HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Stefan Löwen

Integration of Parked Cars into Car4ICT

Bachelorarbeit im Fach Informatik

20. Dezember 2016

Please cite as:
Stefan Löwen, “Integration of Parked Cars into Car4ICT,” Bachelor Thesis (Bachelorarbeit), Heinz Nixdorf Institute, Paderborn University, Germany, December 2016.

Distributed Embedded Systems (CCS Labs)
Heinz Nixdorf Institute, Paderborn University, Germany

Fürstenallee 11 · 33102 Paderborn · Germany

http://www.ccs-labs.org/

Integration of Parked Cars into Car4ICT

Bachelorarbeit im Fach Informatik

vorgelegt von

Stefan Löwen

geb. am 15. Januar 1994
in Gütersloh

angefertigt in der Fachgruppe

Distributed Embedded Systems
(CCS Labs)

Heinz Nixdorf Institut
Universität Paderborn

Betreuer: **Florian Hagenauer**
Prof. Dr. Falko Dressler
Gutachter: **Prof. Dr. Falko Dressler**
Prof. Dr. Johannes Blömer

Abgabe der Arbeit: **20. Dezember 2016**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Stefan Löwen)
Paderborn, 20. Dezember 2016

Abstract

Cars rapidly gain importance for the information and communication infrastructure in modern cities. The Car4ICT project aims at making cars the base for all communication systems in a city. To improve its stability and versatility stable network nodes are needed. Instead of deploying expensive and hard to maintain roadside units it is possible to use clusters of parking cars as stable and powerful nodes.

In this thesis I used existing prototype implementations of Car4ICT and the Virtual Cord Protocol (VCP) to build a prototype of this concept. VCP is used as the routing protocol inside of a parking cluster. I built a system that allows the creation of large VCP networks with limited hardware and extended the VCP prototype with an implementation of a Distributed Hash Table (DHT). I further accelerated reliable data transmissions of the VCP prototype.

Using DHT I enabled Car4ICT nodes to easily keep their service tables in sync. To conclude the integration of VCP into Car4ICT, however, some work remains to be done.

Kurzfassung

Autos gewinnen zunehmend an Relevanz für die Informations- und Kommunikationsinfrastruktur moderner Städte. Das Car4ICT Projekt schlägt Autos als Grundlage für die gesamte Kommunikationstechnologie einer Stadt vor. Um die Stabilität dieser Plattform zu verbessern und neue Anwendungsbereicht zu erschließen, sind allerdings stabile Netzwerkknoten notwendig. Anstatt auf teure und wartungsintensive *Roadside Units* zu setzen, ist es möglich, parkende Autos zu Clustern zusammenzuschließen und somit stabile, leistungsfähige Netzwerkknoten zu schaffen.

Im Laufe der vorliegenden Arbeit habe ich zwei existierende Prototypen für Car4ICT und das "Virtual Cord Protocol" (VCP) verwendet, um einen Prototypen dieses Konzeptes zu entwickeln. VCP wird hierbei als Technologie für die Vernetzung der parkenden Autos untereinander eingesetzt. Ich habe ein System entwickelt, das es erlaubt, mit wenig Hardware große VCP-Cluster zu testen. Desweiteren habe ich den VCP Prototypen um Funktionen erweitert, die es erlauben, ein VCP-Cluster als verteilte Hashtabelle (DHT) zu verwenden. Im Zuge dessen habe ich die Funktionen des VCP Prototypen zur zuverlässigen Datenübertragung durch eine wesentlich schnellere Implementierung ersetzt.

Aufbauend auf der verteilten Hashtabelle habe ich den Car4ICT-Prototypen so angepasst, dass er seine Service-Tabelle verteilt speichern und somit zwischen mehreren Instanzen synchron halten kann. Zur vollständigen Integration parkender Autos in die Car4ICT Plattform müssen jedoch noch einige Änderungen umgesetzt werden.

Contents

Chapter 1

Introduction

Cars are almost ubiquitous in modern cities and will be so in future cities as well. The Car4ICT project, proposed by Altintas et al. [1], aims at using these cars to form a city-wide communication system. The proposed way of achieving this is by enabling cars to wirelessly communicate with their neighboring cars using e.g., IEEE 802.11p [2] based Dedicated Short-Range Communication (DSRC) and thus creating a loosely connected network spanning the city. At the same time these cars can give other machines or humans the possibility to connect to the network (e.g., via WiFi) and offer or request services. The cars can also offer or request services themselves. Offered services can be storage space, computing power, sharing of an internet connection and the like.

Such a network is great in many ways. Since it does not require pre-deployed infrastructure to be present, it can automatically be established wherever there are enough cars available. This not only holds for modern cities but is especially useful in disaster situations where other means of communication are destroyed, but Car4ICT-equipped vehicles (e.g., emergency vehicles) are available [1]. But the Car4ICT network can also complement existing, working communication systems like LTE or WiFi hotspots.

