random mobility model* [143]. In this model, the speed and direction of motion in a new time interval have no relation to their past values coming from the previous period. Although very popular, the *random mobility model* is not a very realistic one. It generates unrealistic mobile behavior such as sharp turning or sudden stopping. Several other variants of this model have been developed (e.g. *Random Waypoint* mobility model), random direction [109] or city section mobility model [41].

In the *group mobility models*, groups of nodes are moved along individual pathways. Some movement within the groups may exist. Those *group mobility models* are based on the assumption that for many applications the movement of a single node is correlated with the movement of some other nodes. This is a realistic assumption.

For this reason, we consider in this thesis the *reference point group mobility model* (RPGM) [63]. In the RPGM, the set of nodes is partitioned in groups. Each group has a logical center. The center's motion defines the entire group's motion behavior. Moreover, inside a group the nodes may have independent random motion. This model captures several scenarios, from users moving around (in groups) with mobile devices to groups of sensor nodes attached in vehicles. It can be likely assumed, in our point of view, that nodes engaged in cooperative processing (like the system proposed in this thesis), move along correlated paths.

In the next section we will present a small overview of the reference point group mobility model. Afterwards, the seamless integration of this model in our dynamic clustering heuristic will be introduced.

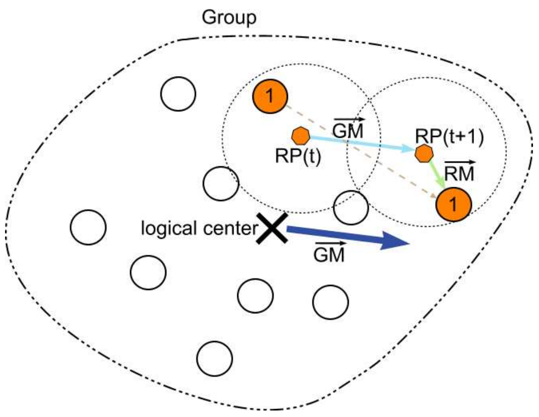


Figure 5.29: Example of a new node position in the group mobility model

5.7.6.1 Reference Point Group Mobility Model

As already said, in the *reference point group mobility model*, each group has a logical center. The movement of the center is controlled by the group motion vector called $\vec{GM}$.

Each node in a group has a so called reference point (*RP*), which follows the movement of the group. The reference point represents the center of the circular area where a specific node can be encountered. A node is randomly placed in the neighborhood of its reference point at each step of the group mobility model algorithm. Therefore, it allows independent random motion behavior for each node, in addition to the group motion.

Figure 5.29 gives an example of how a node moves from time tick τ to $\tau+1$. in Figure, the target node is highlighted. The group has the motion vector $\vec{GM}$. In order to calculate the new position of node 1, the reference point of this node is first moved from $RP(\tau)$ to $RP(\tau+1)$ with the group motion vector $\vec{GM}$. After this, the new node position is generated adding a random motion vector $\vec{RM}$ to this new reference point. The vector $\vec{RM}$ has its length uniformly distributed within a certain radio centered at the reference point, and its direction is also uniformly distributed between 0 and 2π radians.

5.7.6.2 Group Detection Algorithm

In order to improve the performance of the clustering algorithm in environments where the movement of the nodes happens in groups, the group information should be added to the clustering heuristic. For that, it is necessary that the nodes detect the current group they belong to. The following algorithm is responsible for this task.

In order to identify the groups, each group has an integer group ID ($gid \in \mathbb{N}$). The set of nodes in the group where $gid = i$ is given by g_i.

The following data structures are used by the algorithm:

Current Identifiers (*current_ids*): It is an array with two positions, the primary and the secondary *gid*. Each one stores a group ID. When two nodes have at least one of this two *gid* in common, they belong to the same group (the test to detect this condition we will call *group match test*).

Candidates (*candidates*): An array with *n* possible candidates to be included in the group. For each candidate, the node id and the current identifiers are stored. Moreover, a counter of the number of meetings and the time of the last contact are also stored, in order to make the detection of the group possible.

The current_ids are initiated with random values. At the beginning, with high probability, all nodes are in different groups. A beacon message *current_status* with the *current_ids* is sent by the nodes periodically.

Upon receiving a beacon from a node in the vicinity, the link quality of the received message is tested against the threshold θ_g. When the quality is higher than the threshold, the beacon is processed by the node. The possible actions are:

The *group match test* returns positive: This means that the beacon comes from a node that is in the same group. If the primary id of the beacon is the same or smaller than the received primary id, no further action is done. When not, upon receiving a higher id, the node updates its primary identifier, copies the old primary to the secondary field, and discards the current secondary id. This means that the group is changing its id. The rule used is that the higher id wins, i.e., higher identifiers are dominant.

Negative result: The node isn't yet in the same group. If the node is not in the *candidates* list, it is inserted. Otherwise, the counter for this candidate is incremented. If no message is received during a specified period of time (*timeout*), the node is deleted from the list.

When the counter in the candidate list reaches a predefined value $m \in \mathbb{N}$, the node should be considered in the same group. For that, its primary *gid* is compared with the node's own *gid*. If the candidate's *gid* is greater, the node assumes this as new group identifier and updates the secondary by the previous primary one.

After some time, the algorithm will converge, in a way that all nodes having periodic (or permanent) good links among them will have the same identifiers. A problem occurs if for some reason two groups achieve the same identifiers (due to a "contamination" effect or when a group is divided in two, for example). In order to cope with this possibility, offering a consistent and robust functionality, the following mechanism is introduced. Periodically, each node tests whether it should propose two new group ids (they will be the primary and secondary ids). The probability of proposing the new ids is given by ρ_{new_gid}. It should be very small in order to avoid cyclic id changes. When deciding to update the current group id, a node calculates a new primary group id (gid_{t+1}) by $gid_{t+1} = gid_t + r$ where r is a random integer variable distributed uniformly between $[1, m]$. It then upgrades its primary *gid* to this new value and copies the actual primary *gid* to the secondary, overwriting it. Now, this change will be propagated among the group members. After some time, the node generates again a new primary group id. The process repeats. After these two rounds of new primary ids, two different groups that originally have the same *current_ids* will diverge.

In order to avoid periodic beaconing, this protocol can be optimized by embedding the necessary information in messages from other protocols running in the sensor network.

A small example of functionality of the algorithm can be seen in Figure 5.30. In (a), the nodes 1,2,3,4, and 5 have already received one beacon from the neighboring nodes. Nevertheless, they have the random initialized *gid*. The neighbors are in the *candidates* list. In (b), each node has already received enough beacons and recognized the neighbors as belonging to the same group. As already described, the higher primary *gid* replaces the current *gid* when the neighbor is recognized to belong to the same group.

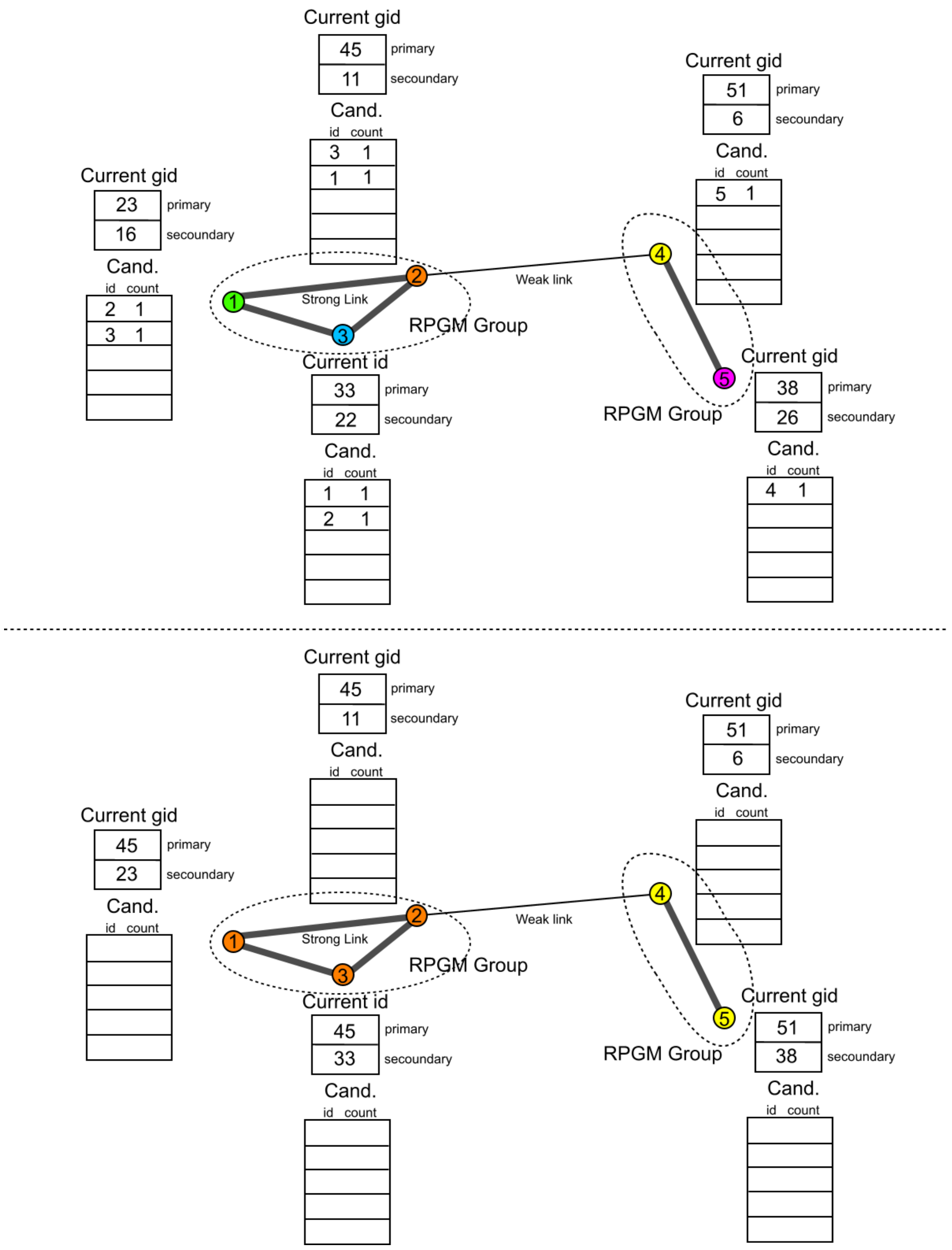


Figure 5.30: Example of a group detection. In (a) the process is beginning, the neighbors have already received the first beacon. In (b) the nodes have already received *m* beacons, therefore the higher primary *gid*'s overwrite the smaller ones.

5.7.6.3 Integrating Group Mobility in the Clustering Algorithm

The integration of the group mobility in the dynamic clustering is based on the group detection algorithm. The response functions will be altered in the clusterhead and membership selection in order to take advantage of the group information gathered by the presented algorithm.

First, it is important to detect whether the groups have been already detected in the neighborhood of a node v. Let $NG(v)$ be the set of the group identifiers of the neighbors of node v (i.e., $i \in NG(v)$ iff $\exists e \in V | e \in (g_i \cap Ngb(v))$). Let $Ngb_{g_i}(v)$ be the set of neighbors of node v that are in the group whose $gid = i$ (i.e., $e \in Ngb_{g_i}(v)$ iff $e \in (Ngb(v) \cap g_i)$). The function $gr : V \rightarrow [0,1]$, presented in Equation 5.22 measures how much the neighborhood of the node v is in the same group and the neighborhood's heterogeneity. Higher values bring a higher confidentiality that the neighborhood has already detected the group.

$$gr(v) = k_1 \cdot \frac{|Ngb_{g_l}|}{\left(\sum_{i \in NG(v)} |Ngb_{g_i}|\right) + |Ngb_{g_l}|} + k_2 \cdot \left(1 - \frac{|NG(v)|}{|Ngb(v)|}\right) \tag{5.22}$$

Where $v \in g_l$. The first term measures the percentage of the neighborhood that is already in v's group. A node with the majority of neighbors in other groups can suppose, with high probability, that its group has not been detected until now (or it is at the border of a group). The second term measures how heterogeneous is the neighborhood, i.e., to how many different groups the neighbors belong to. Higher numbers of groups indicate that the detection of the groups haven't converged yet.

This group detection rating will influence how much the group information will be used by the clustering algorithm. Lower values of the $gr(v)$ point out that the group information is not very trusty and it does not have to influence heavily the clustering process. Conversely, higher $gr(v)$ values indicate that the clustering process should use this information to avoid unstable clusters with members in different groups.

Let's now integrate it to the clustering heuristic. The function gr should influence both clusterhead and membership selection.

Clusterhead Selection The gr function is introduced in the already presented equation 5.9. This can be seen in eq. 5.23 (for $\sum_{j=1}^{4} k_j = 1$). It is possible to negatively influence the clusterhead emergence in the network areas where the groups haven't been already detected.

$$\theta_{gCH_v} = k_1 \cdot \left(1 - \frac{\sum_{u \in Ngb_{Nm}(v)} w(u,v)}{|Ngb_{Nm}(v)|}\right) + k_2 \cdot (1 - E_v) + k_3 \cdot \overline{W_v} + k_4 \cdot (1 - gr(v)) \tag{5.23}$$

Membership Recruitment As already presented, the membership recruitment is based on the equation 5.18. We define in Equation 5.24 a new threshold taking in account the group information. It is based on the version of the threshold without group ($\theta_{recr_{v,i}}$).

$$\theta_{grecr_{v,i}} = \underbrace{\theta_{recr_{v,i}} \cdot (1 - gr(v))}_{\text{No detec.}} + max\Big(\underbrace{\theta_{recr_{v,i}} \cdot gr(v)}_{\text{Detec. same group}}, \underbrace{k \cdot (1 - |\{Clusterhead_i \cap g_l\}|) \cdot gr(v)}_{\text{Detec. diverse group}} \Big) \tag{5.24}$$

The first part of the equation has its contribution to the final threshold controlled by the actual group detection rating of the node v (gr(v)). If no group is detected, this will be the only term of the expression, i.e., it will act exactly as the original $\theta_{recr_{v,i}}$. Nevertheless, as the neighborhood nodes are unifying their view about which group they belong to, the information about the group is receiving a higher importance, as can be seen in the second and third parts of the equation. The second is active when the v's group is the same of the clusterhead. If not, the third part assigns a value $k \geq 1$ to the threshold, meaning that this node should not be preferred for inclusion into the cluster.

5.8 Relation to Self-Organization Principles

In this section, we will analyze the relation of the two presented cluster construction heuristics to the self-organization design principles.

In the paper [101], four design paradigms for applying self-organization in networks are suggested. The idea of the paper was to discuss which principles from self-organized systems in nature and other fields can be successfully transferred to communication systems.

The presented paradigms were:

Design local behavior rules that achieve global properties: Instead of introducing a central entity responsible for coordinating the achievement of the desired property, the responsibility should be divided among the individual entities, where each one contributes to a collective behavior. Local behavior rules must be designed in such a manner that, when applied to the set of entities, lead to the desired global property.

Do not aim for perfect coordination: exploit implicit coordination: In conventional systems, it is natural to avoid conflicting situations by means of using *explicit* coordination, i.e., signaling messages are exchanged in a request-response manner to coordinate resources. The new idea here is to *tolerate* conflicts if they can be managed in a contained manner. Implicit coordination means that coordination information is not communicated explicitly by messages, but inferred from the local environment. For example, a node can observe other nodes in the neighborhood and, based on that observations, draw conclusions about the state of the network.

Minimize long-lived state information: In many approaches the nodes must maintain a (global) long term state information about the network. To achieve a higher level of self-organization, the amount of such long-term information should be minimized.

Design protocols that adapt to changes: The nodes should be capable to adapt to changes in the network and their environment.

Now we analyze the two proposed clustering construction heuristics based on these basic paradigms for self-organizing systems. We start analyzing the cluster selection method. Both heuristics have a similar method for the clusterhead selection. Each node tests itself using a response threshold to decide whether it should turn to a local clusterhead. This complies with the first paradigm, i.e., each node has strictly local behavior rules that just use local information about the neighborhood. Moreover, the second paradigm is also included: we tolerate conflicts (e.g., neighboring clusterheads) and, as described in the link metric, the nodes observe the neighborhood by listening to ongoing communications and, based on them, make conclusions about the current link status. This is implicit coordination. The long-lived state information is also minimized: each node must just know its own and immediate neighborhood states. There is no need for large state information. Moreover, information about the neighborhood can be obtained partially using observations of ongoing communications.

The last design paradigm is clearly encountered in the clusterhead selection of the dynamic algorithm: when the situation changes and not enough nodes are available for forming a cluster with the minimum resource requirement, the clusterhead withdraws its role and the cluster ceases its existence, giving opportunity to the nodes to join an other (hopefully) complete cluster.

In the membership selection it is also possible to find the key elements the self-organization. In the quasi-static clustering, the membership selection is also based on a local behavior rule: each node waits a determined time before answering for the membership request message. This time is also calculated using just local information, minimizing long-lived state information.

The positive and negative feedback used to evaluate the attraction force in the dynamic clustering is based on local rules too. The attraction behavior is also a local interaction among neighboring nodes. Moreover, nodes in two extremes of the cluster are attracting new members independently, what may generate temporarily inconsistencies (e.g. a cluster much larger than the desired size). Nevertheless, after some interactions, the size adjusts itself to a reasonable number of nodes.

Although in the dynamic clustering the membership selection has several characteristics of a self-organizing system (even using the two basic mechanism that are required for self-organization in a more strict sense - positive and negative feedbacks), it also has an aspect that is not conform to the self-organization design principles. The propagation mechanism of the actual amount of resources in a cluster works in a wave-style broadcast (using the *clusteringForward* and *clusteringBackward* messages). Therefore, all nodes of the cluster receive this information. This is necessary because the nodes do not have the information about the actual resource state of the cluster without this exchange.

The design principles used in both heuristics make them robust against environment changes (in particular topology changes in the dynamic version) and scalable (there is not a central commander, i.e., no bottleneck) and save energy by means of localized communication and implicit coordination. The price for those features is the possible existence of a temporarily inconsistent cluster state in some parts of the network (specially in the dynamic heuristic). Moreover, for both heuristics, there is no guarantee about the quality of the generated solutions.

Chapter 6

Simulation and Results

6.1 Simulation Environment

In order to simulate the heuristics presented in the previous chapter, we decided to use the Shox network simulator. The Shox simulator has being developed in our working group and it targets simulations of wireless ad hoc networks. Shox is a discrete-event simulator and assumes that an event may occur at any point in a continuous time. Each event has a *timestamp* that indicates when it should be processed.

This means that the lifetime of the network is discretized into events. The simulation generates series of events that the network would have stepped through when a specific input is given. The execution of an event involves returning it from a sorted queue and feeding a component in the system with it. This component will execute some logic and generate new events. The time is not absolute but a virtual quantity kept in a variable [121].

The algorithm being tested using the simulator is programmed as a set of event handlers inside the selected network layer. The simulator sorts the events chronologically in a queue. An executive unit takes the events one by one and calls the appropriate event handler in the network layers. New events may be insert into the global queue by event handlers.

For better introducing the Shox simulator, we divide its architecture in two main parts. The first is the event architecture, which shows which kind of events are processed by our simulator. The structure of the nodes in the ad hoc simulation are presented in the second part of this section.

The class diagram of the objects related with the events structure is shown in Figure 6.1. Besides the event architecture, the figure also presents the central class of the whole simulation framework: the SimulationManager. The SimulationManager knows all global objects of the simulation (e.g. Topology) and controls the whole flow. The event handling is realized by this class, which contains a reference to the global event queue (priority queue).

As already said, the priority queue is responsible for maintaining all events scheduled for execution in the system. Instead of directly putting the events into the global queue, an envelope object is used (class EventEnvelope). It stores further information about the event and provides a method to process the event encapsulated in the envelope. The Event class is an abstract superclass that represents all the events in the simulation. Events can be of the following types: simulation internal (like movements), for a whole node (ToNode), and for a node and a specific layer (ToLayer).

Simulation events are used internally by the simulation and cannot be issued by the protocols being simulated. Examples of such internal events are movement events that are responsible for changing the physical position of a given node (and, as consequence, the network topology).

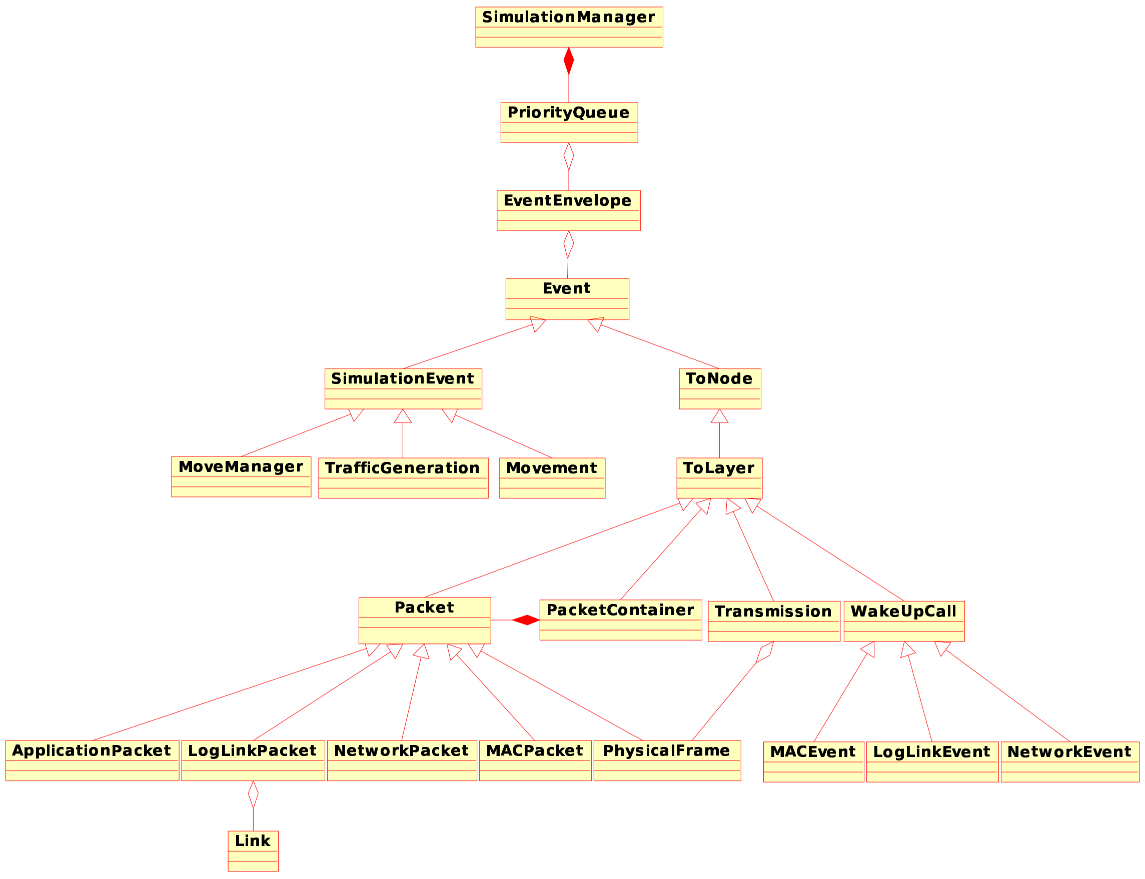


Figure 6.1: Simplified class diagram of the Shox simulator showing the event architecture.

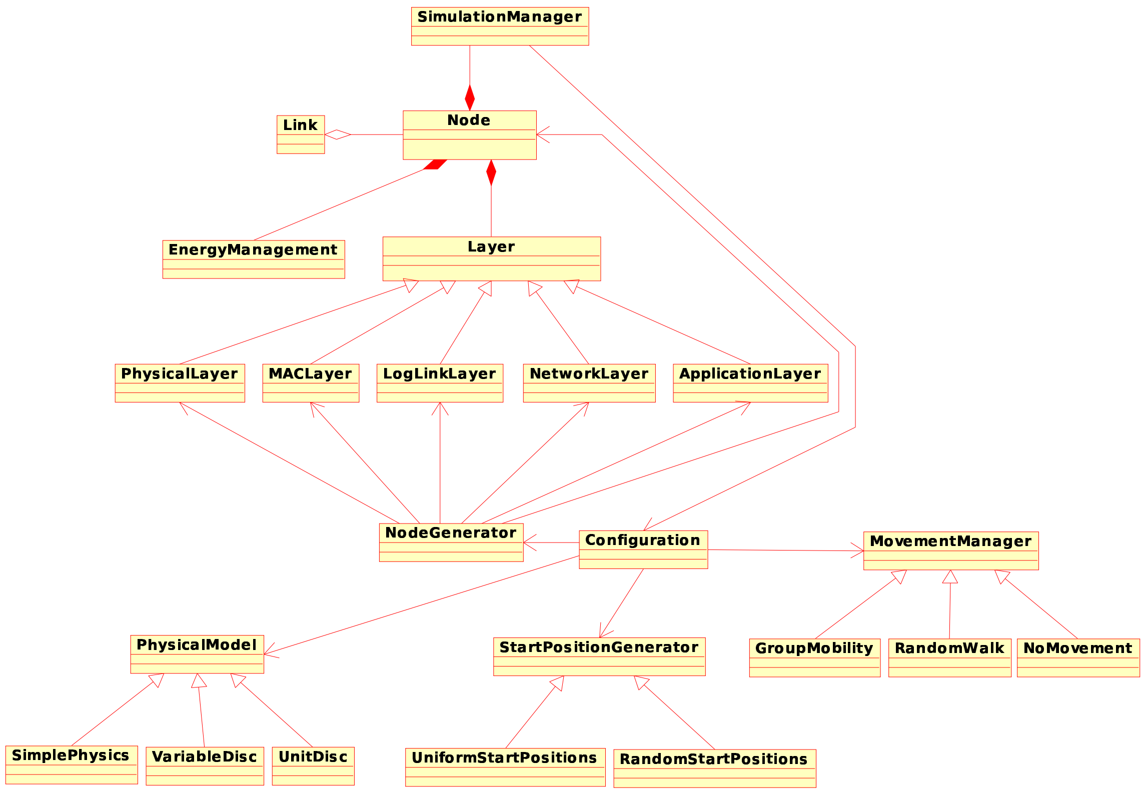


Figure 6.2: Simplified class diagram of the Shox simulator presenting the structure of the nodes.

ToNode events are directed to all layers of a node. An example of such a kind of event is initialize that is responsible for the initialization of a given layer. When this event is launched, all layers of a given node receive it.

ToLayer events are dispatched to a certain layer of a given node. The most important event of this type is the packet: packets are addressed to a given layer of a target node. Logically, every layer may communicate with the same layer of another node. Nevertheless, physically, the packets must travel through the lower layers before being sent through the physical network. As usual for a stack network model, a given packet contains all packets of the upper layers. This is implemented in Shox by means of an aggregation presented in the class Packet.

Another very important ToLayer event is the WakeUpCall. It is used to schedule future events in the system. For example, after sending a packet, the logical link layer, in some implementations, have to wait for an acknowledgment. For this, a timeout is set, and a WakeUpCall event is issued when the timeout expires.

The second part of the Shox architecture represents the structure of the nodes. The class diagram can be seen in Figure 6.2. The central element is the Node. A node is specified via its layers in the network stack. The concrete protocol layers are derived of this abstract Layer object. The simulation manager has the complete collection of nodes of the system. Each node of the simulation corresponds to one instance of the class Node and all its layers.

During the initialization phase, all necessary node instances of the system are generated by the NodeGenerator. The Configuration class is responsible for readinf the configuration of the simulation (an XML file) and for controlling, among other things, the generation of the right number of node instances with the selected layers.

The algorithms developed in the context of this thesis were implemented in selected layers of this stack hierarchy. For example, for the implementation of the link metric, a class derived from the LogLinkLayer was used.

Some advantages of the Shox simulator against other approaches (e.g. ns-2 [2]) are:

- Possibility of implementing new protocols in a very simple and fast way, due to the clean object-oriented design principles. For implementing a new protocol, it is just necessary to derive the new layer from an existing one and extend it. Differently from ns-2, it is not necessary to modify any other source code in the simulator. This means that the new modules have low coupling with the existing Shox code.

- Advantages of the Java language. With Shox, the protocols are implemented in a high-level language. This makes them less error-prone. For example, packet memory deallocation is not necessary due to the garbage collector present in the Java language.

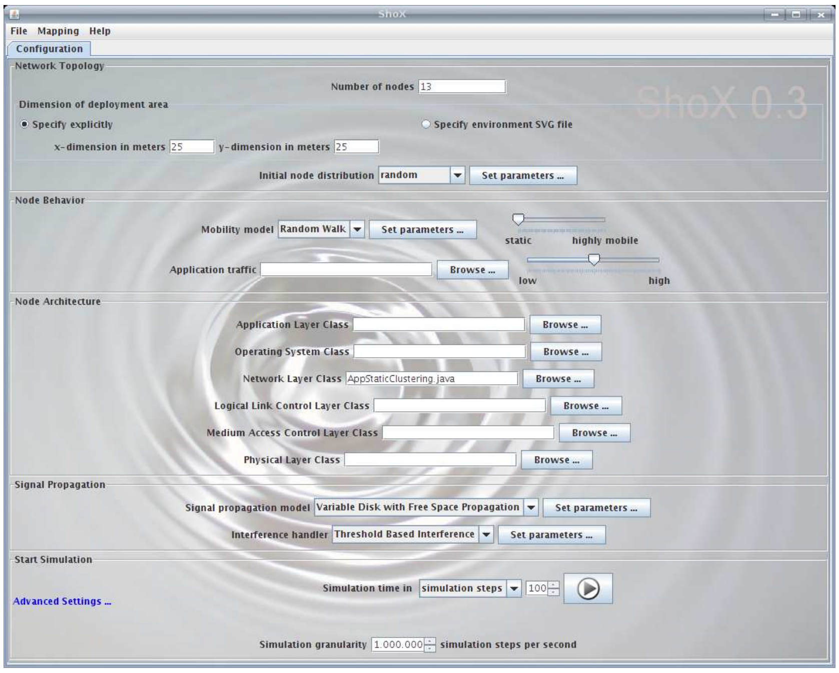
- Comprehensible, flexible configuration. The simulator is configurable using human-readable XML files. Moreover, configurations can be done using a GUI and saved in the XML file. A screenshot of the configuration screen is presented in Figure 6.3.

- Designed exclusively for wireless ad hoc networks simulations. Differently from other approaches (e.g. ns-2, OMNeT++), our simulation engine was developed to cope only with wireless ad hoc networks. This means that the complexity of configuring a general discrete event simulator to target wireless ad hoc networks can be avoided. Moreover, the size and complexity of the simulator can be kept inside manageable limits. In addition, the user is confronted just with parameters and modules that are correlated with ad hoc networks simulations.

- Possibility of demanding visualizations. Our simulator has an integrated visualization tool that enables efficient visualization of the state of the network as well as packet exchanges. When implementing new protocols, the programmer may issue commands in order to attach an attribute to a given node or link. This attribute is composed of name and value. Values may be a countable or non-countable element. All attributes are written in the log file. Before starting the visualization, the user may map the attributes to visual elements. Sequential countable elements receive distinct visual representation (for example, different colors). The non-countable attributes, when visualized, are represented by means of degrees (for example, a link attribute may be represented by the line thickness). An example: in our clustering algorithm, the clusterId is a countable attribute, nodes belonging to different clusters are represented with different visual elements (color of the nodes). The link quality is a non-countable attribute in our simulation: links vary from very good (near zero) to bad quality (near one) in a gradual way. The mapping of a non-countable attribute to some visual element respects this scale.

6.2 Reference Methods for the *Minimum Intracommunication-cost Clustering*

In this section, we will present the reference methods used to normalize the results of our experiments.



Network Topology

Number of nodes 13

Dimension of deployment area

● Specify explicitly ○ Specify environment SVG file

x-dimension in meters 25 y-dimension in meters 25

Initial node distribution random

Node Behavior

Mobility model Random Walk

Application traffic

Node Architecture

Application Layer Class

Operating System Class

Network Layer Class AppStaticClustering.java

Logical Link Control Layer Class

Medium Access Control Layer Class

Physical Layer Class

Signal Propagation

Signal propagation model Variable Disk with Free Space Propagation

Interference handler Threshold Based Interference

Start Simulation

Simulation time in simulation steps 100

Simulation granularity 1.000.000 simulation steps per second

Figure 6.3: Screenshot of Shox configuration dialog.

6.2.1 Modeling as a Integer Linear Program

In order to calculate the optimal solution for each *minimum intracommunication-cost clustering* instance and to compare it to our heuristic algorithms, the problem was modeled with Integer Programming.

Given a *minimum intracommunication-cost clustering* instance (G, w, r, q), the problem is modeled as:

$$\textit{minimize} \sum_{s,t \in V} \sum_{u,v | \{v,u\} \in E)} w(\{u,v\}) \cdot (\alpha \cdot r(s) + (1-\alpha)) \cdot x_{u \to v}^{s \to t} \tag{6.1}$$

where $x_{u \to v}^{s \to t}$ is a binary variable and $w : E \to \mathbb{R}^+$ return the weights of the links. $x_{u \to v}^{s \to t}$ is the binary variable that indicates whether the edge $\{u,v\}$ is being used in the direction from u to v in the path between nodes s and t. r, q, and α are defined in section 5.3. Figure 6.4 shows an example of graph modeled in this way. In the figure, just the possible links between nodes g and j are shown. It is important to remark that the direction that the link is used is also contained in the binary variables.

It is important to highlight that, for each par of nodes, we are summing two times the cost (e.g., for $s,t \in V$, we are adding the cost of the path s,t and t,s). This means that the cost equation (eq. 5.1) is multiplied by two. This is considered in the complete chapter.

The idea behind this modeling is that, given a problem instance, the solution will be the selection of the links that are connecting each two nodes inside a cluster with the shortest path. Other links that are not used for this purpose are simple deleted from the graph. The result is a graph where intra-cluster communication links exists and inter-cluster ones are deleted.

The second binary variable of the problem is $e^{s \to t}$. It has the meaning that there is a connection between nodes s and t and both nodes are in the same cluster.

Let $u,v \in V | \{u,v\} \in E$, $v,o \in V | \{v,o\} \in E$ and $s,t,j \in V$. The following constraints must hold:

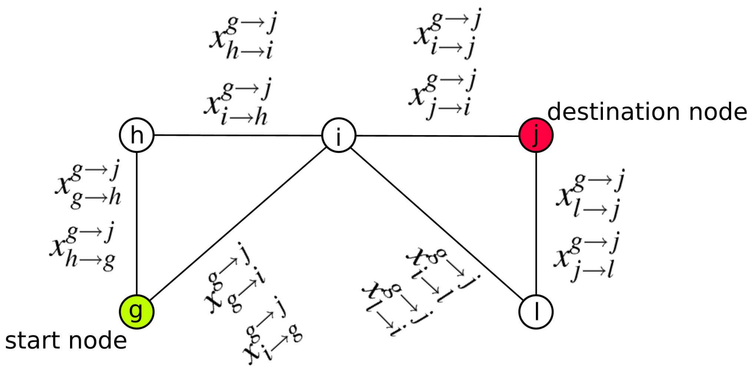


Figure 6.4: Example of modeled graph with integer programming. In the example, just the variables corresponding to the path between nodes g and j are shown.

$$\forall s,t \sum_{v} x^{s\to t}_{s\to v} = e^{s\to t} \tag{6.2}$$

$$\forall s,t \sum_{u} x^{s\to t}_{u\to t} = e^{s\to t} \tag{6.3}$$

$$\forall s,t,v \sum_{u} x^{s\to t}_{u\to v} - \sum_{o} x^{s\to t}_{v\to o} = 0 \ (v \neq s,t) \tag{6.4}$$

The first constraint (eq. 6.2) states that if the nodes s and t are connected (i.e., $e^{s\to t} = 1$ and s and t are in the same cluster), one and at most one link must exist from the node s to a neighboring node that is in the path connecting s to t. In the same way, wherever s and t are in the same cluster, an edge belonging to the path connecting s and t, starting from a neighboring node from t and adjacent to t must be present (eq. 6.3).

The equation 6.4 states that links belonging to the path $s \to t$ must pass trough a node or avoid it (if the node is neither s nor t). It is just possible to have an outgoing link in a node if you have an incoming link.

The three constraints guarantee that the solution will encompass just a single path between any two nodes in the cluster. Moreover, this path will be the shortest (due to the minimization objective). The constraints 6.2 and 6.3 enforce that when two nodes are in the same cluster ($e^{s\to t}$), there must exist a path between them.

Let's state the next constraints.

$$\forall s,t,u,v \quad x^{s\to t}_{u\to v} \leq e^{v\to t} \ (v \neq t) \tag{6.5}$$

$$\forall s,t,j \quad e^{s\to t} + e^{t\to j} \leq 1 + e^{s\to j} \tag{6.6}$$

The eq. 6.5 certifies that all nodes inside the path between two nodes in a cluster (s,t) must be also inside the cluster. This assures the connectivity of the cluster. The eq. 6.6 assures that if nodes s and t are in the same cluster and also nodes t and j, the nodes s and j must belong to the same cluster (transitivity).

$$\forall s,t,u,v \quad x^{s\to t}_{u\to v} = x^{t\to s}_{v\to u} \tag{6.7}$$

The eq. 6.7 states the symmetric relation among two connected vertexes. If vertexes s and t are connected through the edge $\{u,v\}$, the inverse path must be also connected through $\{v,u\}$.

Now the constraint that determines the minimum amount of resources per cluster:

$$\forall s \sum_t r(t) \cdot e^{s \to t} \geq q - r(s) \tag{6.8}$$

The eq. 6.8 expresses the requirement that a node in a cluster must have enough partners in the cluster to cover the minimum required amount of resources per cluster.

6.2.2 Genetic Algorithm

Because we cannot calculate the reference cost of large instances of the *Minimum intracommunication-cost clustering* with the linear integer program, we decide to construct additionally a genetic algorithm (GA) for this purpose. Therefore, for small instances of the problem, the reference solution is the optimal one, calculated via integer linear programming. For large instances, we will compare our distributed, low computational complexity heuristic with the results obtained using this GA.

6.2.3 Basic Concepts

In this section, we will briefly review the basic concepts of a genetic algorithm.

A genetic algorithm is a search heuristic based on the principles of the evolution and natural genetics [119]. They imitate the basic principles of life reproduction and evolution and can be advantageously used for many combinatorial optimization problems. They have been introduced by Holland ([62]) and are based on the Darwin's principle of the evolution of species [108]:

- The population of individuals of certain specie have different properties and abilities.
- Nature creates new individuals with similar properties to the existing individuals.
- Promising individuals are selected more often by the natural selection than the not so promising ones.

The proprieties and abilities of an individual in a population is characterized by its phenotype, that is encoded in the genotype. There is a function that maps the genotype in a corresponding phenotype.

The individuals of a population do not remain the same, they change over generations. New offspring are created and they inherit some proprieties of the parents. The creation of the offspring is based on the recombination and mutation *operators* applied to the genotype of the parents.

The natural selection acts upon the reproduction - promising individuals are more often selected for the reproduction than low-quality solutions. Highly fit individuals are allowed to create more offspring than inferior individuals. With that, the average fitness of the population increases over time.

In the next sections, we will describe the elements of a GA and, at the same time, will introduce how those elements were designed for the purpose of solving our clustering problem.

6.2.4 Representation of the Problem (Coding)

Each possible solution of the optimization problem must be represented in form of a chromosome. Each genotype (chromosome) corresponds to a phenotype, that is, a valid solution instance of our optimization problem. A transformation exists between the genotype and the phenotype, i.e. to construct a valid problem solution based on the genetic information stored in the chromosome.

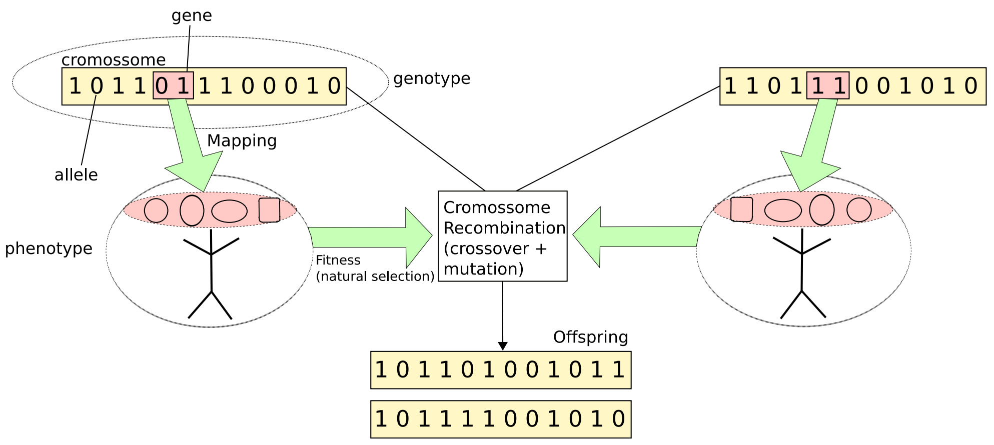


Figure 6.5: Overview of a genetic algorithm run. The valid solutions of the problems are encoded in the chromosome.

A chromosome is formed by several genes. A gene is a sequence of alleles that code one phenotype property of an individual. The alleles are the smallest information units in a chromosome. Often, binary representation is used, i.e. an allele can have either the value 0 or 1. Figure 6.5 depicts the difference between chromosomes, genes and alleles and also between genotype and phenotype. Moreover, a schematic showing the steps of a GA is also illustrated.

In the case of our clustering problem, we decide to use an integer representation for the genotype. Moreover, we are using a representation that is very straightforward. Due to this fact, similarly to problems where the direct representation is used, we need special operators (crossover and mutation). They are carefully designed to generate the desired phenotypes. The standard operators cannot be applied due this near to phenotype representation (but still indirect representation).

Our solution space is encoded as n sequences of clusters $c_1, c_2, ..., c_n$ where $\bigcap_{i=1}^{n} c_i = \emptyset$ and $\bigcup_{i=1}^{n} c_i = V$. Moreover, we define for our tests that each node of the network has one unit of resource. This means that the resource constraint is $\sum_{v \in c_k} r(v) \geq q$ for $k = 1, 2..n$, i.e. $|c_k| \geq q$.

This assumption simplifies the crossover and mutation operators, because they do not need to check for the resource constraint as it will be clear in the following paragraph.

We decide to model the problem with a fixed number of clusters, i.e., we assume that the optimal solution will have $n = \left\lfloor \frac{|v|}{q} \right\rfloor$ clusters. Our representation of the problem instances is an array of m integers; each integer represents a node (node ID), and this array is divided in n groups that correspond to the n clusters. Groups may have from q to $q + (\lceil \frac{|v| \mod q}{n} \rceil)$ integers.

Figure 6.6 presents an example of our representation of one instance of the problem. Moreover, how this genotype is mapped to a problem instance is also shown.

The representation was selected having in mind that the crossover operator effectiveness is highly correlated with the representation's quality. Our representation has the characteristic of preserving high performing partial arrangements (schemata, see [62]). This is because we decided to group the information about one specific cluster into a sequential part of the chromosome. The crossover operator will select two random points in a chromosome and exchange with the chromosome coming from the other parent. Using such kind of crossover, the locality of the chromosome is preserved with high probability, i.e., in our case, very good clusters will form the so called *building blocks* with fitness above the average, increasing their presence in the population. The concept of *bulding blocks* was introduced by Goldberg (see [54]) and can be defined as "highly fit, short-defining-length

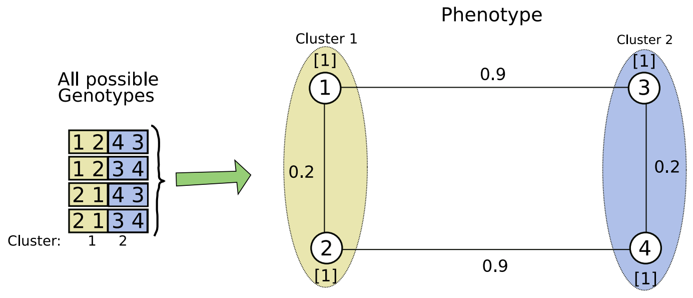


Figure 6.6: Example of the possible genotypes that maps to a given phenotype.

schemata" that "are propagated generation to generation by giving exponentially increasing samples to the observed best". In [108], the building block is described as a solution to a sub-problem that can be expressed as a schema.

Returning to our problem, clusters with high fitness, just with few wrong-selected members are exactly such kind of sub-problem solution and are increasing their presence in the population as described in the Goldberg work.

6.2.5 Crossover Operator

The crossover operator is responsible for recombining the selected highly fit individuals, creating new offspring. The operator is critical to the success of the GA. It is responsible for the exploration of new parts of the solution space and, at the same time, to guarantee the exploitation of the existing highly fit sub-solutions.

Because we use a representation that is near the problem and the standard operators cannot be used on it, we decide to use a modified version of the swap path crossover method presented in the work [3]. It was developed originally to the quadratic assignment problem (QAP), being suitable to other problems with chromosomes represented by permutations.

Our clustering problem has some similarities with the QAP. This happens if we consider the k nodes as possible placement points (sites) and we have also k facilities to be located in those points. If we consider $n = \left\lfloor \frac{|v|}{q} \right\rfloor$ the number of desired clusters, we can create n groups of facilities, with unitary communication between all elements of them. The distance between the sites (nodes) are measured using our link metric.

If we find a solution to this QAP problem, we will find a good solution to our clustering problem. Nevertheless, some constraints of the *minimum intracommunication-cost clustering* are not present in this instance of the QAP. We use it to illustrate how the crossover operator developed for the QAP can present good results to our clustering problem. As already said, instead of using a standard, problem-independent crossover operator, we are using the swap path crossover that has been developed to problems where the chromosomes represent permutations.

We will now describe the swap path crossover. Let I_1 and I_2 denote two individuals representing two selected solutions with a high fitness. The crossover operator produces two children from those parents, we will denote them I_3 and I_4.

The first step of the swap crossover is two select two chromosome positions, p_{start} and p_{end}. This part of the chromosome will be analyzed by the crossover operation in a cyclic fashion, from left to right. The first child, I_3, is generated based on the chromosome of the parent I_1, while I_4 is based on the parent I_2. Let's analyze the construction of I_3. As already said, it is based on the parent I_1,

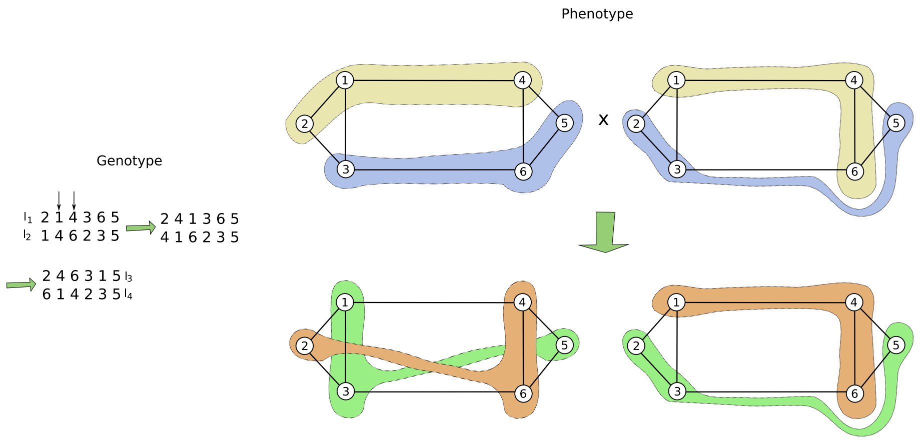


Figure 6.7: Example of the crossover operation, for $q = 3$ and nodes with unitary amount of resource.

and the alleles from I_2 inside the interval p_{start} and p_{end} will be copied to I_3. At the same time, the copies will cause swapping operations. Each position (allele) is analyzed. If the two alleles at the inspected position are the same, we move to the next position. Otherwise, the allele from I_3 will be swapped with another allele in the same chromosome that is identical with the inspected allele in the other parent (I_2). For generating I_4, the same mechanism is used (inverting I_1 and I_2). To clarify the presented concept, Figure 6.7 shows an example.

In the figure, we can see that some child may be exactly like the parent, although in a problem instance with bigger networks this is rare. Inserting nodes 1 and 4 in the first cluster, coming from I_1 in I_2 does not generate any change, i.e., the generated child is identical to one parent (I_2).

We can highlight an important characteristic of our genetic algorithm. We are allowing invalid clusters to be present in the population. For example, we have in the figure clusters where the connectivity propriety does not hold. Such invalid solutions are penalized by the fitness function and we will discuss this issue later on. The crossover operator may generate, from valid solutions, invalid ones and conversely. The results show, as expected, that the number of valid solutions increase in the population at each new generation.

6.2.6 Mutation Operator

The mutation operator is important to bring diversity in the actual population. It enables the optimization process to escape from local optima with respect to the crossover operation [65]. This is achieved by the introduction of random variation in the population.

In our program, we are transversing all alleles of a chromosome and testing whether they should be mutated. If the mutating test returns positive, a new random node is selected to substitute the actual node (allele). Similarly to the crossover, the swap operation is done in the mutating allele in order to keep the integrity of the chromosome.

In Figure 6.8, an example of our mutation operator is shown. In the example, each allele was sequentially tested whether it should mutate (given a mutation probability). When selected for mutation, the allele is exchanged with a random selected node using the swap operation.

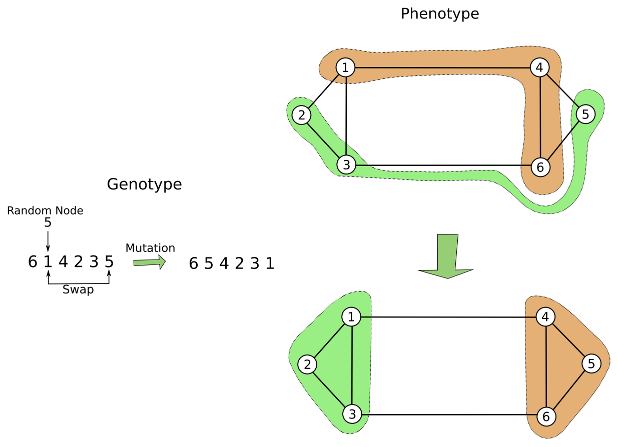


Figure 6.8: Example of the mutation operation, for $q = 3$ and nodes with unitary amount of resource.

6.2.7 Fitness Function

The fitness function estimates how good is a selected solution (chromosome) and it is used in the selection operator in order to select the appropriate parent to apply the crossover operator. For example, for a maximization problem the simplest way of developing a fitness function is to directly take the cost function, i.e., for a feasible solution α, the fitness is exactly $cost(\alpha)$.

It is important to remark again that we decide to allow invalid solutions in our population. But, as we will describe in this section, we are penalizing them. At the end of the GA's run, we ensure that the best solution found is a valid one. The penalization of the invalid solutions helps the heuristic to drive the population towards valid ones.

Let's consider the clustering problem instance (G,w,r,q) (see section 5.3). Every chromosome of the population has an equivalent cluster configuration $C_k = \{c_{k1}, c_{k2}, ..c_{k(nk)}\}$, where $c_{ki)}$ is the i^{th} cluster of the k^{th} configuration. This mapping from chromosomes to configuration is straightforward from the representation of the problem. Our fitness function uses a modified version of the cost function presented in the problem definition:

$$fitness = max\_cost - cost_p(C_k,(G,w,r,q)) = max\_cost - \sum_{i=1}^{nk} \sum_{u,v \in c_{ki}} DP_{p,c_{ki}}(u,v) \tag{6.9}$$

Where max_cost is the highest possible cost of a solution, $DP_{p,c_{ki}}(u,v)$ is a modified distance function that penalize links outside the cluster c_{ki}. We will call $p \geq 1$ the penalization factor. The definition of the distance is the same presented in the equation 5.2 of the section 5.3, but instead of using the already presented link metric, the following modification is applied:

$$WP_{p,c_{ki}}(u,v) = \begin{cases} w(u,v) & \text{if} \quad u,v \in c_{ki} \\ w(u,v)\cdot p & \text{otherwise} \end{cases} \tag{6.10}$$

With this modification, when the cost of a given cluster is being evaluated, all links external to the cluster are penalized. In order to better clarify, consider the example of Figure 6.9. In the example, the genotype describes the cluster configuration presented in the phenotype. The fitness function is composed by the cost of each of the two clusters. Like described in the figure, when calculating the cost of one cluster, all links not belonging to this cluster are penalized by the p factor. When a valid solution is presented, just links inside the cluster are used and no penalization takes place. The

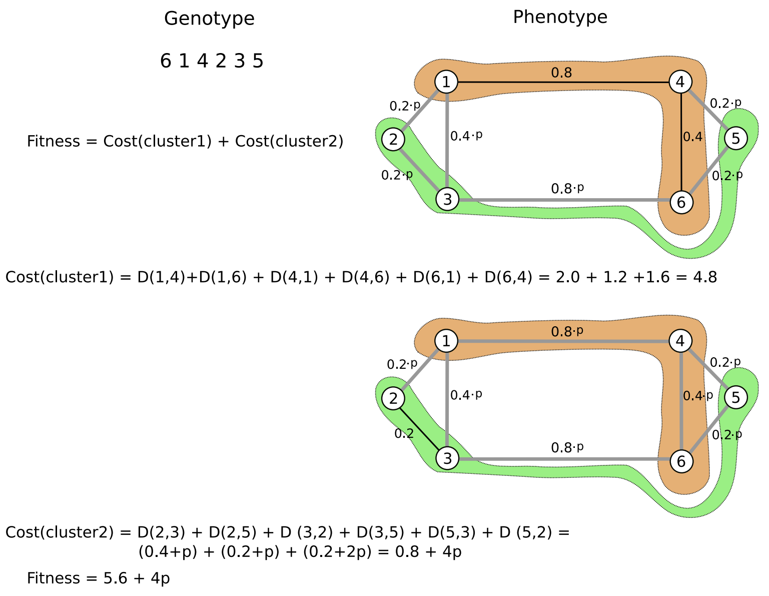

Genotype
Phenotype
6 1 4 2 3 5
Fitness = Cost(cluster1) + Cost(cluster2)
0.8
0.2·p
0.4·p
0.4
0.2·p
0.2·p
0.8·p
0.2·p
Cost(cluster1) = D(1,4)+D(1,6) + D(4,1) + D(4,6) + D(6,1) + D(6,4) = 2.0 + 1.2 +1.6 = 4.8
0.8·p
0.2·p
0.4·p
0.4·p
0.2·p
0.2
0.8·p
0.2·p
Cost(cluster2) = D(2,3) + D(2,5) + D (3,2) + D(3,5) + D(5,3) + D (5,2) =
(0.4+p) + (0.2+p) + (0.2+2p) = 0.8 + 4p
Fitness = 5.6 + 4p

Figure 6.9: Example of the fitness function. $p \geq 1$ describes the penalization factor for links external to the cluster being analyzed.

first cluster of the figure is valid and the cost is not penalized. The second infringes the connectivity constraint and has an increased cost.

The constant *max_cost* is calculated using the complete graph as a cluster but also applying the penalization factor to all links.

With this fitness function, the population is driven toward valid solutions.

6.2.8 Selection Operator

The selection is responsible for choosing two appropriate parents to apply the crossover operator. There are several types of selection methods, e.g. roulette wheel selection, tournament selection, etc. The individuals selected by the operator are those whose genes are inherited by the next generation. Therefore, it is important to select highly fit individuals in order to increase the selection pressure in the direction of good (optimal) solutions. The selection pressure drives the GA to improve the population fitness over succeeding generations [95].

The adjustment of selection pressure represents a trade-off: when the pressure is too low, the convergence rate is slow, taking unnecessary long time to find a good (optimal) solution. On the other hand, applying a too high pressure increases the chance of converging to an incorrect (sub-optimal) solution.

Because of its efficiency and flexibility, we decided to use in our work the tournament selection operator. The flexibility arises from the fact that adjusting the tournament size automatically changes the selection pressure. Another advantage of the tournament selection is its independence with regard to the absolute values of the fitness function - for the tournament selection, just a totally ordered set of the current population is important, and this is obtained easily using the described fitness function.

In the tournament selection, s competitors are randomly selected from the current population. s is the tournament size. The winner is the individual with the highest fitness among the competitors. This procedure is repeated in order to select all parents that will be matched by the crossover operation.

The average fitness calculated over the selected individuals is normally higher than the fitness of the entire population.

Increasing the size of s also increases the selection pressure, because the winner from a larger group has, on average, higher fitness than the winner of a smaller tournament [95].

6.2.9 GA Behavior

We test our *GA* in order to confirm its convergence and capability of escaping of local optimum. For the test, we use randomly generated instances of the clustering problem with 13 nodes randomly uniformly distributed in a $25m \times 25m$ wide area. This scenario is explained in details in the next section and it is called small-sparse (see Table 6.1).

For each instance, the a integer linear program was solved in order to find the optimal solution. For that, we use the lp_solve suite.

The following parameters were analyzed in the simulations:

Selection Pressure: We use three different tournament sizes for the purpose of experimenting with different selection pressures.

Number of individuals: A trade-off between time complexity and the probability of finding a very good solution is represented by the population size. A small population size increases the chance of converging to a local optimum whose fitness is far from the global optimum. A large population size increase the computational power necessary to execute the GA.

Mutation rate: A large mutation rate increases the diversity of the population. Nevertheless, a too large mutation rate may destroy very good schemata, hindering the heuristic's convergence.

Figures 6.10 and 6.11 depict the results of our experiments. The first figure presents the average cost of all individuals of the population whereas in the second one the best cost is picket up.

For all used parameters, the meta-heuristic has shown a very good and fast convergence. As expected, a higher selection pressure brought faster convergence. However, for $s = 3$, the best cost was achieved. It took a slightly larger time to converge, but it could better avoid local optimum.

The mutation rate had a similar impact on the convergence. A higher mutation rate (in our experiment, 0.01 instead 0.005 used in the other cases) brought a faster convergence. The same configuration with lower mutation at the end of 100 generations has almost the same fitness, but with slower convergence.

Increasing the number of individuals has resulted in higher convergence and very good final solution. But it increases the computation cost.

We conclude with our experiments that the reduction of the selection pressure up to acceptable level has presented the better final solution. Nevertheless, as it can be seen in the figures, the variation of the different runs are not very high and good solutions could be found in every run. This demonstrates the robustness and appropriateness of the design of our genetic algorithm.

6.3 "Quasi-Static" Clustering Heuristic Simulation

6.3.1 Assumptions

For our simulation, the following assumptions are used:

- Each node has a transmitter with fixed power. Therefore, the maximal reachability is fixed.

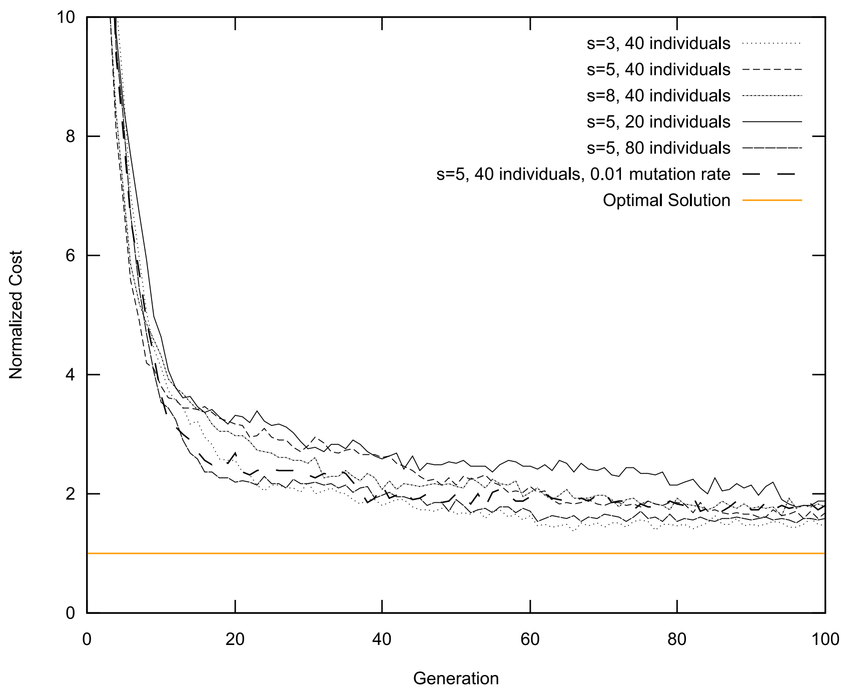


Figure 6.10: Average performance of the GA for different parameters. The normalized mean of the cost over all individuals is used in this diagram.

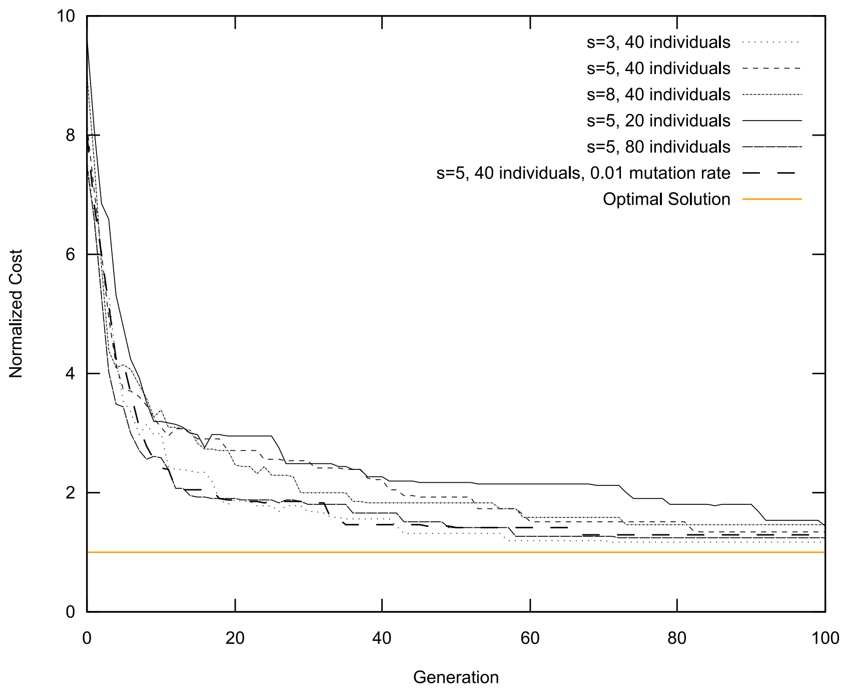


Figure 6.11: Average performance of the GA for different parameters. The best normalized cost over all individuals are presented.

- Links are bidirectional. In reality, this can be achieved just by ignoring unidirectional links.
- The Friis Free Space propagation model for isotropic point source in an ideal propagation medium is used to calculate the received signal strength indication (RSSI).
- Every node has one unit of energy.
- Every node has also one unit of resource.
- The weights of the equation parameters (clusterhead and member selection) were experimentally selected.
- We use $\alpha = 0$ (see eq. 5.1, Section 5.3).

6.3.2 Simulation Scenarios

Simulation scenarios are intended to mimic the real-word conditions as much as the simulation allows. In the same way, the network topology of the simulation scenarios should resemble real-work topologies. Unfortunately, the deployment topology is normally not known. For our simulation scenarios, we decided to use randomly deployed nodes. The deployment is uniform. Such kind of deployment can be statistically analyzed and the several characteristics inferred. In this section, we will present our concrete simulation scenario and also infer some characteristics of these scenarios.

In order to evaluate our "quasi-static" clustering heuristic, we decided to use two different groups of scenarios, each one with its own problem size. In the first group, we have a square field of $25m \times 25m$ where 13 nodes are randomly deployed with independent uniform probability. Here the desired cluster size is $q = 3$. The second group of scenarios has a square field size of $100m \times 100m$ with 120 nodes. The selected cluster size for the large scenarios was 10. Each node in our simulations has a unity of resource, therefore $q = m$ means that each cluster must have at least m nodes.

The small size scenarios were selected in order to enable the calculation of the optimal solution using linear integer programming (see Section 6.2.1). With the optimal solution, we can measure how good our heuristic is performing. In the large scenarios, we are using the described genetic algorithm to perform the assessment.

In each scenario inside one group, we select different radio ranges in order to vary the node density (measured by the degree of the node). An overview of the selected scenarios with small and large number of nodes are shown in Table 6.1.

Scenario Name	Field Size (m^2)	Number of Nodes	Cluster size (q)	Radio Range	Connection Probability	Node density	Average Degree (Theoretic)	Average Degree (Measured)
				Small Scenarios				
small-sparse	25x25	13	3	10	0.98	0.02	6.5	5.06
small-dense	25x25	13	3	17	~ 1	0.02	18.89	10.4
				Large Scenarios				
large-sparse	100x100	120	10	15.24	0.98	0.012	8.76	8.59
large-medium	100x100	120	10	19.45	0.999	0.012	14.26	12.74
large-dense	100x100	120	10	23.66	~ 1	0.012	21.1	17.98
large-very-dense	100x100	120	10	27.87	~ 1	0.012	29.28	23.48

Table 6.1: Overview of the different simulation scenarios.

For the tests, it is important to have a connected graph, i.e., there exist a multi-hop path between any pair of nodes in the formed graph. The connection probability was the parameter used to select the test scenarios. In order to calculate the connection probability, two parameters are important. The first one is the *node density*. In an uniform random distribution with large number of nodes n, we can define a node density $\rho = \frac{n}{A}$ where A is the deployment area. The second important parameter is the radio range. As we are using omnidirectional antennas with free space attenuation model, it is possible to calculate the *radio range* based on the transmitted power (P_0) and the sensibility of the receiver. The received power follows $P(r) \propto r^{-\gamma} P_0$ where r is the distance to the transmitter and γ is the path loss exponent, which depends on the environment. The transmission range can be mapped to the equivalent transmission power using a threshold for the receiver sensitivity [15]. For our simulation, the node density and the transmission range are shown in Table 6.1.

The connection probability of the resulting graph can be calculated by [15]:

$$P(d_{min} > 0) = \left(1 - e^{-\rho \pi r^2}\right)^n \tag{6.11}$$

for a homogeneous Poisson point process in two dimensions (the kind of graph formed by this process is called geometrical random graph). $P(d_{min} > 0)$ is the probability that the minimum degree of the network is higher than zero. ρ is the node density, r is the radio range and n is the number of nodes. The minimum degree of a network $d_m in$ is the degree of the node with the smallest degree of the complete graph.

It is important to remark that the equation 6.11 calculates the probability that no node will be isolated. The fact that there isn’t any isolated node does not mean that the graph is connected. Therefore, $d_m in > 0$ is a necessary condition for the graph to be connected, but it is not sufficient. Nevertheless, Penrose [99] has proved that for large n, the geometric random graph becomes connected at the moment that it achieves a minimum degree $d_{min} > 0$. This is valid for any graph generated in an euclidean plane with dimension higher than one.

The probability of being connected changes very fast from 0 to 1 as the range of the radio increases. This is called “phase transition” phenomenon in the random graph theory.

In Figure 6.12, the connectivity probability for a $25m \times 25m$ field is presented. The x axis represents the number of nodes and the z axis the radio range. For the small field, we select two radio ranges based on the connectivity probability. The first is slightly crossing the border from disconnected to connected network (10m range, 13 nodes). For the second, we decide to select a higher range where the connectivity probability is converging to one (17m range, 13 nodes). These two scenarios are highlighted in the figure.

Based on the Poisson distribution, we can calculate the average degree of a node in the network by $d_{avg} = \rho \cdot \pi r^2$, i.e., the average amount of nodes located in the radio area of a single node. For example, for the scenarios small-sparse and small-dense presented in Table 6.1, Figure 6.13 presents the probability mass function for the degree of a node. It is important to say that those numbers are expected when $n \gg 1$ and very large areas are used (or when the toroidal distance metric is used). The so called *border effect* explains the difference between the theoretic average node degree and the measured one. Nodes near the border may only have links towards the middle of the areal. Their node degree is, on average, lower than that of nodes in the middle. Therefore, the theoretical value of r is a lower bound for the range that is required in the simulation environment. The effect gets even worse when the relation between size of the field and radio range decreases, that is the reason for bigger errors in the experiments with larger ranges.

In the large scenarios, an analogous methodology was used to calculate their connectivity probability and average degree. The large-sparse scenario has the same connection probability of the

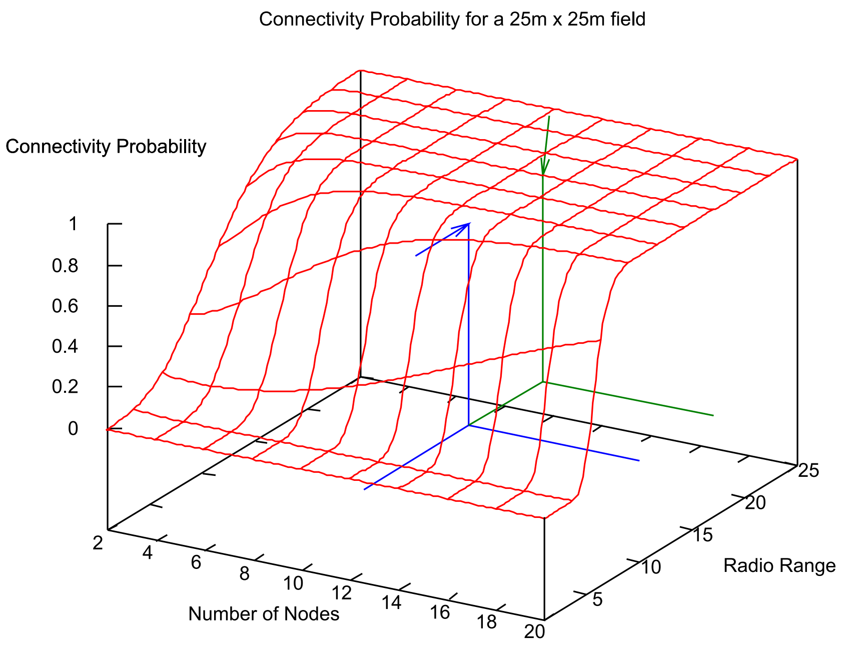


Figure 6.12: Connectivity probability for a 25$m \times$ 25m field.

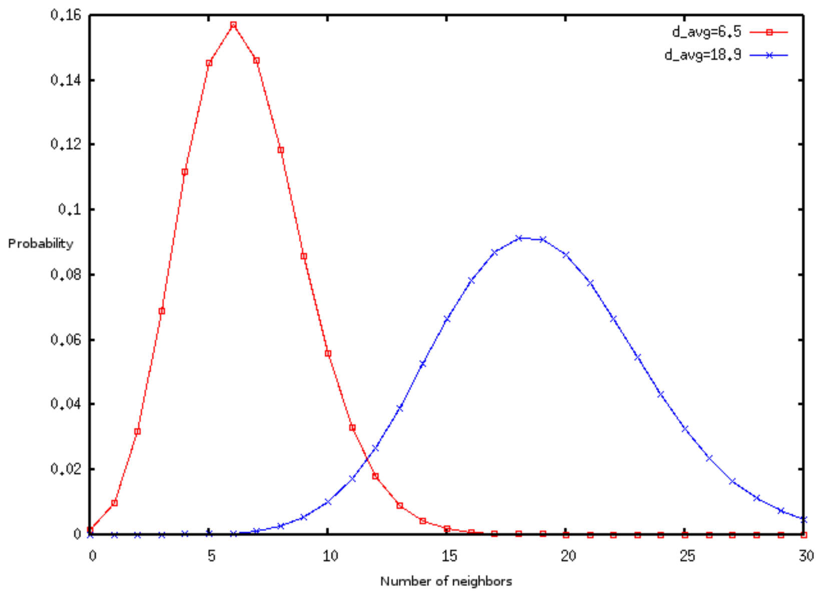


Figure 6.13: Theoretical probability mass function of the number of neighbors of a node for the small scenarios.

small-sparse. In the same way, the small-dense has the same probability of the large-dense. Two additional scenarios were included in this group (large-medium and large-very-dense).

6.3.3 Algorithms under Evaluation

We evaluate the described scenario using different algorithms. For the small scenarios, the optimum solution was calculated using the linear integer programming formulation presented in the section 6.2.1. For the small and big scenarios, the GA, our emergent clustering heuristic and an additional standard heuristic were simulated.

There is no other heuristic that is able to solve the clustering problem formulated in this thesis. Therefore, we had to adapt an existing one to do this task. We select the well-known expanding ring heuristic and adapt it.

In the original expanding ring heuristic, the clusterhead request for members in rounds. The first round collects members one hop away, the second two, and so successively until the cluster achieve q elements. In the original heuristic, this bound should not be overcome, in our heuristic it must be achieved and may be overcome. Moreover, instead of using just completely random clusterhead selection as in the original heuristic, we decide to introduce our clusterhead selection based in the division of labor in social insects in the expanding ring heuristic.

An additional important comment is that, in the results, we refer to our clustering heuristic as emergent clustering.

6.3.4 Results

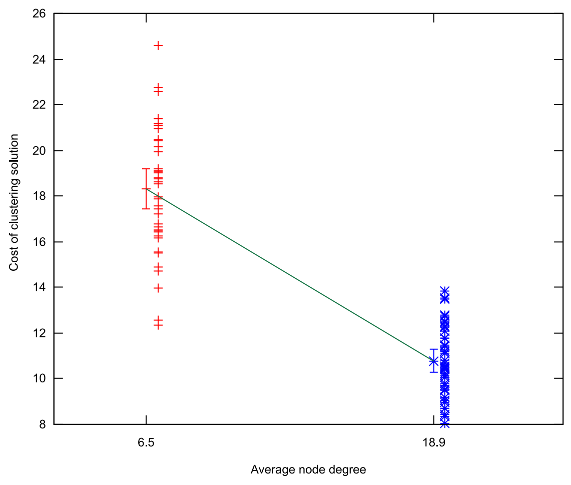
We realize 40 experiments for each scenario presented in the table 6.1. The results presented further in this section are based on these experiments.

6.3.4.1 Clustering Optimal Cost

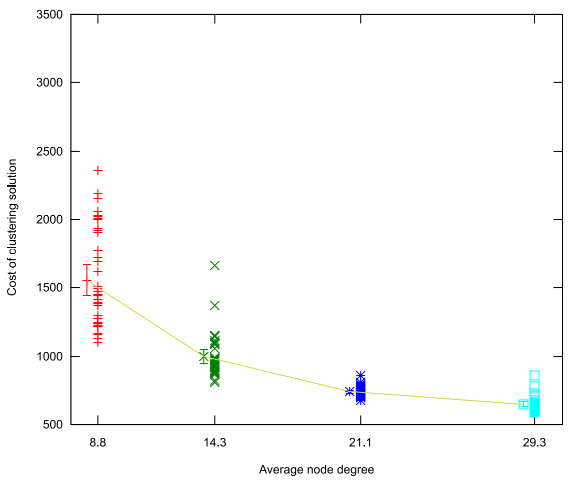
We will start analyzing how the node density influence the clustering cost for the reference solutions. As already said, the reference solutions are: optimal for small scenarios and GA for the large ones. The Figure 6.14(a) presents the optimal cost for the clustering problems of the small scenarios. As we can see, the sparse scenario has, in average, a much higher cost than the dense scenarios. This can be attributed to the fact than with larger radio ranges, each node has a larger number of well connected neighbors (with small link metric). The clustering possibilities are also much greater, which brings a substantial reduction of the cost when compared to sparse scenarios. Furthermore, the average link metric in the dense scenarios is smaller, fact that contributes to a smaller clustering cost.

Due to the lack of links between the nodes in the sparse scenario, a special situation that leads to large optimum costs for the these instances is depicted in the Figure 6.15. The three adjacent nodes with very good link metric must be split among two clusters, because the lack of links between the nodes in the extreme left and right sides. This poses also a challenge to our emergent clustering heuristic, as described later in this chapter.

The result for large scenarios is consistent with our argumentation for small ones. They are shown in the Figure 6.14(b). It is important to remark that the results here were not the optimal ones, but an approximation made using the described GA. As we have shown in the section 6.2.9 and will emphasize in the next one, our genetic algorithm has been correctly designed and is able to return very good solutions. Therefore, we can conclude that the described intrinsic characteristic of the *minimum intracommunication-cost clustering* is also found in scenarios with large number of nodes.



(a) Optimal cost (small-sparse, small-dense).



(b) GA cost (large-sparse, large-medium, large-dense, large-very-dense).

Figure 6.14: The cost of the *minimum intracommunication-cost clustering* for different node densities. For large scenarios, a genetic algorithm approximation is used.

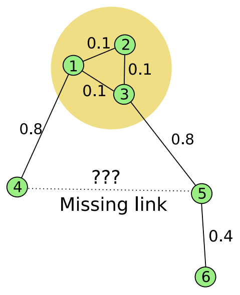


Figure 6.15: Example of scenario with increased cost for sparse networks ($q = 3$).

Nevertheless, the difference between the clustering costs of different densities becomes smaller when dense scenarios are analyzed. For example, the cost difference between large-sparse and large-medium scenarios is much higher than between large-dense and large-very-dense. This can be explained by a saturation of good neighbors: with a fixed q (10) used in the experiments, after some density, there are always enough good-quality neighbors that bring a small clustering cost.

6.3.4.2 Experiment Results

In this section, we present the outcome of our 40 experiments and make a first analyze. In Figure 6.16, selected results are depicted. The executions of the GA, Emergent Clustering and Modified Expanding Ring are normalized against the optimal cost in the small scenarios. In the large scenarios, the results are normalized against the genetic algorithm.

We will start analyzing the small scenarios. Figure 6.16(a) shows the results for the small-sparse scenario whereas Figure 6.16(b) shows the small-dense. For both scenarios, the GA could find, in many cases, the optimal solution. In most cases, the GA could find better solutions than the other two heuristics.

Nevertheless, in the small-sparse, the GA was overcome by our Emergent Clustering and the Modified Expanding Ring in some experiments. In general, the sparse scenario has less possibilities and the costs may change suddenly with very small modifications in the clusters. This affects the behavior of the GA in a negative way, because an easy hill climbing is not aways possible.

Although in the small-sparse scenario our heuristic had in average a better performance than the modified expanding ring in some experiments, the emergent clustering was defeated by the expanding ring. In the small-large, this was not the case in the majority of the experiments. Such behavior can be explained again by the degree of freedom in both node densities: sparse topologies do not leave a large choice range for different membership after the clusterhead has been elected. Therefore, the careful selection of members realized by the Emergent Clustering is restricted and the advantage of other clustering algorithms like the expanding ring is reduced. This increases the role of the initial selection of the clusterheads in the overall result of the heuristic. And the modified expanding ring uses the same method to select the clusterheads as the emergent clustering. Nevertheless, even with those restrictions, our heuristic in average beats the modified expanding ring. Average and standard deviation analysis are presented later on.

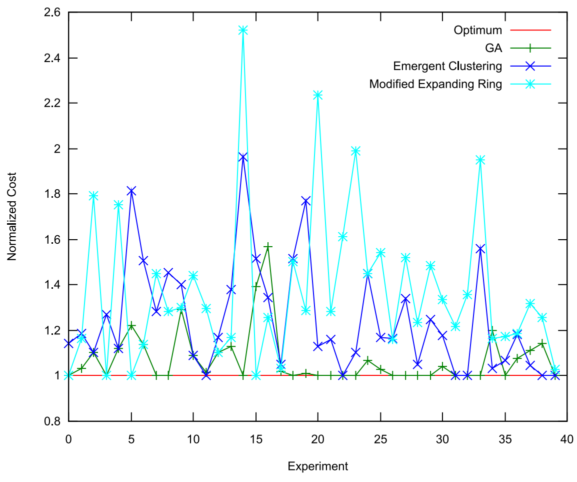
Our heuristic has relatively better performance (in comparison to the modified expanding ring) in the small-dense scenario. Here our membership selection mechanism could select suitable and well-connected members among the several available neighbors.

We encountered very similar results in the large scenarios, depicted in Figures 6.16(c) and 6.16(d). In the sparse scenario, the modified expanding ring could find a better solution for some test cases. The reason is the same as for the small-sparse scenario. In fact, the results of the small-sparse and large-sparse were very similar.

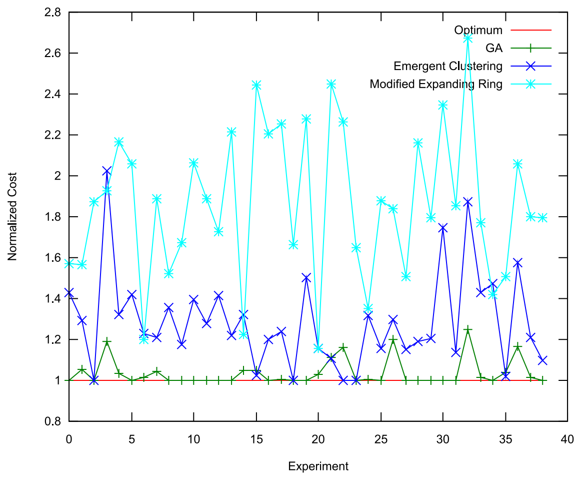
In the dense scenario, our emergent clustering algorithm beats the expanding ring in almost all cases. However, the difference of performance is less than in the sparse scenario. A reason for that will be presented later.

6.3.4.3 Heuristic’s Clustering Costs

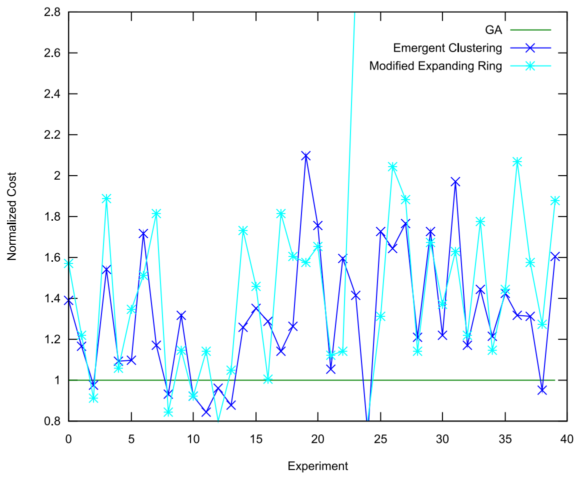
In this subsection, the average of the achieved costs of each heuristic for all scenarios will be presented. Figure 6.17 presents the achieved costs for all scenarios of Table 6.1. A confidence interval



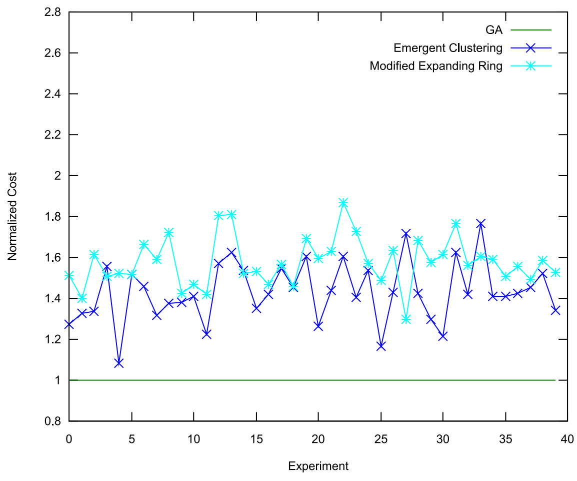
(a) small-sparse



(b) small-dense



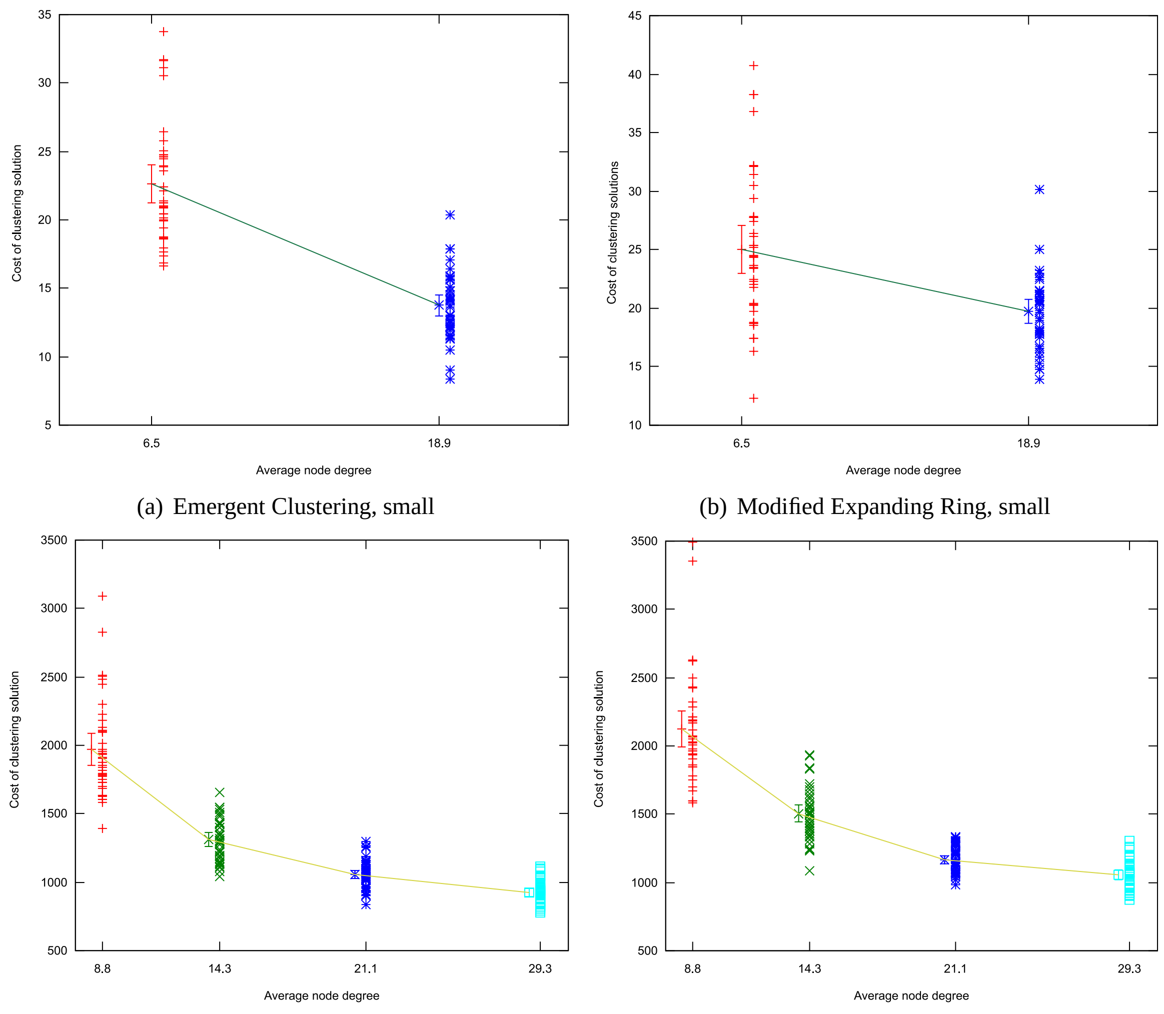
(c) large-sparse



(d) large-dense

Figure 6.16: Normalized results of selected experiments.

of 95% is also presented. For the confidence interval, we suppose that our runs are $X_1, \ldots, X_{40}$ independent samples from a normally distributed population with mean μ and variance σ^2. Further in this work, the probability density function of the samples will be presented in order to confirm our assumption of a normal distribution.



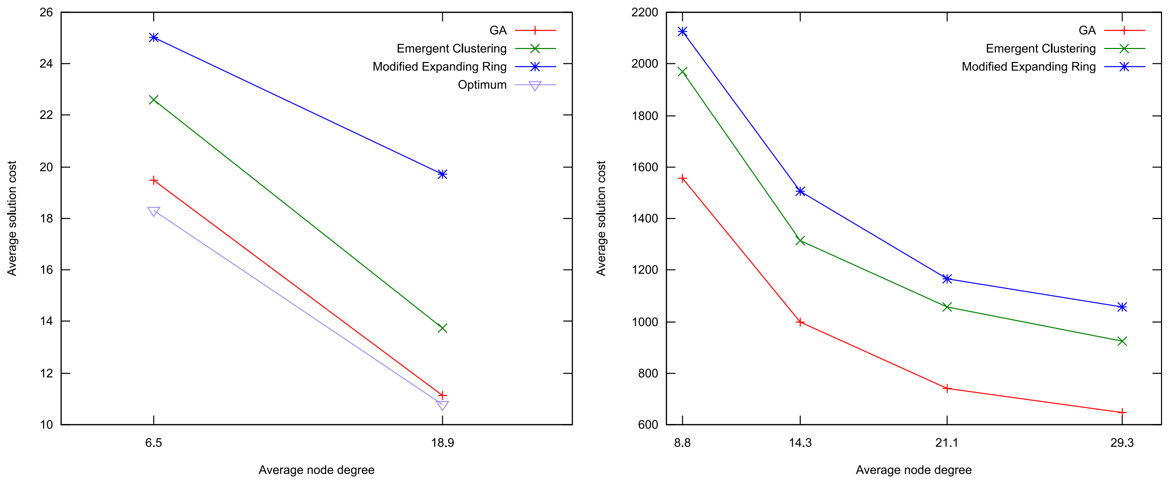
(a) Emergent Clustering, small (b) Modified Expanding Ring, small

(c) Emergent Clustering, large (d) Modified Expanding Ring, large

Figure 6.17: Clustering costs for the different heuristics with small and large problem size.

As we had 40 samples, we assume that the sample mean has a a Student’s t distribution with 39 degrees of freedom. By standardizing we get a random variable $T = \frac{\overline{X}-\mu}{S/\sqrt{n}}$, with $\overline{X} = \sum_{i=1}^{40}(X_i - \overline{X})^2$ and $S^2 = \frac{1}{n-1}\sum_{i=1}^{40}(X_i - \overline{X})^2$. With these assumptions, the confidence interval was calculated. As it can be seen in Figure 6.17, with increased density, the confidence interval is also reduced, because the variance of the samples is also smaller. The standard deviation of the samples will be presented in Section 6.3.4.4.

In the same way as the optimal solution, both algorithms (Emergent Clustering and Modified Expanding Ring) deliver a better cost, in average, for denser networks. This could be verified in both small and large scenarios. Moreover, the Emergent Clustering showed a better performance than the Modified Expanding Ring in all simulated scenarios. Figure 6.18 presents a comparison of the costs



(a) Small Scenarios (b) Large Scenarios

Figure 6.18: Overview of the costs for the optimum, GA, Emergent Clustering and Modified Expanding Ring solutions for all scenarios.

for the different algorithms. For the small scenarios, the mean cost of the GA, Emergent Clustering, Modified Expanding Ring and optimum are presented (Fig. 6.18(a)). Because the optimal result for the large scenarios is not known, Figure 6.18(b) does not present it.

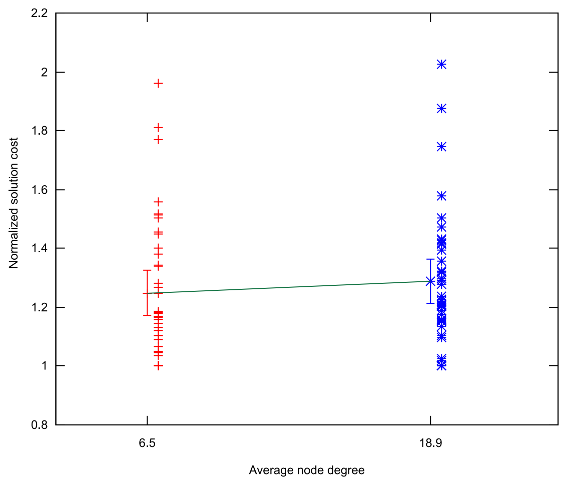
As it can be seen in Figure 6.18(a), the optimum and GA average costs are very close by, fact that allows us to use the GA for comparisons in the large scenarios. The difference of costs for those two approaches are slighter higher in the sparse network. The explanation for this is that small differences may bring very big cost changes in sparse environments, which affect the performance of the GA.

In the same way, the performance of our emergent clustering is affected by the lack of neighboring options in the sparse scenarios. Therefore, its performance has an improvement in comparison with the Modified Expanding Ring for dense scenarios. Nevertheless, this is much more visible in the small scenarios. In the large scenarios, our heuristic has still a better performance than the expanding ring, however, there is no effective improvement in comparison to the expanding ring in the dense scenarios. We will describe a possible reason for this below.

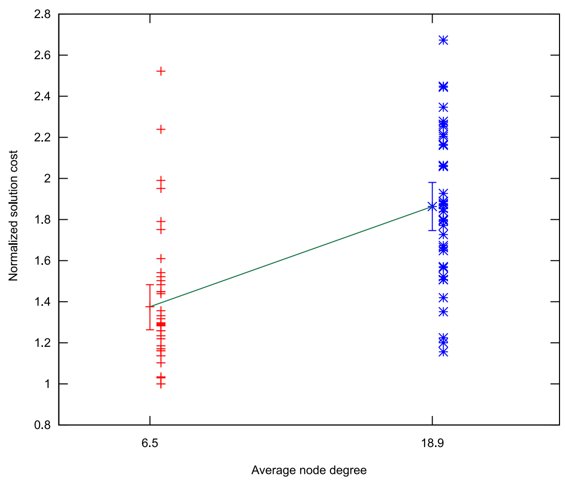
At this point we need to highlight again that the cost of the clustering reduces with the network density, but not linearly for all approaches. The reasons for that were already discussed in Section 6.3.4.1.

The Figure 6.19 presents the same achieved costs, but normalized against the reference solution. The average cost of our emergent clustering heuristic never overcomes 1.44 times the reference solution (optimal for small and GA for large), for all experimented scenarios. Moreover, the variation of the relative cost over different densities was not very high. There is a small tendency of increasing the relative cost with larger densities. This can be explained by the fact that dense networks present much more possibilities for the cluster construction, which are explored by the linear integer programming and GA, and, due the small computational cost requirement of our distributed heuristic, cannot be explored. In sparse scenarios, the solution space is smaller.

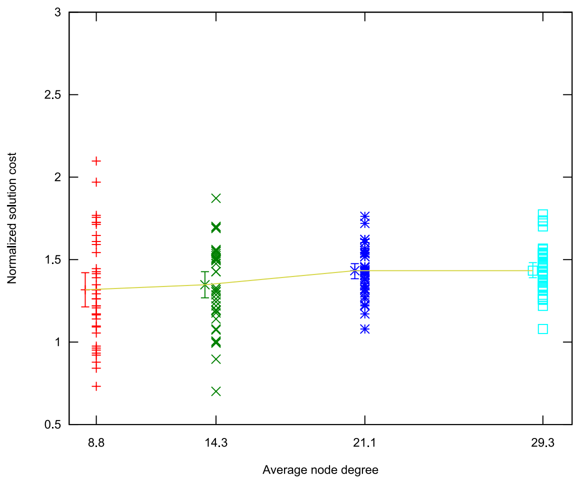
The modified expanding ring, for small scenarios, presents a much higher difference of performance when sparse and dense scenarios are compared. The fact that it does not rank appropriately the links when selecting the members makes it slightly improper for small and dense scenarios. This is especially important in the last round, and, in the dense scenarios, sometimes just one round is



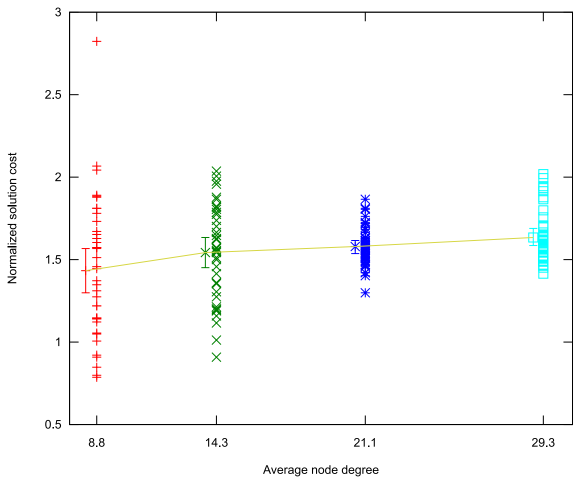
(a) Emergent Clustering, small



(b) Modified Expanding Ring, large



(c) Emergent Clustering, large



(d) Modified Expanding Ring, large

Figure 6.19: Normalized clustering costs for the different heuristics with small and large problem size.

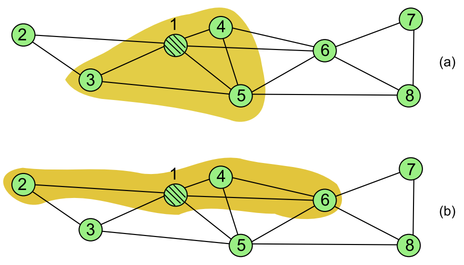


Figure 6.20: Example of situation where the selection of the best members by the clusterhead yields a bad global clustering solution ($q = 4$).

necessary to select all nodes, which makes this effect bigger. In sparse scenarios, there are fewer possibilities for membership selection in each round, which reduces the penalty of not ranking the candidates. Moreover, the effect of the last round is also smaller in small and sparse scenarios. One possible explanation for the relatively good performance of the expanding ring in large and dense scenarios is the fact that, without selecting the best candidates, some “bridge nodes” are left and can be used by other isolated nodes in order to form clusters. In our heuristic, those isolate nodes would not form any new cluster and will be integrated in the next cluster at the end of the heuristic.

Figure 6.20 exemplifies this situation. In (a), the clusterhead (node 1) selects the best possible members (using its local information). Nevertheless, now nodes 2, 6, 7 and 8 are isolated (there is no direct path that links node 2 with the others). Hence, they cannot build another cluster. They will be integrated in the same cluster by the enforce phase of the clustering heuristic at the end of the execution. In (b), the clusterhead in the expanding ring algorithm selects the new members of the cluster without differentiating them. Therefore, some path is left that connects the other four nodes. Now it is possible to form another cluster from nodes 3, 5, 7 and 8.

If we check the number of clusters generated by our heuristic and by the modified expanding ring (Section 6.3.4.6), this expectation is confirmed. The expanding ring, in average, is able to build more clusters than our heuristic in the big dense scenarios. For the small scenarios, due to the small cluster size, this effect is not such important. Nevertheless, even being able to construct more clusters than our heuristic, the global cost of the expanding ring is larger than the cost of our emergent clustering heuristic. This means that the clusters in the modified expanding ring are of very poor quality.

An additional point that must be highlighted is that not only the cluster cost is increased by the modified expanding ring. Due to the larger number of clusters where members of different clusters are geographically mixed (“blend effect”), a cluster-wide coordination of the MAC protocol in order to avoid interference of transmissions is hindered. Such cross layer optimizations are more difficult.

In order to illustrate the “blend effect”, the output of an execution of our heuristic and the expanding ring is shown in Figure 6.21. In this figure, just the best links among nodes inside one cluster are shown. Links among different clusters are hidden. We can see very clearly in this figure that the blend effect is much stronger in the clusters found by the modified expanding ring. In sparse scenarios, this is not so important.

Finally, in Figure 6.22, we summarize the normalized costs for all scenarios. Again, the effects discussed in this section can be easily observed. An additional important remark is that in the emergent clustering, in dense scenarios, the last cluster has the tendency of being very costly because all good links have been already used by faster clusters. This increases the global cost a little bit.

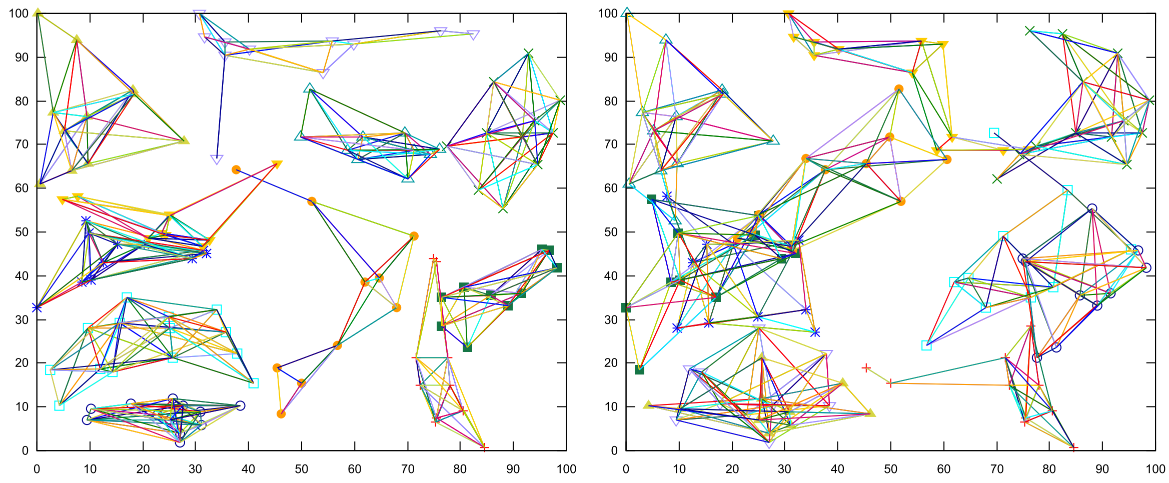
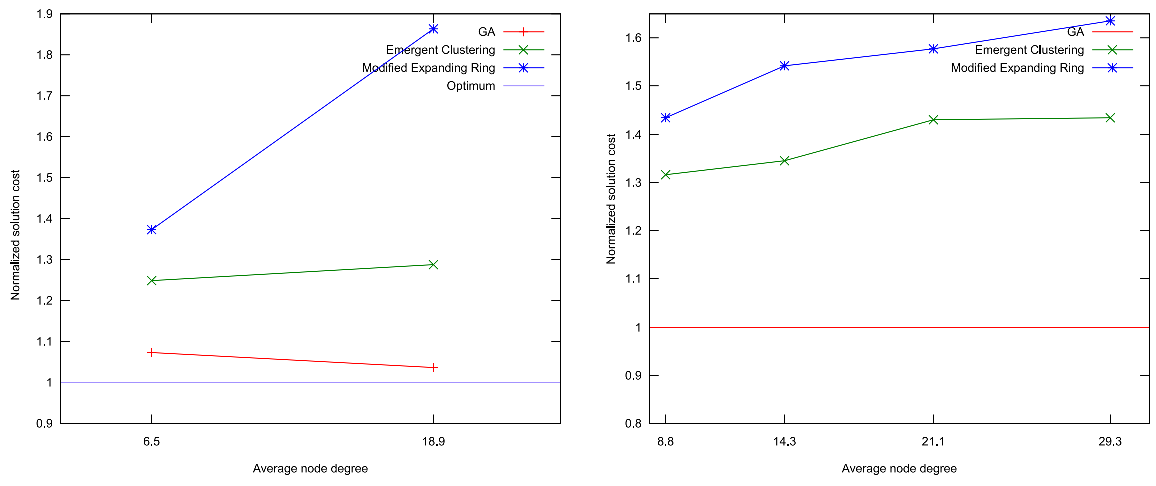
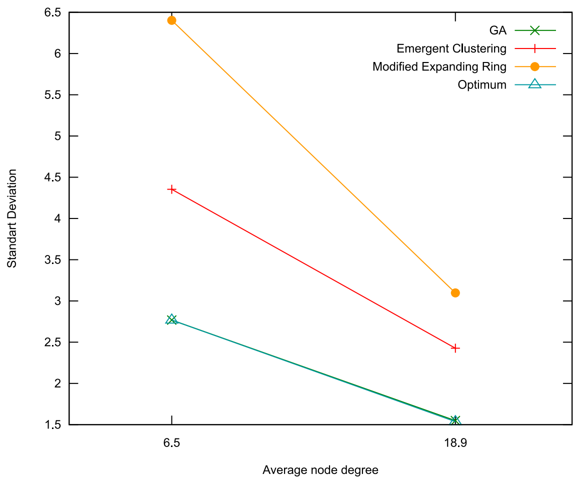

(a) Emergent Clustering (b) Modified Expanding Ring

Figure 6.21: Example of cluster solution found by the Emergent Clustering and the Modified Expanding Ring.

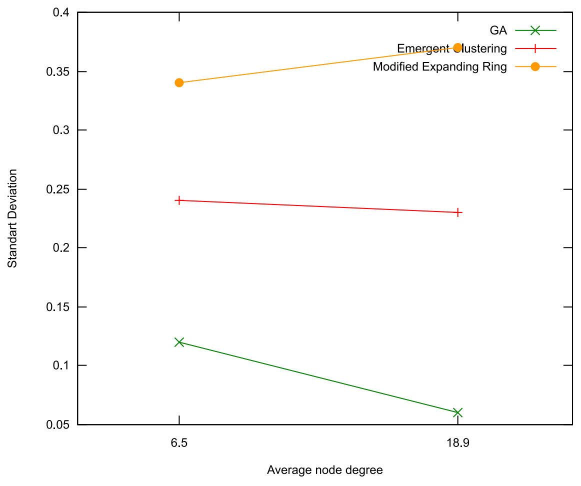


(a) Small Scenarios (b) Large Scenarios

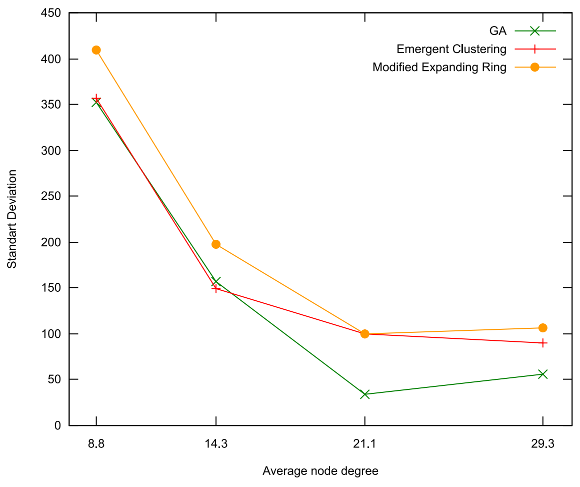
Figure 6.22: Overview of the normalized costs from the GA, Emergent Clustering and Modified Expanding Ring for all scenarios.



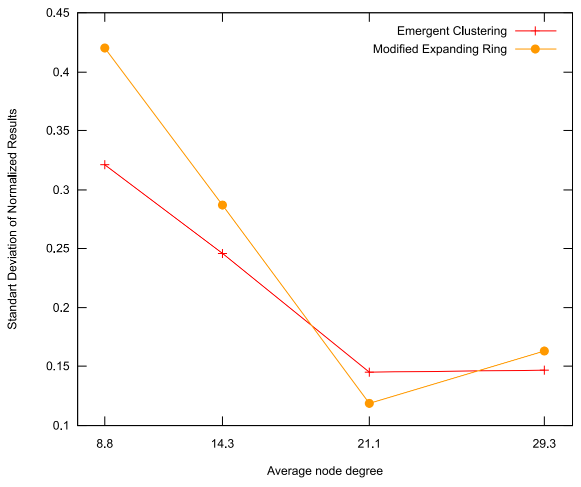
(a) Standart Deviation, small



(b) Standart Deviation of Normalized Results, small



(c) Standart Deviation, large



(d) Standart Deviation of Normalized Results, large

Figure 6.23: Standard deviation for the different heuristics with small and large problem size.

6.3.4.4 Statistical Dispersion

In this section, we will present and analyse the estimated standard deviation (s) of the outcomes of our experiments. In real sensor network deployment, if our result has a small statistical dispersion, we could better anticipate the performance of the clustering algorithm. With the statistical dispersion we can estimate how far from the average the result of a real deployment can be.

Figure 6.23 presents an overview of the estimated standard deviation from the mean of the absolute and the normalized results. In Figures 6.23(a) and 6.23(c) we can notice that the intrinsic standard deviation of the problem instances reduces with increasing density of nodes (looking at the optimal and GA results). This is an expected result, due to the fact that the means also reduces. Moreover, the Emerging Clustering and the Modified Expanding Ring demonstrate the same tendency of the reference solution. In the small scenarios (Figure 6.23(a)), it is possible to see that the deviation of the optimum solution and of the genetic algorithm are almost the same, which again increases our confidence of a very good GA design.

Figures 6.23(b) and 6.23(d) present the statistical dispersion when the results were normalized against the reference solution. We can see that the emergent clustering yields, in average, a smaller standard deviation than the expanding ring for the majority of cases. It is interesting to notice that, for large networks, a higher difference among sparse and dense scenarios has been encountered. This can be also visualized in Figure 6.16. We conclude from this result that networks with large number of very connected nodes present a larger number of possibilities for member selection, sometimes with redundancy (several nodes with similar quality or several paths to catch the same node by the clusterhead). This brings more homogeneous results, with a constant divergence from the optimal solution. Opposite to that, in sparse networks the quality of the result was very connected with the position of the clusterhead, because it does not have a large pool of nodes for membership selection. Therefore, for experiments where the clusterheads occasionally emerge in a not so optimal position, the resulting cost has a large distance to the optimal. In other experiments, where the clusterheads are, by random factor, better located, the cost is nearer to the optimal. In dense scenarios, an unfavorably positioned clusterhead has the chance of building a good cluster due to the presence of plenty of links and nodes.

The modified expanding ring has overcome the emergent clustering in the large-dense scenario. We presume that this has been partially caused by the "bridge" nodes presented in the previous section. Moreover, being indifferent to the fitness of a member candidate, the modified expanding ring produces much of the same kind of cluster over the areal. This is different from building very good clusters at the beginning and not so good at the end of the execution, as our approach. Therefore, a higher regularity can be expected from the modified expanding ring, and this reduces the variance of the normalized cost.

Summarizing, we conclude that for dense networks our algorithm presents a significantly lower standard deviation of the normalized results than for sparse networks. Moreover, the expanding ring and our emergent clustering have shown very low and similar standard deviations.

6.3.4.5 Distribution of Results

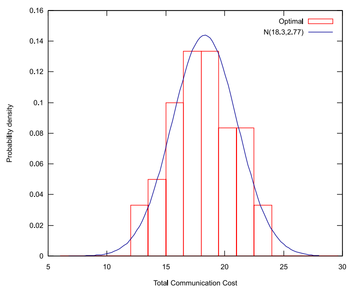
In this section, we will present the tabulated frequencies encountered in our experiments. The histograms presented here represent density estimations. In each histogram, the theoretic normal curve is also presented. It is calculated using the estimated parameters (estimated mean $\bar{x}$ and standard deviation s^2)

We present here two hypotheses to be checked by our frequency histograms. We argue that:

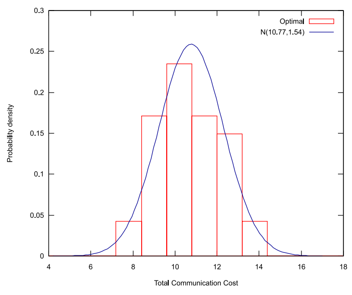
1. The optimal solution of the *minimum intracommunication-cost clustering* for experiments realized with constant parameters and different, randomly generated geometric random graphs as network topology follows a normal distribution.

2. When normalized against the reference solution, our Emergent Clustering and the Expanding Ring solution costs also follows a normal distribution.

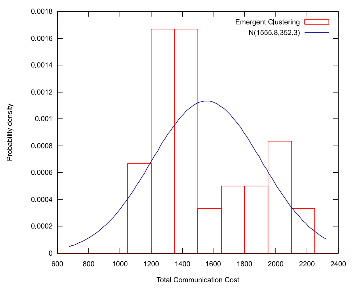
The total cost of an instance of the *minimum intracommunication-cost clustering* is composed of the sum of the individual clusters' cost. We can consider the cost of each cluster as a random variable that shares the same probability distribution (because one part of the geometric random graph is similar to other parts and in its complete area). The problem cost is given by the sum of those random variables, which means that, when we approach an infinite number of clusters, the distribution of the cost over several instances of the problem will converge towards a normal distribution. If the cost of each single cluster has expected value μ as well as standard deviation σ, and we have n clusters in the optimal solution, our population of experiments should approach a normal distribution $N(n\mu, n\sigma)$ when n goes to ∞.



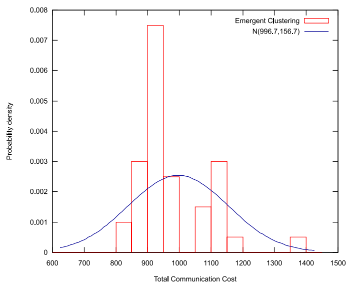
(a) Optimum, small-sparse



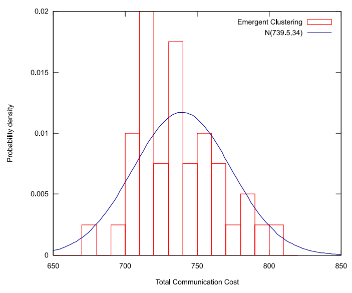
(b) Optimum, small-dense



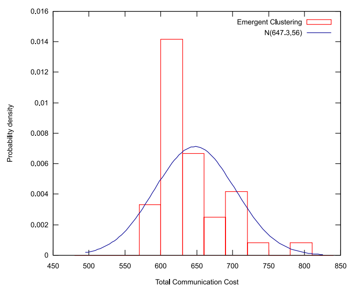
(c) GA, large-sparse



(d) GA, large-medium



(e) GA, large-dense



(f) GA, large-very-dense

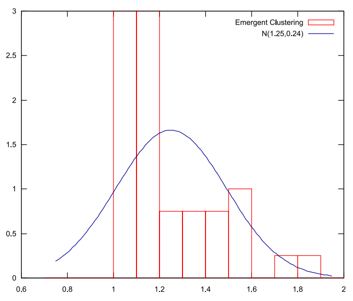
Figure 6.24: Density estimation for the optimal (GA) solutions of instances of the *minimum intracommunication-cost clustering* with geometric random graphs as input.

Figure 6.24 shows the resulting density histogram for our reference solutions with the theoretical normal curve. As we can see, the optimum solutions (Figure 6.24(a) and 6.24(b)) agree very well with the theoretically calculated normal curve, increasing our confidence in our first hypothesis. For the large networks, a similar result has been found, although some outrunners in the frequency distribution could be verified (Figures 6.24(c) to 6.24(f)). We guess that they could be some artifact introduced by the genetic algorithm. Moreover, for large networks, a larger number of samples may be necessary to improve the normal approximation.

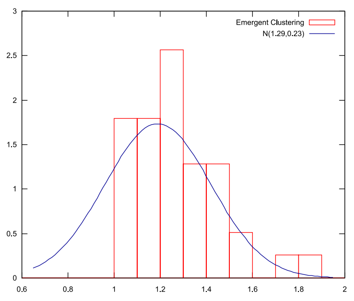
At this point we need to concentrate on the results of the emergent clustering and modified expanding ring heuristics. Figure 6.25 shows the distribution of the normalized results for all experiment scenarios. The fact that the results approximate a normal distribution can be seen very clearly in the large scenarios (Figure 6.25(c) to 6.25(f)). In the small ones, we suppose that the outrunners are caused by the fact that there are fewer problems with overlapping, competing clusters than in large ones, bringing a slightly higher frequency in good (small cost) results. The lack of nodes reduces sometimes the possibility for even more reduced cost, which brings a small dis-balance in the frequencies encountered in the experiments (Figures 6.25(a) and 6.25(b)). Nevertheless, all results resemble, with exception of a small number of outrunners, the normal curve.

For the modified expanding ring heuristic, similar results have been found. They are showed in Figure 6.26. It is important to remark that the confidence interval presented in earlier sections was based on the assumption of this section, i.e., our results converge to a normal distribution. This has been shown here.

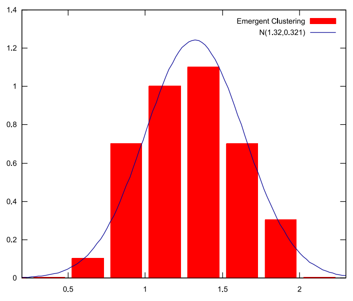
We will now analyze the cumulative histogram of our experiments to see how much of our outcomes have lower cost. Figure 6.27 presents the cumulative density for the large scenarios. The normalized results are used here. In the Figure 6.27(a), the results for the emergent clustering are depicted, and in Figure 6.27(b) the results of the modified expanding ring. For both heuristics, the



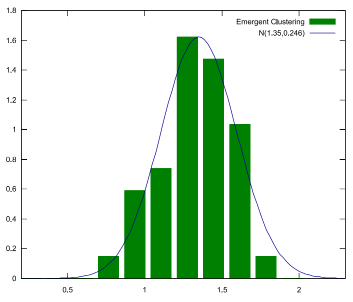
(a) Emergent Clustering, small-sparse



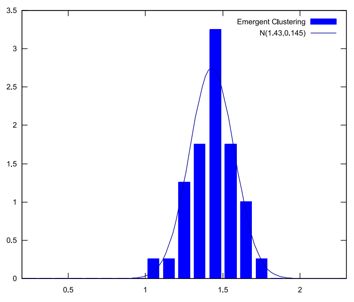
(b) Emergent Clustering, small-dense



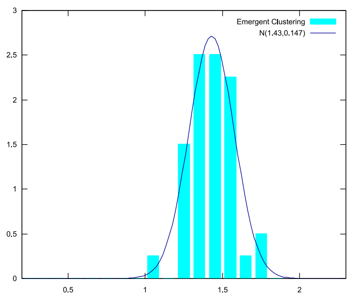
(c) Emergent Clustering, large-sparse



(d) Emergent Clustering, large-medium

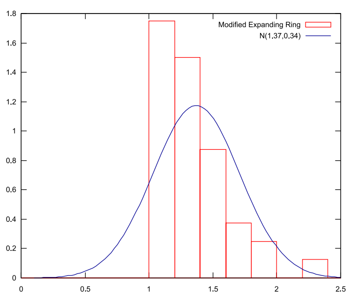


(e) Emergent Clustering, large-dense

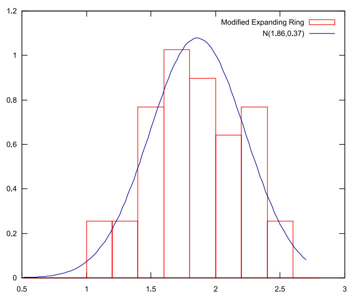


(f) Emergent Clustering, large-very-dense

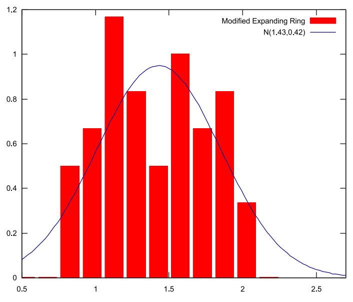
Figure 6.25: Density estimation for the emergent clustering solutions of instances of the *minumum-intracommunication clustering* normalized against the reference solutions.



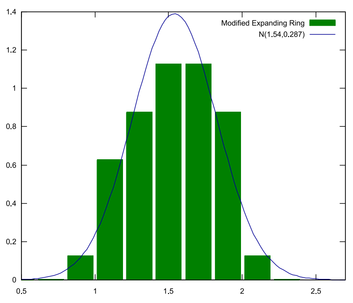
(a) Modified Expanding Ring, small-sparse



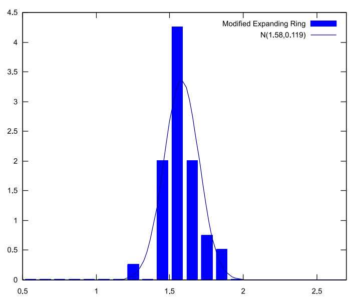
(b) Modified Expanding Ring, small-dense



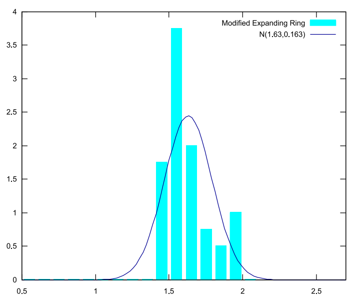
(c) Modified Expanding Ring, large-sparse



(d) Modified Expanding Ring, large-medium

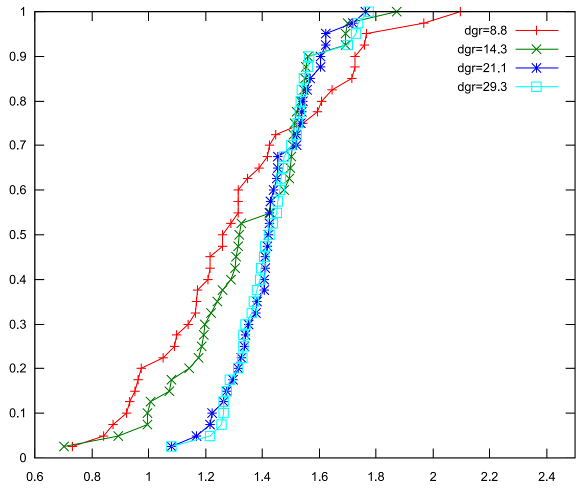


(e) Modified Expanding Ring, large-dense

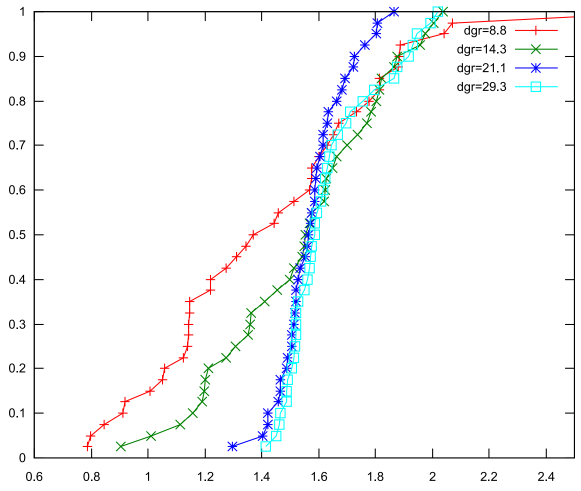


(f) Modified Expanding Ring, large-very-dense

Figure 6.26: Density estimation for the modified expanding ring solutions of instances of the *minumum-intracommunication clustering* normalized against the reference solutions.



(a) Emergent Clustering, large



(b) Modified Expanding Ring, large

Figure 6.27: Cumulative histogram of normalized results for large scenarios.

slope of the cumulative histogram increases with the density of nodes of the scenario. An exception can be noticed for the large-dense and large-very-dense scenarios, where similar slopes are found.

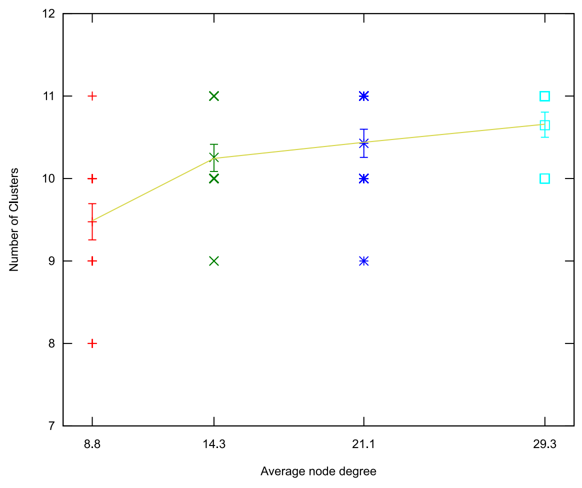
The emergent clustering presents the majority of the results (90%) below 1.8 times the reference, for any node density. Moreover, for medium and dense networks, the result is even better. For the expanding ring, 90% of the results were below 2.1 times the reference solution. Although the average normalized cost of the sparse scenarios was smaller than for dense scenarios, the smaller variance of the latter one constraints the results in a narrower range. This brings about that, when we analyze the majority of cases (90%) for the dense (and very dense) scenarios, they are below 1.63 times the optimal solution, while for the sparse scenarios they are below 1.8 times the optimal solution.

From the exposed diagrams, we can conclude that our emergent clustering presents, for any density, a better cumulative histogram than the modified expanding ring. This means for a given cost limit (e.g. 1.8 times the optimal), more outcomes of the emergent clustering lies below the limit than the modified expanding ring. Moreover, almost all results of the emergent clustering were relatively near the reference solution. This was achieved using a distributed heuristic without global information and very low computational cost.

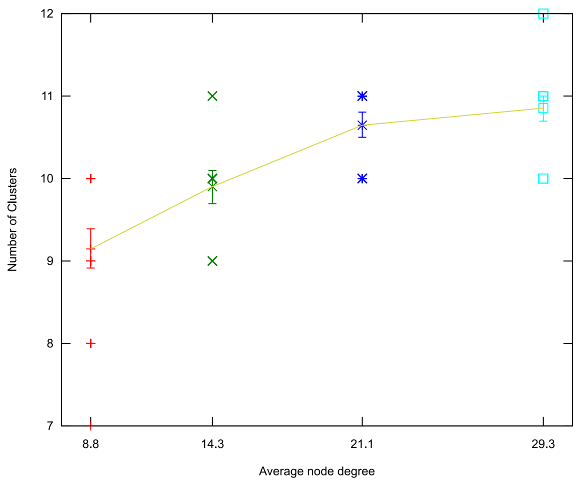
6.3.4.6 Number of Clusters

In this subsection, we analyze the number of clusters formed in the simulated heuristics for large scenarios. Figure 6.28 presents the achieved results. The confidence interval is also presented.

For both heuristics, the number of clusters increases with the node density. This can be explained by the fact that sparse scenarios present less connection possibilities than big ones. The situation presented in Figure 6.15, which brings a higher optimal solution cost, acts also in both heuristics, but here it reduces the number of clusters and increases the cost. As the figure shows, when nodes 1, 2 and 3 are included in a three-nodes cluster (suppose $q = 3$), the other nodes cannot build an autonomous cluster. In the optimal solution, this would not happen, because the whole network would build a big cluster. A one-cluster solution is more expensive than a two-clusters one (when nodes 4, 1, 2 form one cluster whereas the rest forms another one). Nevertheless, our clusterhead selection method has a greedy nature as well as the selection of members. This means that, with high probability, nodes 1,



(a) Emergent Clustering, large



(b) Modified Expanding Ring, large

Figure 6.28: Number of clusters achieved for different node densities.

2 and 3 would form one cluster, and, at end of the heuristic, due to the fact that the remaining nodes cannot form another cluster, they will be included in this one, yielding a high cost solution with less number of clusters than desirable.

This effect has a higher probability to appear in sparse networks than in dense ones. “Islands” consisting of nodes that are not able to form a cluster must be included in some existing one, which leads to a smaller number of clusters in sparse networks, which results in increasing total costs. Both algorithm are susceptible in the sparse scenario. Due to the superior membership selection of the emergent clustering, it has a superior performance with respect to the number of clusters in such scenarios.

It is important to state here that a higher number of clusters is a desirable property and means that the clusters are nearer the given bound q. A higher number of clusters, in several cases, brings also a smaller cost (as defined in our optimization problem). Nevertheless, it is also possible to have a solution with higher number of clusters and, at the same time, higher cost than another. This happens when we compare the emergent clustering with the modified expanding ring in dense scenarios.

For dense (and very dense) experiments, both algorithms present a much better performance concerning the number of clusters. Nevertheless, the modified expanding ring presents a slightly higher average of the number of clusters. The fact that the emergent clustering selects the better neighbors to belong to the cluster increases the chance of appearance of such “islands” even in the dense scenarios. The modified expanding ring produces more “mixed” clusters, where nodes enclosed by members of a given cluster may be not included in it and can serve as “bridge” between such islands, increasing the number of successfully formed clusters. This is also illustrated in Figure 6.20.

Nevertheless, it is important to state here, that even having a higher number of clusters, due to the bad quality of such clusters, the expanding ring incurs a higher communication cost than the emergent clustering solution with less clusters.

6.3.4.7 Number of Messages

In this section, we will analyze the necessary number of messages to decompose the given network in a set of clusters. We assume that the energy spent by the algorithm to construct the clusters is mainly

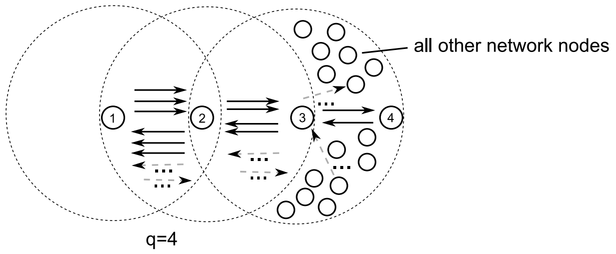


Figure 6.29: Worst case clustering scenario when considering the number of messages ($q = 4$).

influenced by the number of messages used.

For $q \ll n$, where n is the number of nodes of the network, the worst case message complexity of constructing a single cluster happens when all nodes receive the call for members messages at the last round: $m = O(qn)$, all nodes of the network will respond to that message (contributes to the n in the complexity). Moreover, the messages must be routed through the $q-1$ members of the cluster until arriving the clusterhead (contributes to the q factor in the complexity).

Figure 6.29 shows an example of scenario where the worst case message complexity is verified. The node 1 emerges as clusterhead and starts the membership selection phase. In each round, just one new member can be achieved by the call for members message, therefore a single node is incorporated in the cluster. When the cluster has $q-1$ nodes and it starts the last call for members message (node 3, in the example), all other nodes of the network are reached and respond to that call. Naturally, one node of the responding ones will be integrated in the cluster and the other ones will be refused.

The necessary number of messages in the worst case is:

$$m = 2 \cdot \sum_{i=1}^{q} (q-i) + 2 \cdot (n-q) \cdot (q-1) = (q^2 - q) + 2 \cdot (n-q) \cdot (q-1) = O(nq) \tag{6.12}$$

The worst case complexity for construction one cluster with the expanding ring is $O(n)$, because when receiving all messages from all nodes of the network that aren't in the cluster in the last round, instead of routing individually each one to the clusterhead, they are summarized in one single (large) message. Due to the waiting time principle of our heuristic, this is not possible. Nevertheless, we discuss in the conclusion an alternative to reduce the worst case complexity of the emergent clustering. The proof of the worst case message complexity of the expanding ring is: in any round in which the cluster size is less than q, the total number of messages exchanges is polynomial in q. In the worst case, all nodes in the network may receive messages in the last round [77]. Therefore, the worst-case message complexity is $O(n)$ since $q \ll n$.

Now, we will consider the results of our experiments.

Figure 6.30 presents the total number of messages used, in average, for each large scenario simulation. From the results, we can see that a similar number of messages was necessary in all scenarios. Nevertheless, the sparse scenarios needs a slightly higher number of messages than the other ones. This can be explained by the fact that, in the sparse scenario, the degree of the nodes is below the minimum cluster size. This means that for the average case it would be necessary to search for members within more hops than for the other cases. We can see in the equation 6.12 that a quadratic component (q^2) dominates when every round has few nodes reached. In sparse scenarios, the average cluster diameter is larger and the quadratic exponent has a higher effect on the number of messages.

This fact is confirmed by Figure 6.31. In this experiment, we test how the cluster size influences the necessary number of messages. For the same scenario, we make 20 simulations for each selected

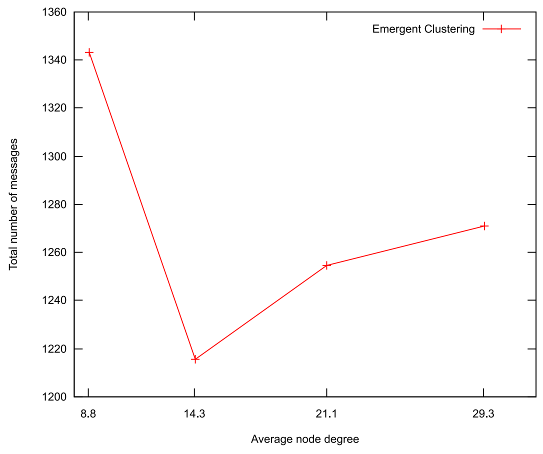


Figure 6.30: Total number of messages for large scenarios

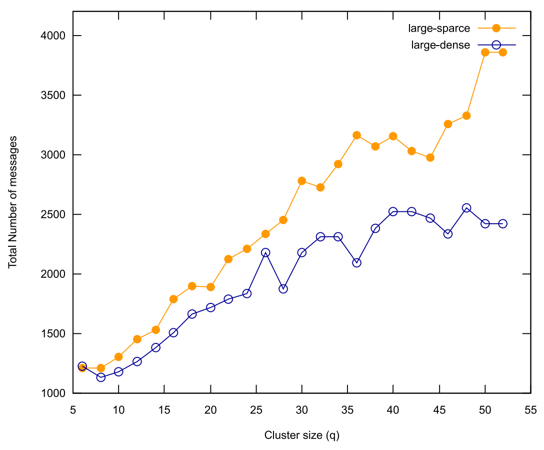


Figure 6.31: Number of Messages with different cluster sizes.

cluster size. We select a range from $q = 6$ to $q = 52$ for minimum cluster size. We simulate the large-sparse and large-dense scenarios.

As can be seen in the figure, the difference between the necessary number of messages in the sparse scenario and in the dense one increases with increasing cluster size. This happens because in sparse scenarios more hops are necessary to catch the members of the cluster. More hops means more messages per included node, this means, sparse scenarios has a higher quadratic component in the number of messages than dense ones (more hops are necessary). More density means less hops, therefore less messages per included member are necessary.

In Figure 6.30, we can see that after achieving the minimum amount of messages in the large-medium scenario, the necessary amount of messages starts to increase for more dense ones. Since in the medium, dense and very dense scenarios the average number of neighbors is higher than the necessary number of members in the cluster, they have similar cluster diameters. However, in the dense scenarios, an additional effect can be observed: each call for members arrives at much more nodes than necessary for the cluster formation. This, in some way, resembles the value n in the worst-case complexity. As the necessary number of members has been completed, several unnecessary replies will be sent to the clusterhead by the nodes that have been requested (*call_members*) but are answering later than the selected members. This again increases the number of necessary messages.

In Figures 6.32 and 6.33, the cumulative histogram and the standard histogram of a single simulation run are shown, respectively. The results show that a large exchange of messages happened in the beginning of the simulation, when several clusterheads emerged from the transition functions. Moreover, at the beginning, more nodes are free and will respond to call for members messages. Another small peak can be found in the end of the maximum clustering time [1]. This peak is relative to the enforce phase, at the end, to include nodes that were unable to build a cluster in the existing ones.

[1]The maximum clustering time is a parameter of the algorithm that has relation with the individual clustering time t_c and the collision probability described in the previous chapter.

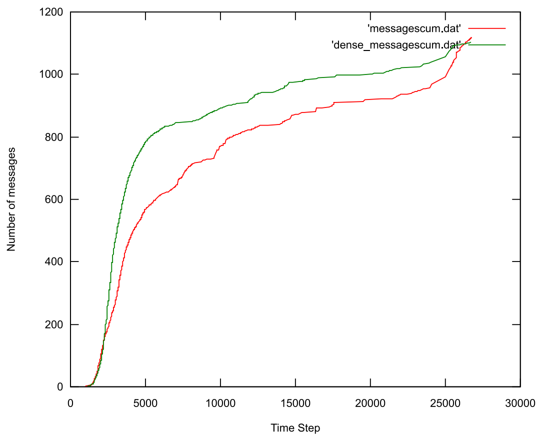


Figure 6.32: Cumulative number of messages for a single simulation run of large-sparse and large-dense scenarios.

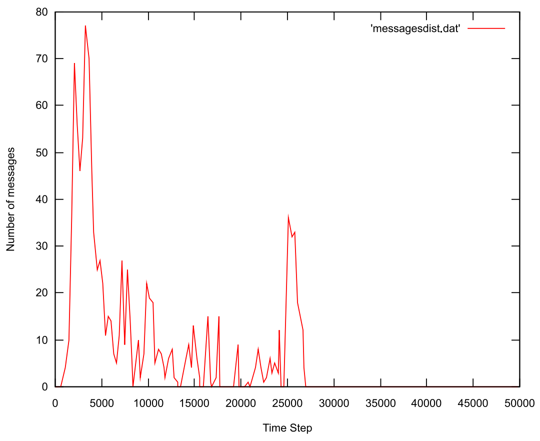


Figure 6.33: Distribution of the messages through the simulation time for a single simulation run (large-sparse).

6.4 Service Distribution Simulation

In this section, we will present the simulation results of our basic and extended service distribution heuristics.

6.4.1 Assumptions

For our simulation, the following assumptions are used:

- Each node has a transmitter with fixed power. Therefore, the maximal reachability is fixed.
- Links are bidirectional. In the reality, this can be achieved just by ignoring unidirectional links.
- The Friis Free Space propagation model for isotropic point source in an ideal propagation medium is used to calculate the received signal strength indication (RSSI).
- RSSI was the only metric used in the virtual distance.
- Every node has a unit of energy.
- Each node has enough resources for a single service and a single task.
- Tasks request different, randomly selected services.
- The bandwidths needed in the different communications were randomly selected.

6.4.2 Simulation Scenarios

For the evaluation of the service distribution, we also use different problem sizes. In the first group of experiments, we have a field of $80m \times 60m$ where 10 nodes are randomly deployed with independent uniform probability. We have 8 services, with 6 tasks requesting services. The small scenarios were selected in order to enable the calculation of the optimal solution.

Scenario Name	Field Size (m^2)	Number of Nodes	Radio Range	Connection Probability	Node density	Average Degree (Theoretic)	Num. Services, Requesters
			Small Scenarios				
small-sparse-sd	80x60	10	28	0.002	0.9	5.13	8, 6
small-dense-sd	80x60	10	43	0.002	1	12.1	8, 6
			Large Scenarios				
large-sparse-sd	102x77	100	13	0.013	0.9	6.7	20, 40

Table 6.2: Overview of the different simulation scenarios.

For large scenarios, it is not possible to calculate the optimal (reference) solution of our discrete optimization problem due to its computational complexity. Therefore, we cannot compare the results with a reference solution. Nevertheless, we decided to make an example simulation of a large scenario to show that its behavior is similar to small instances. We use an areal of $102m \times 77m$ where 100 nodes were deployed. Moreover, 20 services are serving 40 different tasks.

The network graph was generated using the Friis free space propagation model with a fixed maximal reachability. We will call it “radio range”. Any two nodes inside this range are able to communicate. In this case, a link exists.

For the generation of the task/service graph for each task, a random number of services was selected. The tasks request those services with a random bandwidth requirement (normalized). Moreover, each node has the following memory restriction: just one task and one service can be hosted by a node. This means that e.g. two services cannot be placed in the same node. The overview of all simulated scenarios can be observed in Table 6.2.

An additional important point is that we are using the Dijkstra’s shortest path algorithm to find the route between the requesters and the services of our network.

6.4.3 Algorithms under Evaluation

The presented scenarios were evaluated using different algorithms. For the small scenario, the optimum solution was calculated using a branch-and-bound algorithm. For all scenarios, our basic and the extended ant-based service distribution heuristic were simulated. We allow swap operations of services when the settlement phase does not found a node with enough resources for the migration.

Moreover, we decided to calculate the cost of a completely random assignment, i.e., the tasks and services are randomly distributed among the nodes of the network.

6.4.4 Results

We executed 40 experiments for each scenario presented in Table 6.2. In the next sections, we will present and analyze the results of the experiments.

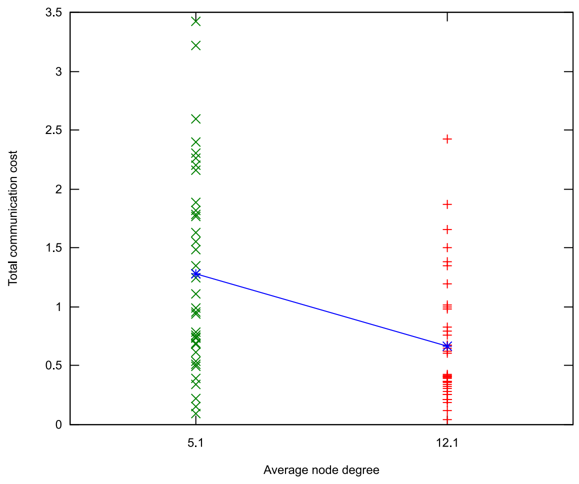


Figure 6.34: Optimal assignment cost of sparse and dense scenarios.

6.4.4.1 Optimal Assignment Cost

In this section, we will analyze the results achieved with the optimal cost assignment. Figure 6.34 presents the optimal service assignment cost for the small-sparse-sd and small-large-sd scenarios[2]. As expected, denser scenarios present a smaller assignment cost. This can be explained by the fact that better links (lower cost) are available for the communication between the tasks and services, reducing the total cost. Moreover, due to the higher amount of neighbors (higher node degree), assignments that yield a high amount of costs in sparse environments may be attractive in dense environments because of the existence of several new links.

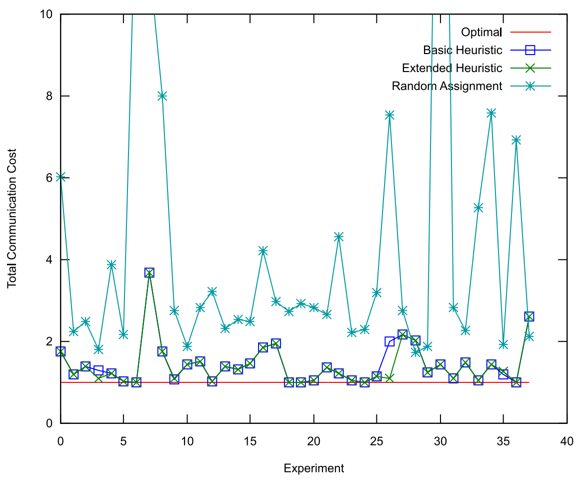
6.4.4.2 Experiment Results

This section presents the outcome of our 40 experiments for the presented scenarios. In Figure 6.35, the results for our three scenarios are depicted. For the small scenarios, each result is normalized against the optimal assignment. For the large scenario, we present the nominal result.

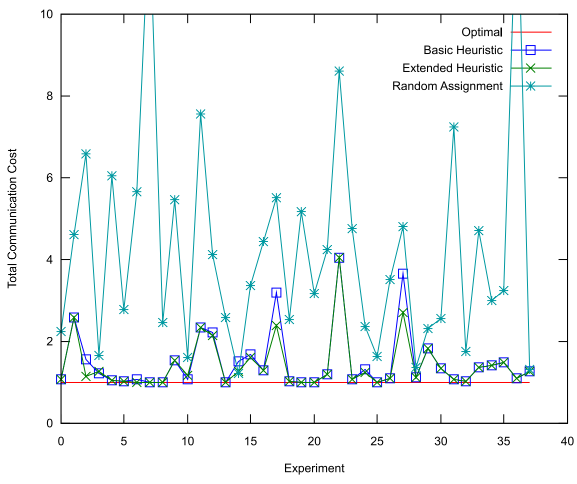
As we can see in Figures 6.35(a) and 6.35(b), our heuristic found the optimal solution in several cases. Moreover, for the vast majority of cases, the heuristic has a much better performance than the random initial assignment. The extended heuristic and the basic one have also a very similar behavior, nevertheless, for some experiments, the extended one has a much better performance than the basic. The reasons for these outcomes will be discussed further.

Figure 6.35(c) shows the results for a large scenario. Due to the fact that we do not have, for large scenarios, a reference approach, it is not possible to make statements about the absolute performance of the algorithms. Nevertheless, it is possible to notice that the heuristics could find a much better cost than the initial random assignment. Moreover, the behavior of the extended and basic heuristics are similar to the observed in the small experiments.

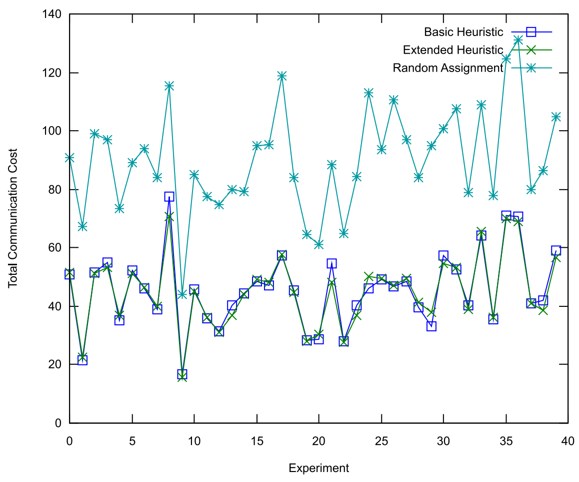
[2]For the service distributions, the terms total communication cost (presented in the figure) and assignment cost have the same meaning.



(a) small-sparse-sd, normalized



(b) small-dense-sd, normalized



(c) large-sparse-sd

Figure 6.35: Results of the realized experiments.

6.4.4.3 Heuristics’ Assignment Costs

In this section, the mean value of the achieved costs for each heuristic for all scenarios will be presented. The cost for the absolute assignment of our test scenario is shown in the Figure 6.36. The random assignments and basic and extended heuristic assignments have the same tendency of the optimum: for sparse networks, they deliver always a assignment with higher cost. This is expected due to the relation between the assignment cost and the link costs, and for sparse environments, the average link cost increases.

In Figure 6.36(d), the different costs are shown together for the small scenarios. As it can be seen in the picture, our basic and extended heuristics have a good performance, not far from the optimal solution. The basic and extended heuristics have a very similar performance. We will discuss about the reasons and the performance difference further.

In Figure 6.36(e), the results of our large scenario are depicted. It is possible to see that they are very similar to the small scenario, improving our confidence that the heuristics could find good solutions for small as well as large scenarios.

Figure 6.37 shows the normalized results for the small scenarios. The optimal assignment is used as reference. It is possible to notice that, for all cases, a very small difference could be verified for sparse and dense scenarios. The basic heuristic has an average cost of 1.44 times the optimal cost for sparse environments and 1.5 for dense ones. The extended heuristic shows a small improvement: 1.41 for sparse and 1.43 for dense scenarios. This means that the cost of the basic heuristic was about 2% higher in sparse scenarios and 5% in dense scenarios. Similar behavior has been found in the large scenario.

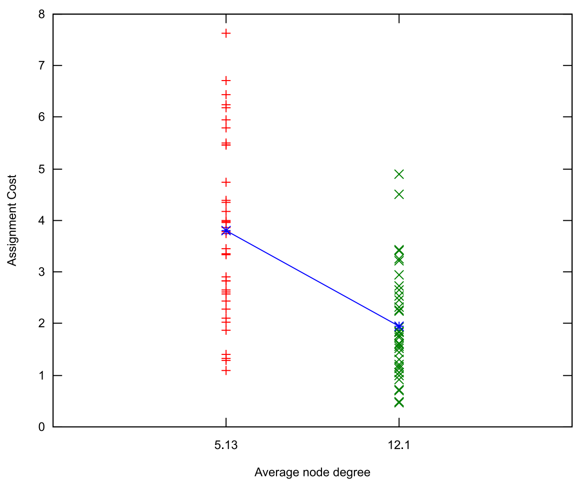
As it could be seen in the Figures 6.35(a) and 6.35(b), the basic and extended heuristic, for several experiments, could find solutions with very similar costs and for some experiments, the extended overcame the basic one. For the experiments where the results were similar, we suppose that there aren’t flow correlations that helps the heuristic behavior. For the experiments where the extended heuristic has a much better performance, correlations could be found and a better service migration was realized.

The question that arises from the results is why correlations were not so common? We guess that the reason was the selected routing algorithm together with the influence of the Friis Free Space Model in our link metric. Because of the exponential path loss, nodes near to each other have a greater advantage in the signal strength than others with a small higher physical distance. The link metric reflects very much this exponential path loss due to the fact that we are, for our simulations of service distribution, relying strongly on the signal strength to calculate the link metric. The Dijkstra’s shortest path algorithm always selects the shortest path between any two nodes and does not try to divide the load among the existing link channels. Further we are also not taking into account the link utilization (and possible congestion). Together, such facts act in a way that effectively just a small subset of links is used for all communications. A kind of backbone emerges in the network. This leaves less space for our flow correlations.

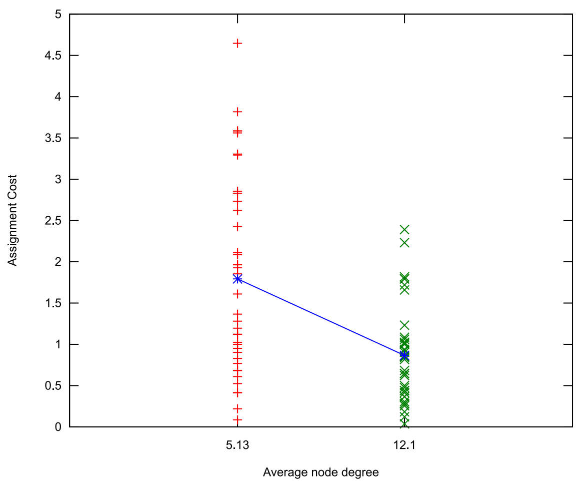
We suppose that, in real scenarios, where the link metric has a more irregular nature and routing mechanisms that divide the transmission effort among different routes, the extended heuristic will increase its performance in relation to the basic one.

6.4.4.4 Cumulative Distribution

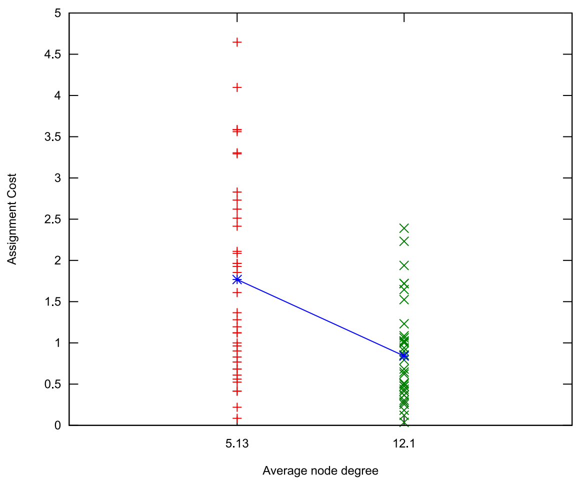
Figure 6.38 shows the cumulative distribution of the normalized results for the small scenarios. Both basic and extended heuristic have a similar behavior, and also sparse and dense scenarios brought



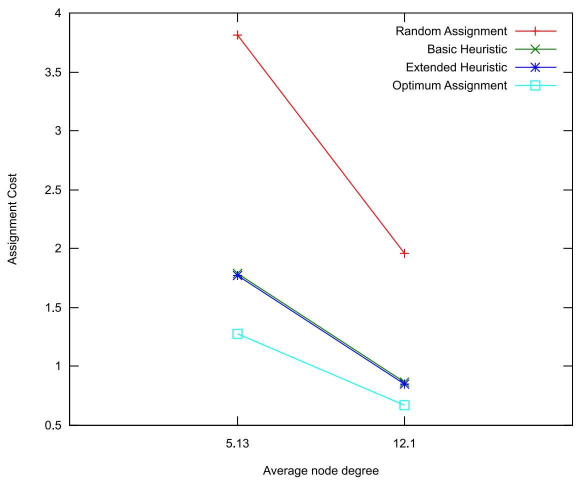
(a) Random Assignment, small



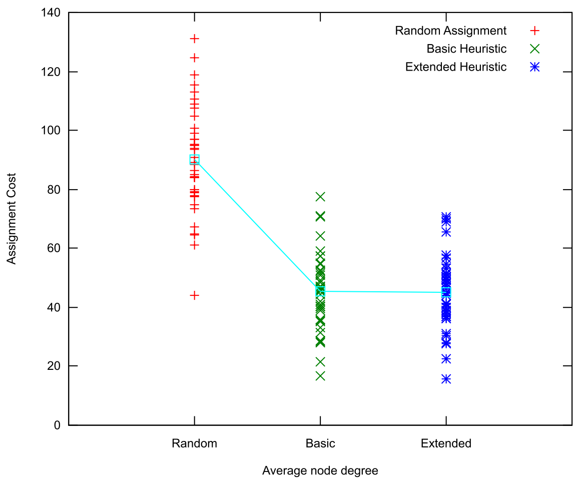
(b) Basic Heuristic, small



(c) Extended Heuristic, small

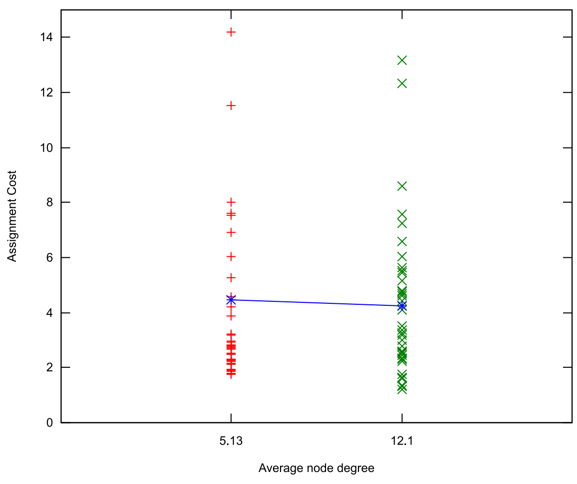


(d) All, small

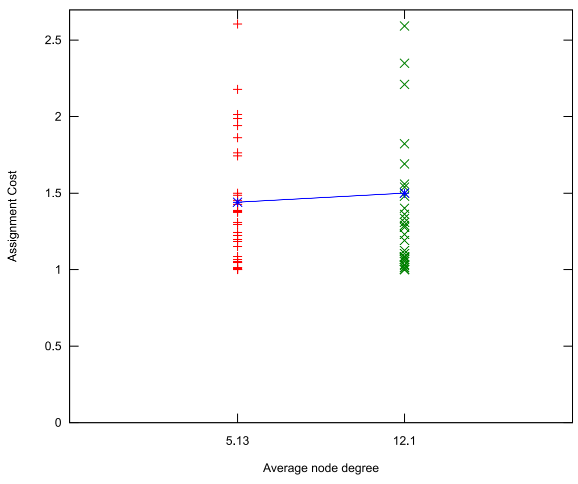


(e) All, large

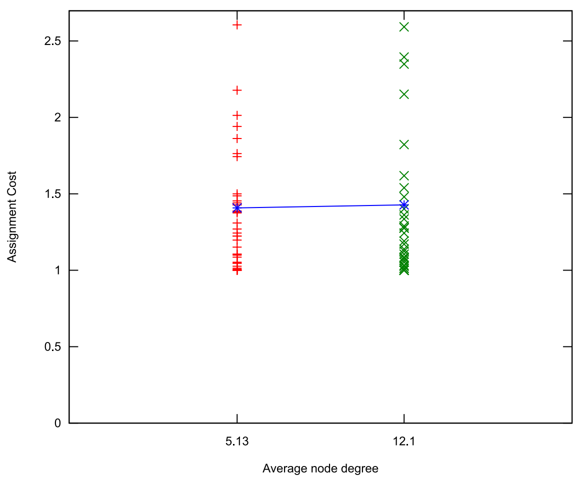
Figure 6.36: Absolute assignment costs for the different heuristics with small and large problem size.



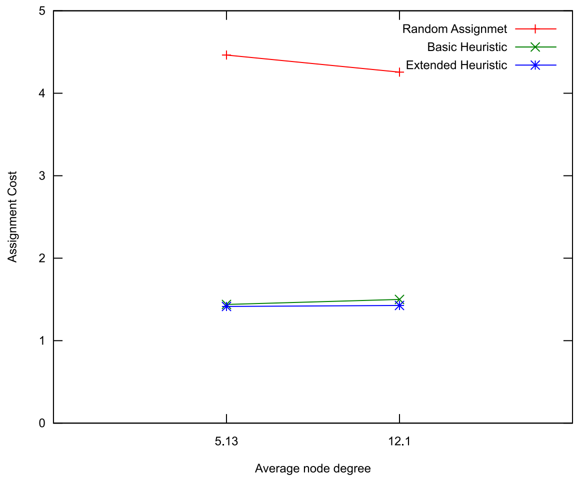
(a) Random Assignment



(b) Basic Heuristic



(c) Extended Heuristic



(d) All

Figure 6.37: Normalized assignment costs for the different heuristics for the small scenarios.

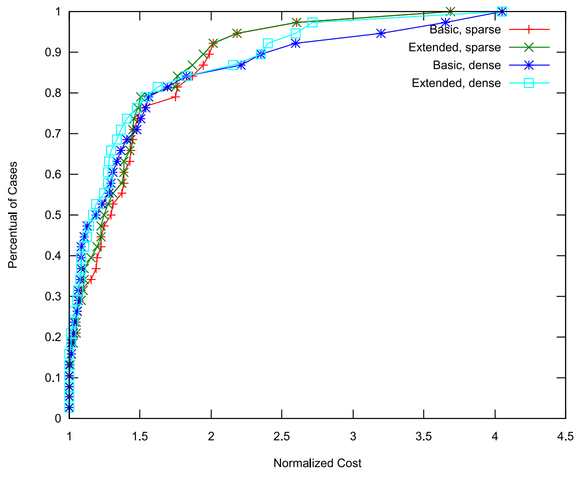


Figure 6.38: Cumulative distribution of the cases for small scenarios.

similar results. However, small differences can be noticed. It is possible to see that the extended heuristic outstands the basic one for both sparse and dense scenarios. Moreover, it was possible to notice that in sparse scenarios, more results were below 2.5 times the optimal value than for dense scenarios.

From the figure it is possible to see that for a majority of cases (83%) the results were below 2 times the optimal results, for both heuristics in all scenarios.

6.4.4.5 Distance Between Requesters and Providers

In Figure 6.39, the total distance between the requesters and providers, measured in hop count, normalized in relation to the optimal solution is plotted. It is important to highlight that we count the hops over the path used by the communication between the requesters and providers (shortest path calculated using the link metric).

The results are similar to the cost measured by means of traffic and link metric. Our heuristics have a slightly higher cost than the optimal solution.

6.4.4.6 Number of Migrations

Figure 6.40 presents the number of migrations performed by the basic and extended heuristics, for allowed_h=1 and 2, i.e, when single and two-hops migrations are allowed. We notice that similar numbers of migrations were necessary for the basic and extended heuristics. This reflects also the similar performance of both methods. Moreover, if the services are allowed to jump a higher number of hops in a single migration, as expected, less migrations are necessary. The experiments with allowed_h=1 have about 1.28 more migrations than when allowed_h=2 was used. This value is far less then the maximum theoretical value of two because the potential pheromone (used in multi-hop migrations) may lead, sometimes, to false decisions. Moreover, when the final destination of a migrating service is an odd number of hops away, turning allowed_h=2 does not half the necessary number of migrations used to achieve the destination.

We suppose, for bigger networks, that the performance of the algorithm will increase when larger migrations are allowed. Nevertheless, in dynamic networks as well as in systems where services are using other services, multi-hop migrations may bring an instability to the system.

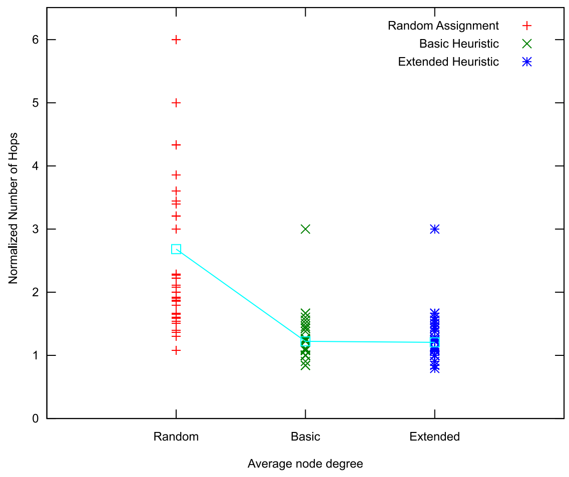


Figure 6.39: Normalized total distance measured in hops from requesters to services, small-sparse-sd scenario.

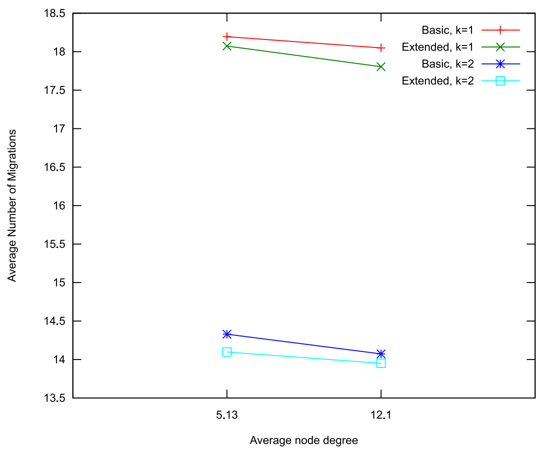


Figure 6.40: Number of migrations for different allowed_h=k.

An additional comment is that due to the fact that the network was slighly loaded with services, the swap operation was used with certain frequency. With less loaded networks, we expect a smaller number of migrations.

6.5 Discussion

The simulation results presented in this chapter focus on key aspects of our emergent clustering and service distribution heuristics. For the purpose of increasing the relevance of these results, we use several scenarios with different node densities and also different parameters in the algorithm, as cluster size and number of servers and requesters. Our statistical examinations were used to prove the significance of the results.

The emergent clustering heuristic has shown a very good performance for scenarios with different node densities. The average cost was at most 1.44 times the reference approach for all simulated scenarios. In addition, small standard deviations could also be verified. Our emergent clustering outperforms the modified expanding ring in almost all simulated scenarios. These are very good results that were achieved by means of local iterations between the WSN nodes.

Nevertheless, there is a cost for our algorithm: in the worst case, the necessary number of messages to build a single cluster is $O(qn)$, where n is the number of nodes in the network. The expanding ring has a slightly smaller complexity $O(n)$. Nevertheless, there are algorithms, like the *rapid*, where the message complexity to construct a single cluster is $O(q)$. This low message complexity comes with a drawback: the *rapid* algorithm may produce clusters very far from the given bound q (smaller), because members without free available neighbors to include in the cluster simply do not re-allocate their residual budget. The *persistent* algorithm does that and the worst case complexity is $O(q^2)$. The expanding ring and our emergent clustering have a worst case complexity of forming one cluster depending on n due to their broadcast nature: when searching for members, all neighboring nodes are approached, fact that can lead to this high message complexity.

However, in more realistic scenarios, much lower message complexity can be expected. We accept a relatively high message complexity during the cluster construction because it will lead to a smaller communication cost during the operational phase of the NanoOS. The idea is that, after the cluster is completed, services and requesters will communicate in an intensive way within a cluster. The initial set-up cost will be amortized in the operational phase. Moreover, other algorithms like *rapid* and *persistent* do not guarantee that the bound (q) will be achieved, being not appropriate for our purpose.

Our service distribution heuristics have demonstrated a very good performance when compared to the optimal one. In all tested scenarios, the average achieved cost was at most 1.5 times the optimal one. This was achieved with a low computational cost. Nevertheless, the extended heuristic has shown just a slightly better performance than the basic one.

We suppose that this occurs due to the small subset of links that are used to route almost all packets in the network. Due to the fact that our heuristic is dependent on the underlying routing protocol, and, as already said, we are using the optimal single-path routing Dijkstra algorithm, the selection of the communication paths has a high correlation with the exponent of the Friis free space propagation model, since the link metric reflects it strongly in the service distribution experiments (where the exponent two was used). The higher the exponent, the less diverse are the routing paths, i.e., a “backbone” is formed in the network and all packets are routed through it. For the extended heuristic, this is not very favorable, since then the chance of the existence of correlated flows is reduced.

Due to this fact, for a large number of real applications, the basic heuristic yields adequate results and the extra computational effort necessary for the extended heuristic may not be compensated. However, in real environments, where the link metric does not follow in a regular way the Friis path loss, and different routing mechanisms can be used, the extended heuristic may bring better results and it can be used with advantage.

Such a more realistic simulation where the environment is not so idealized is a point for further work. More realistic physical models, wireless devices and network stack should be used. Further, dynamic topologies must be tested. We aim to use the reference point group mobility [63]. In addition, how each parameter of the heuristics influences their behavior, for the emergent clustering and for the service distribution, should be also studied. Tuning both heuristics may improve the good results encountered in our simulations.

A further future work is to test our service distribution heuristics with different routing protocols. Because our basic heuristic is sensible to instabilities in the routing algorithm when selecting routes at different points in time (with the same topology), we expect a better performance of our extended heuristic compared to the basic one for certain routing protocols.

It is important to highlight that some aspects were not tested in the experiments. For example, in the clustering heuristic, nodes with different energy levels may yield different clusterhead election, which influence the behavior of our algorithm. Since the expanding ring (among others) does not take in account the energy in the clusterhead election phase, we believe that we can achieve a longer clusterhead longevity with our emergent clustering. Nevertheless, rotating clusterhead heuristics have an advantage in this item when compared with our. We plan to introduce clusterheads rotation, similar to the described in our clustering heuristic for dynamic networks. Moreover, nodes with different amount of resources should be also introduced in our simulations.

An additional important further work is the complete simulation and analysis of our dynamic clustering algorithm. We have implemented it using the Shox network simulator and a working proof of concept already exists. It is able to decompose a dynamic network in a set of clusters which comply with the minimum bound q. Moreover, it showed robustness against moderate topology changes. Nevertheless, numeric results to allow comparison with the emergent clustering for “quasi-static” networks are missing and they present a further step in this work.

Chapter 7

Conclusion

Wireless sensor networks enable a wide range of new applications. The system software of a sensor should be flexible and powerful to enable the easy development of different kinds of WSN applications.

This thesis presents the architecture of an innovative OS for sensor nodes, which integrates local hardware management with abstractions for enabling cooperative processing among geographically distributed nodes. NanoOS supports generic, complex distributed in-network processing. Due to the resource constrained hardware, it is not possible to provide all necessary functionality at the node level, therefore, the network as a whole should offer an aggregated capability and functionality.

Although several middleware approaches present different kinds of distributed processing, they lack flexibility, being adequate for a certain type of application. Normally, they envision typical WSN applications like processing queries coming from a user, managing sensor events or coordinating data fusion. Many of them are targeted towards data-centric applications or in-network processing related to such applications (e.g. data fusion). The database middleware hides from the programmer the complications of distributed programming, enabling to program the sensor network as a whole. Nevertheless, there are only pre-defined ways to process the data. In our approach, a much more flexible way of programming is offered.

The virtual machines try to remedy the limited flexibility at the expense of increasing the programmer's responsibility. But different from our approach, the distribution of the distributed algorithm is controlled also by the programmer, i.e., the replication or migration of an executing segment must be explicitly done. For several applications, this programming model pose a challenge to the developer.

Our OS offers support to the distributed processing in the conventional sense, i.e., distributed algorithms can be implemented using a server-client paradigm with automatic service instantiation, discovery and migration. Although we are providing a client-server paradigm, typical sensor applications like data fusion can also be easily implemented within our OS. We want to achieve a localized processing, in order to make the sensor network more autonomous, without relaying on an external access point to gather all the data at all times. Moreover, the cooperation between nodes enables the calculation of a more abstract system level decision from the raw sensor data. Instead of sending low-level sensing information to the gateway (or an other interested entity), the high-level processed data can be sent, reducing the amount of communication necessary. The distributed processing by means of the application and OS services enables a higher functionality in the sensor network. Nevertheless, the cost for this is the increased amount of interaction among modules residing at nearby nodes.

A central point of the OS is the automatic placement of the services in order to reduce the quantity of communication of the system, saving the scarce energy resource. For that, we present in this thesis a basic and an extended heuristic for controlling the migration of the mobile services. Since the processor assignment problem is NP-complete, our heuristics are best-effort. Nevertheless, they are distributed and only using local interactions to achieve the global goal.

Both versions of the heuristic are based on stigmergetic communication: when requesters communicate with the services, they leave pheromone on the used network path. When a service decides to migrate, the pheromone trails guide the selection of the new service location. Therefore, just local interaction and a very low amount of communication is necessary to choose a new service destination. This new placement aims to reduce the amount of communication between the communicating modules.

Additionally, we are grouping the nodes in clusters in order to reduce the organization overhead of the network and allow centralized algorithms to be used inside a single cluster without compromising the whole scalability. All services requested by processing threads inside one cluster must be placed in the same cluster. Hence, each cluster must have enough resources for the service instances. We present in our work two heuristics that enable this cluster formation: one targets stable topologies and the other dynamic ones.

The clustering heuristics should decompose the network in well-connected clusters, because much of the interaction between nodes will occur inside them. The base of our heuristics is the selection of a subset of nodes – the clusterheads – that represent the clusters. Each clusterhead then starts to allocate members. For the selection of clusterheads, we used a method derived from the division of labor in social insects. The membership selection, for our first heuristic (capable of dealing with quasi-static topology), is made based on the fitness of the different member's candidates. Nodes with higher fitness respond first and have higher priority to be incorporated into the cluster. In the second clustering heuristic, a positive and negative feedback mechanisms, typical for self-organizing systems, is used to construct the clusters. The positive feedback is responsible for the aggregation of each cluster, i.e., for its capability of attracting new members. Due to the snow-ball effect of the positive feedback, if not controlled, a network with one single cluster will be achieved. Therefore, the creation of structure, i.e., a cluster with controlled size, is done by means of a negative feedback. Each time that the cluster grows more than necessary, the negative feedback starts to play a larger role and limits the attraction, constraining the cluster size. It shapes the emergent structure in our self-organizing process.

The performance of our proposed algorithms was evaluated using the Shox wireless network simulator. We test the proposed heuristics with different scenarios where network size and node density were adjusted. The results were normalized against a reference approach, which for small scenarios provides the optimal solution and, for large ones, a centralized, computationally intensive genetic algorithm.

The basic and extended service distribution heuristics have shown, in average, very good results for all node densities and network size experimented. They were at most 1.5 times the optimal solution. Both heuristics scale with node density. Nevertheless, although the higher complexity of the extended heuristic, it has shown just a small advantage against the basic one. We suppose that this is due to on the small subset of links used to route almost all packets in the network. For more realistic simulations, where the link metric uses more parameters (not following so regularly the path loss curve) and other factors like congestion and energy are included in the routing algorithm, we expect than our extended heuristic performs much better in relation with to the basic one. Then, the extra effort of the extended heuristic will be compensated.

We also simulated the network decomposition heuristic for "quasi-static" networks. Moreover, an existing heuristic was modified and compared to our results. Our emergent clustering outperforms the modified expanding ring in almost all simulated scenarios in terms of cost of the decomposed network. Moreover, the average cost was at most 1.44 times the reference approach (optimum or GA solutions) for all simulated scenario. Our clustering heuristic has demonstrated a higher predictability with lower standard deviation. Moreover, a stronger geographic separation of the clusters could be verified. This is important to support correlated in-network processing and helps to avoid disturbance among clusters (using, for example, a MAC control inside each single cluster). For increasing node density, the emergent clustering showed just a slightly higher normalized cost. This fact shows that our approach scales with the node density. Further, the predictability of the results increases with node density, i.e., lower variance was found in very dense scenarios.

Concluding, the proposed heuristics, for service distribution as well as for network decomposition can be used successfully in our NanoOS because their small computational cost and local interaction features. Even with those constraints, our simulations showed that very good results could be achieved for all developed heuristics. Other applications of the WSN or wireless ad-hoc network areas may profit from our heuristics too. Automatic distribution of communicating modules as well as cluster formation can be used in several different applications.

Our work gave an additional evidence that emergent properties and self-organization existing in nature can be successfully transferred to computer systems. The very nice properties of self-organizing systems, e.g. emerging structures (global behavior) achieved solely using local iteration (no central control), robustness and high scalability could be verified in the developed heuristics.

Several enhancements can be done in our heuristics. As already described, our clustering algorithm for quasi-static topologies suffers from a worst case cluster complexity of $O(qn)$. A simple procedure can improve this picture drastically: when the clusterhead has accepted enough members in the cluster (the cluster is complete), every member of the cluster is informed about it and then broadcast this information to all new possible candidates (nodes that are still waiting to respond). Upon receiving this information, a candidate just cancels its timer and does not respond to the call for members message.

In order to better distribute the burden imposed to the clusterhead, some mechanism to allow clusterhead rotation should be implemented in the heuristic for “quasi-static” topologies. Such a mechanism is included in our second clustering heuristic.

An additional drawback of our emergent clustering for networks with low topology changes is the lack of a negotiation after the clustering completion. With such negotiation, it could be avoided that isolated nodes have to be simply inserted on existing clusters. This happens because nodes can be exchanged among clusters in order to link isolated nodes and provides the chance that an additional cluster is formed using those isolated nodes.

Moreover, a detailed simulation of our clustering heuristic for networks with moderate topology changes must be done. Further, the addition of the accuracy and realism of the existing simulations, concerning modeling better the hardware and the wireless channels is a future task. How different link metrics reflect in the form and cost of the clusters is an additional point to be verified. Because flat clusters increase the spacial correlation among members, a modification of the clustering objetive in order to improve the formation concerning this aspect should be studied.

The service distribution methods are also candidate for improvements. For example, we aim to integrate our short-range migration method, which is targeted towards a better load balancing among the same instance of a given service by means of migration of a single context, with the service distribution described here.

We are planning to integrate a Sensorware-like migrating control script that allows queries to be easily inserted in the WSN. In the Sensorware, the scripts are used to tie the building blocks implemented inside a service API in some useful application. Nevertheless, the amount of functionality that those API can provide is restricted. We target to combine the idea of those migrating lightweight scripts with our distributed service architecture. Instead of accessing just a restricted local API, the mobile services of NanoOS offer to the scripts larger functionality and distributed processing. The scripts carry a high level query that is executed by our distributed service architecture. Scripts control their migration/replication by themselves (e.g. trying to match certain data in the sensor network) whereas the service API migration is driven by the OS. This brings a better combination of data-centric applications with our address-centric distributed service architecture.

An additional very important step into this work is the integration of the proposed heuristics in a real implementation of the NanoOS. This will enable the development of a demonstrator where the advantages of our approach can be tested in several real scenarios.

Bibliography

[1] 3d integrated micro/nano modules for easily adapted applications (e-cubes project). accessed on november 23, 2007. http://ecubes.epfl.ch/public/.

[2] The ns-2 network simulator. accessed november 10, 2007. http://nsnam.isi.edu/nsnam/index.php/User_Information.

[3] Ravindra K. Ahuja, James B. Orlin, and Ashish Tiwari. A greedy genetic algorithm for the quadratic assignment problem. *Comput. Oper. Res.*, 27(10):917–934, 2000.

[4] A. D. Amis, R. Prakash, T. H. P. Vuong, and D.T. Huynh. Max-min d-cluster formation in wireless ad hoc networks. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 32–41vol.1, 26-30 March 2000.

[5] B. Awerbuch, A. V. Goldberg, M. Luby, and S. A. Plotkin. Network decomposition and locality in distributed computation. In *IEEE 30th Annual Symposium on Foundations of Computer Science*, NV, USA, 1989.

[6] D. Baker and A. Ephremides. The architectural organization of a mobile radio network via a distributed algorithm. *IEEE Transactions on Communications*, 29(11):1694–1701, Nov 1981.

[7] D. J. Baker, A. Ephremides, and J. A. Flynn. The design and simulation of a mobile radio network with distributed control. *IEEE J. on Selected Areas in Communications*, SAC2(1):226–237, 1984.

[8] S. Bannerjee and S. Khuller. A clustering scheme for hierarchical control in wireless networks. In *Proceedings of the IEEE INFOCOM*, Anchorage, AK, April 2001.

[9] Rimon Barr, John C. Bicket, Daniel S. Dantas, Bowei Du, T. W. Danny Kim, Bing Zhou, and Emin Sirer. On the need for system-level support for ad hoc and sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(2):1–5, 2002.

[10] S. Basagni. Distributed and mobility-adaptive clustering for multimedia support in multi-hop wireless networks. In *Vehicular Technology Conference, 1999. VTC 1999 - Fall. IEEE VTS 50th*, volume 2, pages 889–893vol.2, 19-22 Sept. 1999.

[11] S. Basagni. Distributed clustering for ad hoc networks. In *Parallel Architectures, Algorithms, and Networks, 1999. (I-SPAN '99) Proceedings. Fourth InternationalSymposium on*, pages 310–315, 23-25 June 1999.

[12] S. Basagni, I. Chlamtac, and A. Farago. A generalized clustering algorithm for peer-to-peer networks. In *Proceedings of the Workshop on Algorithmic Aspects of Communication (Satelite workshop of ICALP), Bologna, Italy*, 1997.

[13] Stefano Basagni. Distributed and mobility-adaptive clustering for ad hoc networks. Technical report, University of Texas, July 1998.

[14] P. Basu, N. Khan, and T. Little. A mobility based metric for clustering in mobile ad hoc networks. In Proceedings of Distributed Computing Systems Workshop, 2001.

[15] Christian Bettstetter. On the minimum node degree and connectivity of a wireless multihop network. In *MobiHoc '02: Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing*, pages 80–91, New York, NY, USA, 2002. ACM.

[16] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. Mantis os: An embedded multi-threaded operating system for wireless micro sensor platforms. *ACM/Kluwer Mobile Networks and Applications (MONET) Journal, Special Issue on Wireless Sensor Networks*, 2005.

[17] S. H. Bokhari. On the mapping problem. *IEEE Trans. Computer*, C-30:207–214, 1981.

[18] E. Bonabeau, G. Theraulaz, and J.-L. Deneubourg. Quantitative study of the fixed threshould model for the regultaion of division of labour in insect societies. In *Proceedings Roy. Soc. London*, number 263 in B, pages 1565–1569, 1196.

[19] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press, New York, NY, 1999.

[20] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press, Santa Fe Institute Studies in the Sciences of Complexity, New York, NY, 1999.

[21] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management*, number 3-14, 2001.

[22] A. Boulis and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proc. of the First International Conference on Mobile Systems, Applications, and Services (MobiSys 2003), San Francisco, CA, USA*, San Francisco, CA, USA, May 2003.

[23] William Joseph Butera. *Programming a paintable computer*. PhD thesis, Massachusetts Institute of Technology, 2002.

[24] Scott Camazine, Jean-Louis Deneubourg, Nigel R. Franks, James Sneyd, Guy Theraulaz, and Eric Bonabeau. *Self-Organization in Biological Systems.* University Presses of CA, 2003.

[25] T. Camp, J. Boleng, and V. Davies. A survey of mobility models for ad hoc network research. *Wireless Communications & Mobile Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications*, 2(5):483–502, 2002.

[26] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. In *IEEE TOSE*, pages 141–154, Feb. 1988.

[27] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat monitoring: application driver for wireless communication technology. In *Proceedings of the Workshop on Data Communication in Latin America and the Caribbean*, Costa Rica, April 2001.

[28] Alberto Cerpa, Naim Busek, and Deborah Estrin. Scale: A tool for simple connectivity assessment in lossy environments. Technical report, Canter for Embedded Networked Sensing (CENS), University of California, Los Angeles, September 2003.

[29] Dipanjan Chakraborty, Anupam Joshi, Tim Finin, and Yelena Yesha. GSD: A Novel Group-based Service Discovery Protocol for MANETs. In *4th IEEE Conference on Mobile and Wireless Communications Networks (MWCN)*, Stockholm. Sweden, September 2002.

[30] Steve J. Chapin. Distributed and multiprocessor scheduling. In Allen B. Tucker Jr., editor, *Computer Science and Engineering Handbook*, chapter 87, pages 1870–1882. CRC Press, 1996.

[31] Harry Chen, Tim Finin, and Anupam Joshi. Service discovery in the future electronic market. In *Proceedings of the Workshop on Knowledge-based Electronic Markets*, Austin, Texas, USA, 2000.

[32] Y. Chen, A. Liestman, and J. Liu. Clustering algorithms for ad hoc wireless networks. In Ad Hoc and Sensor Networks, 2004.

[33] Y. P. Chen and A. L. Liestman. A zonal algorithm for clustering ad hoc networks. *International Journal of Foundations of Computer Science*, 2003.

[34] Shyamal Chowdhury. The greedy load sharing algorithm. *J. Parallel Distrib. Comput.*, 9(1):93–99, 1990.

[35] Brent N. Clark, Charles J. Colbourn, and David S. Johnson. *Unit disk graphs*, volume 86. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 1990.

[36] Microsoft Corp. Understanding universal plug and play. http://www.upnp.org, 2000.

[37] Douglas S. J. De Couto, Daniel Aguayo, John C. Bicket, and Robert Morris. A high-throughput path metric for multi-hop wireless routing. In *Proceedings of the ACM International Conference on Mobile Computing and Networking (MobiCom)*, San Diego, CA, September 2003.

[38] Douglas S. J. De Couto, Daniel Aguayo, Benjamin A. Chambers, and Robert Morris. Effects of loss rate on ad hoc wireless routing. Technical report, Massachusetts Institute of Technology (MIT), Laboratory for Computer Science, March 2002.

[39] David E. Culler, Jason Hill, Philip Buonadonna, Robert Szewczyk, and Alec Woo. A network-centric approach to embedded software for tiny devices. In *EMSOFT 01: Proceedings of the First International Workshop on Embedded Software*, pages 114–130, London, UK, 2001. Springer-Verlag.

[40] Eoin Curran. Swarm: Cooperative reinforcement learning for routing in ad-hoc networks. Master's thesis, University of Dublin, Trinity College, September 2003.

[41] Vanessa Davies. Evaluating mobility models within an ad hoc network. Master's thesis, Colorado School of Mines, 2000.

[42] Vinicius C. de Almeida, Luiz F. M. Vieira, Breno A. D. Vitorino amd Marcos A. M. Vieira, Jose A. Nacif, Antonio O. Fernandes, Diogenes C. da Silva, and Claudionor N. Coelho Jr. Sistema operacional yatos para redes de sensores sem fio. Technical report, Universidade Federal de Minas Gerais, 2005.

[43] Marco Dorigo and Alberto Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics*, 26:1–13, 1996.

[44] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *First IEEE Workshop on Embedded Networked Sensors*, Florida, USA, 2004.

[45] Prabal K. Dutta and David E. Culler. System software techniques for low-power operation in wireless sensor networks. In *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 925–932, Washington, DC, USA, 2005. IEEE Computer Society.

[46] e Cubes consortium. Report about appropriate real-time operating systems. Technical report, EU Project e-Cubes, 2006.

[47] A. Ephremides, J. E. Wieselthier, and D. J. Baker. A design concept for reliable mobile radio networks with frequency hopping signaling. *Proceedings of the IEEE*, 75:56–73, 1987.

[48] M. Eshaghian and Y. Wu. Mapping heterogeneous task graphs onto heterogeneous system graphs. In *Proceedings of Heterogeneous Computing Workshop*, 1997.

[49] David Fernandez-Baca. Allocating modules to processors in a distributed system. *IEEE Transactions on Software Engineering*, 15(11):1427–1436, November 1989.

[50] Christian Frank, Vlado Handziski, and Holger Karl. Service discovery in wireless sensor networks. Technical report, Technical University of Berlin, 2004.

[51] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with smartnet. In *In Proceedings of Heterogeneous Computing Workshop*, 1998.

[52] A. Fuggeta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Eng.*, 24(5):342–361, 1998.

[53] Mario Gerla and Jack Tzu-Chieh Tsai. Multicluster, mobile, multimedia radio network. *Wirel. Netw.*, 1(3):255–265, 1995.

[54] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addision-Wesley, 1989.

[55] Deepak Gupta and Pradip Bepari. Load sharing in distributed systems. In *Proceedings of the National Workshop on Distributed Computing*, 1999.

[56] E. Guttman. Service location protocol: Automatic discovery of ip network services. *IEEE Internet Computing*, vol. 3(no. 4):pp. 71–80, 1999.

[57] W.B. Heinzelman, A.P. Chandrakasan, and H. Balakrishnan. An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions on Wireless Communications*, 1(4):660–670, Oct. 2002.

[58] Wendi B. Heinzelman, Amy L. Murphy, Hervaldo S. Carvalho, and Mark A. Perillo. Middleware to support sensor network applications. In *IEEE Network*, volume 18, pages 6–14, Jan/ Feb 2004.

[59] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient cammunication protocol for wireless microsensor networks. In *Proceedings 33rd Hawaii International Conference on System Sciences*, Hawaii, Januar 2000.

[60] Hans-Ulrich Heiss and Michael Schmitz. Decentralized dynamic load balancing: The particles approach. In *Information Sciences*, May 1995.

[61] F. Heylighen, C. Gershenson, S. Staab, G.W. Flake, D.M. Pennock, D.C. Fain, D. De Roure, K. Aberer, Wei-Min Shen, O. Dousse, and P. Thiran. Neurons, viscose fluids, freshwater polyp hydra-and self-organizing information systems. *IEEE Intelligent Systems*, 18(4):72–86, Jul-Aug 2003.

[62] J.H. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[63] Xiaoyan Hong, Mario Gerla, Guangyu Pei, and Ching-Chuan Chiang. A group mobility model for ad hocwireless networks. In *Proceedings of the 2nd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, 2000.

[64] Xiaoyan Hong, Mario Gerla, Hanbiao Wang, and Loren Clare. Load balanced energy aware communications for mars sensor networks. In *IEEE Aerospace Conference Proceedings*, 2002.

[65] Juraj Hromkovic. *Algorithmics for Hard Problems*. Springer, Berlin, 2004.

[66] H.B. Hunt III, M.V. Marathe, V. Radhakrishnan, S.S. Ravi, D.J. Rosenkrantz, and R.E Stearns. Nc-approximation schemes for np- and psapce-hard problems for geometric graphs. *Journal of Algorithms*, 26(2), February 1998.

[67] Chaiporn Jaikaeo and Chien-Chung Shen. Adaptive backbone-based multicast for ad hoc networks. In *Proceedings of the IEEE International Conference on Communications*, 2002.

[68] Peter Janacik. Service distribution in wireless sensor networks. Master's thesis, University of Paderborn, 2005.

[69] Holger Karl and Andreas Willing. *Protocols and Architectures for Wireless Sensor Networks*. Wiley, 2005.

[70] Minkyong Kim and Brian Noble. Mobile network estimation. In *Proceedings of the ACM Conference on Mobile Computing and Networking*, Rome, Italy, June 2001.

[71] Ulas C. Kozat and Leandros Tassiulas. Service discovery in mobile ad hoc networks: an overall perspective on architectural choices and network layer support issues. *Ad Hoc Networks*, 2(1):23–44, 2004.

[72] Nectarios Koziris, Michael Romesis, Panayiotis Tsanakas, and George Papakonstantinou. An efficient algorithm for the physical mapping of clustered task graphs onto multiprocessor architectures. In *Proc. of 8th Euromicro Workshop on Parallel and Distributed Processing (PDP2000)*, pages 406–413, Rhodes, Greece, 2000. IEEE Press.

[73] U. C. Kpzat, G. Kondylis, B. Ryu, and M. K. Marina. Virtual dynamic backbone for mobile ad hoc networks. In *IEEE International Conference on Communications (ICC)*, Helsinki, Finland, June 2001.

[74] P. Krishna, N. H. Vaidya, M. Chatterjee, and D. K. Pradhan. A cluster-based approach for routing in dynamic networks. *Computer Communication Review*, 49:49–64, 1997.

[75] R. Krishnan and D. Starobinski. Message-efficient self-organization of wireless sensor networks. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC)*, New Orleans, USA, March 2003.

[76] Rajesh Krishnan. *Efficient Self-Organization of Large Wireless Sensor Networks*. PhD thesis, Boston University, College of Engineering, 2004.

[77] Rajesh Krishnan and David Starobinski. Efficient clustering algorithms for self-organizing wireless sensor networks. In *Ad Hoc Networks*, volume 4, pages 36–59, January 2006.

[78] Rajinish Kumar, Matthew Wolenetz, Bikash Agarwalla, JunSuk Shin, Philip Hutto, Arnab Paul, and Umakishore Ramachandran. Dfuse: A framework for distributed data fusion. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 114–125, 2003.

[79] Mauri Kuorilehto, Marko Hännikäinen, and Timo D. Hämäläinen. A survey of application distribution in wireless sensor networks. *EURASIP J. Wirel. Commun. Netw.*, 5(5):774–788, 2005.

[80] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. In *ACM Computing Surveys (CSUR)*, volume 31, pages 406–471, December 1999.

[81] Christophe Lang, Michel Trehel, and Pierre Baptiste. A distributed placement algorithm based on process initiative and on a limited travel. In *PDPTA*, pages 2636–2641, 1999.

[82] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *ASPLOS X*, 2002.

[83] Q. Li, J. Aslam, and D. Rus. Online power-aware routing in wireless ad-hoc networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (ACM Mobicom '01)*, 2001.

[84] Shuoqi Li, Ying Lin, Sang H. Son, John A. Stankovic, and Yuan Wei. Event detection services using data service middleware in distributed sensor networks. In *IPSN*, 2003.

[85] J. Lian, G.B. Agnew, and S. Naik. A variable degree based clustering algorithm for networks. In *Proceedings of the 12th International Conference on Computer Communications and Networks*, 2003.

[86] J. Lifton, D. Seetharam, M. Broxton, and J. Paradiso. Pushpin computing system overview: a platform for distributed. In *Proceedings of the Pervasive Computing Conference*, 2002.

[87] N. Linial and M. Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993.

[88] Hongzhou Liu, Tom Roeder, Kevin Walsh, Rimon Barr, and Emin Gun Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 149–162, 2005.

[89] Ting Liu and Margaret Martonosi. Impala: A middleware system for managing autonomic parallel sensor systems. In *Proceeding of the ninth ACM SIGPLAN symposium on principles and practice of parallel programming*, 2003.

[90] V. M. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Trans. Comput.*, 37(11):1384–1397, 1988.

[91] Hong Luo, Jun Luo, Yonghe Liu, and Sajal K. Das. Energy efficient routing with adaptive data fusion in sensor networks. In *Proceedings of the 2005 joint workshop on Foundations of mobile computing*, pages 80–88, 2005.

[92] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: a tiny agregation service for ad-hoc sensor networks. In *Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[93] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[94] Sun Microsystems. Jini - technology core platform specification. accessed october 21, 2007. http://www.sun.com/software/jini/specs/jini1.2html/discovery-spec.html.

[95] Brad L. Miller and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.

[96] Thomas J. Mowbray and William A. Ruh. *Inside Corba - Distributed Object Standards and Applications*. Addison Wesley Longman, Amsterdam, 1997.

[97] Job Mulder, Stefan Dulman, and Lodewijk van Hoeseland Paul Havinga. Peeros - system software for wireless sensor networks. Technical report, Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente Enschede, the Netherlands, 2004.

[98] Michael G. Norman and Peter Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Comput. Surv.*, 25(3):263–302, 1993.

[99] M. D. Penrose. On k-connectivity for a geometric random graph. *Wiley Random Structures and Algorithms*, 15(2):145–164, 1999.

[100] Raffaele Perego. A mapping heuristic for minimizing network contention. *Journal of Systems Architecture: the EUROMICRO Journal*, 45:65–82, October 1998.

[101] Christian Prehofer and Christian Bettstetter. Self-organization in communication networks: Principles and design paradigms. *IEEE Communications Magazine*, pages 79–85, July 2005.

[102] Camille C. Price, Stephen F. Austin, and M. A. Salama. Scheduling of precedence-constrained tasks on multiprocessors. *Computer special issue on parallel computing*, 33:219–229, 1990.

[103] S. Ramakrishnan, I. H. Cho, and L. Dunning. A close look at task assignment in distributed systems. In *IEEE INFOCOM 91*, pages 806 – 812, Miami, 1991.

[104] C. V. Ramamoorthy, A. Bhide, and J. Srivastava. Reliable clustering techniques for large, mobile packet radio networks. In *Proceedings of the 6th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 87)*, San Francisco, USA, April 1987.

[105] G. E. Robinson. Regulation of division of labor in insect societies. *Annual Rev. Entomol*, 37:637–665, 1992.

[106] Kay Romer. Programming paradigms and middleware for sensor networks. In *GI/ITG Fachgespraech Sensornetze*, pages 26–27, Feb 2004.

[107] Kay Romer, Oliver Kasten, and Friedemann Mattern. Middleware challenges for wireless sensor networks. In *Mobile Computing and Communications Review*, volume 6, pages 59–61, October 2002.

[108] Franz Rothlauf. *Representations for Genetic and Evolutionary Algorithms*. Springer, Berlin, 2006.

[109] E. Royer, P. M. Melliar-Smith, and L. Moser. An analysis of the optimum node density for ad hoc mobile networks. In *Proc. of IEEE International Conference on Communications (ICC)*, 2001.

[110] Sartaj Sahni and Teofilo Gonzalez. P-complete approximation problems. *J. ACM*, 23(3):555–565, 1976.

[111] Mikko Sarela. Measuring the effects of mobility on reactive ad hoc routing protocols. Technical report, Helsinki University of Technology Laboratory for Theoretical Computer Science, 2004.

[112] Vivek Sarkar and John L. Hennessy. Compile-time partitioning and scheduling of parallel programs. In *SIGPLAN Symposium on Compiler Construction*, pages 17–26, 1986.

[113] Gregor Schiele, Christian Becker, and Kurt Rothermel. Energy-efficient cluster-based service discovery for ubiquitous computing. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop: beyond the PC*, page 14, New York, NY, USA, 2004. ACM Press.

[114] C. C. Shen, C. Srisathapornphat, and C. Jaikaeo. Sensor information networking architecture and applications. In *IEEE Personal Communucations*, 2001.

[115] Behrooz A. Shirazi, Krishna M. Kavi, and Ali R. Hurson, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[116] Brian Shucker, Jeff Rose, Anmol Sheth, James Carlson, Shah Bhatti, Hui Dai, Jing Deng, and Richard Han. Embedded operating systems for wireless microsensor nodes. In Ivan Stojmenovic, editor, *Handbook of Sensor Network: Algorithms and Architectures*. Ivan Stojmenovic, 2005.

[117] S. Singh and C.S. Raghavendra. Pamas: power aware multi-access protocol with signalling for ad hoc networks. *SIGCOMM Comput Communications*, 28:5–26, 1998.

[118] Pradeep K. Sinha. *Distributed Operating Systems*. IEEE Computer Society Press, 1997.

[119] Oliver Sinnen. *Task Scheduling for Parallel Systems*. Wiley-Interscience, University of Aukland, New Zealand, 2007.

[120] Kazem Sohraby, Daniel Minoli, and Taieb Znati. *Wireless Sensor Networks - Technology, Protocols and Applications*. Wiley-Interscience, 2007.

[121] Avinash Sridharan, Marco Zuniga, and Bhaskar Krishnamachari. Integrating environment simulators with network simulators. Technical Report 04-836, University of Southern, California, Log Angeles, 2004.

[122] Ivan Stojmenovic, editor. *Clustering in large-scale networks was proposed as a mean of achieving scalability through a hierarchical approach [122].*, chapter Chapter 4, pages 107–140. John Wiley & Sons, 2005.

[123] Ivan Stojmenovic, editor. *Handbook of Sensor Networks*. John Wiley and Sons Inc, 2005.

[124] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. Software Eng.*, SE-3:85 –93, 1977.

[125] Wind River Systems. Vxworks 5.4 - product overview, June 1999.

[126] Kenjiro Taura and Andrew A. Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *Heterogeneous Computing Workshop*, pages 102–115, 2000.

[127] Hanbiao Wang, Deborah Estrin, and Lewis Girod. Prepocessing in a tiered sensor network for habitat monitoring. In *Jornal on Applied Signal Processing*, 2003.

[128] Hanbiao Wang, Deborah Estrin, and Lewis Girod. Preprocessing in a tiered sensor network for habitat monitoring. *EURASIP Jornal on Applied Signal Processing*, 4:392–401, 2003.

[129] Alex Wild. Insect photography. accessed january 23, 2007. http://www.myrmecos.net/myrmicinae/pheidole.html.

[130] E. O. Wilson. The relation between caste ratios and division of labour in the ant genus pheidole (hymenoptera formicidae). *Behav. Ecol. Sociobiology*, 16:89–98, 1984.

[131] Alec Woo and David Culler. Evaluation of efficient link reliability estimators for low-power. Technical report, UC Berkeley, 2002.

[132] Alec Woo, Sam Madden, and Ramesh Govindan. Networking support for query processing in sensor networks. *Communications of the ACM*, 6:47–52, June 2004.

[133] Alen Woo, Terence Tong, and David Culler. Taming the underlaying challenges of reliable multihop routing in sensor networks. In *Proceedings of the ACM Conference on E,bedded Networked Sensor Systems (SenSys)*, Los Angeles, November 2003.

[134] Michael Wooldridge. *An Introduction to Multiagent Systems*. John Wiley and Sons, Chichster, England, 2002.

[135] John Yannakopoulos and Angelos Bilas. Cormos: A communication-oriented runtime system for sensor networks. *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 342–353, 31 jan-2 feb 2005.

[136] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD*, 31(3), September 2002.

[137] Stephen S. Yau and Fariaz Karim. An energy-efficient object discovery protocol for context-sensitive middleware for ubiquitous computing. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1074–1085, 2003.

[138] W. Ye, J. Heidemann, and D. Estrin. An energy-efficiet mac protocol for wireless sensor networks. In *Proceedings of IEEE Infocom*, 2002.

[139] Yasuhiko Yokote. The apertos reflextive operating system: The concept and its implementation. In *OOPSLA Proceedings*, pages 414–434, 1992.

[140] Y. Yu, B. Krishnamachari, and V. Prasanna. Issues in designing middleware for wireless sensor networks. In *IEEE Network Magazine, 2003*, 2003.

[141] Jerry Zhao and Ramesh Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *Proc. of ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Los Angeles, CA, November 2004.

[142] Biao Zhou, Kaixin Xu, and Mario Gerla. Group and swarm mobility models for ad hoc network scenarios using virtual tracks. In *Proc. of Military Communications Conference (MILCOM)*, Monterey, CA, October 2004.

[143] M. M. Zonoozi and P. Dassanayake. User mobility modeling and characterization of mobility patterns. *IEEE Journal on Selected Areas in Communications*, September 1997.

Own Contributions

[144] Carsten Boeke, Marcelo Goetz, Tales Heimfarth, Dania El Kebbe, Franz J. Rammig, and Sabina Rips. (re-)configurable real-time operating systems and their applications. In *Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, 2003.

[145] Florian Dittmann and Tales Heimfarth. Clock frequency vatiation of partially reconfigurable systems. In *Proceedings of the 19th International Conference on Architecture of Computing Systems*, Frankfurt, Germany, Mar. 2006.

[146] Tales Heimfarth, Klaus Danne, and Franz J. Rammig. An os for mobile ad hoc networks using ant based hueristic to distribute mobile services. In *Proceedings of the Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services (ICAS/ICNS 2005)*, page 77. IEEE Computer Society, 2005.

[147] Tales Heimfarth and Peter Janacik. Ant-based heuristic for os service distribution on ad hoc networks. In *1st IFIP International Conference on Biologically Inspired Cooperative Computing (BICC 2006), volume 216 of IFIP International Federation for Information Processing*, pages 75–84, Boston, MA, USA, August 2006. Springer.

[148] Tales Heimfarth, Peter Janacik, and Franz J. Rammig. Self-organizing resource-aware clustering for ad hoc networks. In *Proceedings of the 5th IFIP Workshop on Software Technologies for Future Embedded and Ubuquitous Systems (SEUS 2007)*, Santorini Island, Greece, Mai 2007.

[149] Tales Heimfarth and Achim Rettberg. Nanoos - reconfigurable operating system for embedded mobile devices. In *In Proceedings of the International Workshop on Dependable Embedded Systems (WDES)*, Florianopolis, Brazil, 2004.

[150] Peter Janacik and Tales Heimfarth. Cross-layer architecture of a distrubuted os for ad hoc networks. In *Proceedings of the International Conference on Autonomic an Autonomous Systems (ICAS 2006)*, Silicon Valley, USA, January 2006.

[151] Peter Janacik and Tales Heimfarth. Emergent distribution of operating system services in wireless ad hoc networks. In *Proceedings of the IFIP Conference on Biologically Inspired Cooperative Computing (BICC)*, Santiago, Chile, 2006.

[152] Peter Janacik, Tales Heimfarth, and Franz J. Rammig. Emergent topology control based on division of labour in ants. In *Proceedings of the IEEE 20th International Conference on Advanced Information Networking And Applications (AINA 2006)*, Vienna, Austria, Apr. 2006.

[153] Franz J. Rammig, Marcelo Goetz, Tales Heimfarth, Peter Janacik, and Simon Oberthuer. Real-time operating systems for self-coordinating embedded systems. In *Proceedings of the 9th IEEE International Symposium on Object and component-oriented Real-time distributed Computing (ISORC 2006),* Gyeongju, Korea, Apr. 2006.