Simulations of Altintas et al. [1] have shown that such a Car4ICT network is already reliable at densities starting at 85 vehicles per km^2, but nevertheless the high mobility of cars and additional connectivity issues caused by buildings and other obstacles can make the network unstable [3]. One option to further stabilize the network would be to rely on existing infrastructure or to deploy Roadside Units (RSUs), but that would make the platform infrastructure-dependent again and thus mitigate one of its benefits.

A different way to reach the goal of further stabilizing the network is based on the observation that parked cars are present in exactly the contexts in which RSUs would be needed to support a Car4ICT network (i.e., cities). Parking cars could

form local clusters and act as a solid, powerful network node. The additional battery usage for parked cars is tolerable as shown by Sommer, Eckhoff, and Dressler [4, Section 4.1]. This makes the use of parked cars a good alternative to pre-deployed infrastructure for improving Car4ICT's stability. Additionally, they would make the network even more versatile by providing large amounts of storage space, computing power, sensor data and the like. This approach was first proposed by Hagenauer et al. [5].

The goal of this thesis is to evaluate this approach by implementing a prototype of a Car4ICT network that uses the Virtual Cord Protocol (VCP) for clustering parked cars. In order to do this I will build on existing prototype implementations of Car4ICT (without the parked cars) and VCP and integrate them into one system. The Car4ICT prototype is the result of the Master Thesis "Prototype Implementation of the Car4ICT Framework" by Simon Merschjohann [6]. The VCP prototype is the result of the Bachelor Thesis "Experimenteller Aufbau und Validierung von VCP im Rahmen des BATS-Projekts" by Bernhard Weber [7].

Chapter 2

Fundamentals

In this chapter I am going to introduce terms, concepts and other works that this thesis builds upon and that are needed to understand it. I am going to focus mainly on VCP and Car4ICT since they are the fundamentals for the prototype developed in this thesis.

2.1 Vehicular Networking

Vehicular Communcation has seen many advancements in recent years. On the one hand cars are getting more and more connected to Infrastructure like LTE to provide passengers with information and entertainment. On the other hand there is a lot of research regarding DSRC based Inter-Vehicular Communication (IVC). IVC has many safety applications but is not limited to this area. Concepts like Car4ICT suggest using such connected cars as the main communication infrastructure.

2.1.1 DSRC

DSRC is short for Dedicated Short-Range Communication. The term is generally used to refer to short-range to medium-range wireless communications between two cars or between cars and other electronic devices, like RSUs.

An important document specifying DSRC is the IEEE 802.11p [2]. It is an amendment to the IEEE standard 802.11-2007 [8] for Wireless Local Area Networks (WLANs) and is meanwhile incorporated into the newest revision of the standard [9]. It specifies the use of frequencies in the licensed Intelligent Transportation Systems (ITS) band of 5.9 GHz (5850 - 5925 MHz) for Vehicular Networks. The use of these dedicated frequencies reduces interferences with consumer applications working with IEEE 802.11a/n/ac which use the 5 GHz band (5150 - 5350 MHz and 5470 - 5725 MHz in Europe) [10, p.8].

Traditional WLANs require the creation of a Basic Service Set (BSS) before communication is possible [8, p.24]. IEEE 802.11p further defines a mode called *OCB-Mode* which allows the transmission of data frames outside the context of a BSS [2, p.2]. This is especially useful in Vehicular Ad Hoc Networks (VANETs), where the duration of communications between fast-moving cars is often very short, because the *OCB-Mode* allows immediate communication.

2.1.2 Car4ICT

Car4ICT is a research and development project of the Distributed Embedded Systems Group at the University of Paderborn[1] in cooperation with the Toyota InfoTechnology Center. The project aims to investigate vehicles as the main Information and Communications Technology (ICT) resource. Its goal is to make vehicles the base for communication in future smart-cities [1] and even for inter-city communications [11]. What is special about this is that it is not about making cars communicate but to use them for establisihng a base-network which is then used for all other kinds of communication by others.

One of the reasons behind making this project is that future smart cities are probably going to produce and consume huge amounts of data. Centralized and infrastructure-based communication technologies such as cellular networks could be quickly overloaded and are also susceptible to natural disasters. Rebuilding them after such disasters takes large amounts of time and resources. Cars, however, will probably be ubiquitous in future cities (even after disasters) and will be equipped with a wide range of communication technologies as well as large amounts of processing power. This makes them an excellent choice as a base for communication.

Car4ICT defines two roles for entities participating in a network. First there are the consumers and providers, also called *users*. A user can be a sensor offering its sensor-readings, a human using a smartphone, a car providing storage space or any other technical device that is able to somehow participate in the network. Second there are the so-called *members*. Members in Car4ICT are always cars. They are the core of the network and are reponsible for routing messages inside the network and keeping track of all offered and requested services. It is important to note that it is possible for a car to act as user and as a member at the same time. Members mainly communicate via DSRC, although other means of communication are possible. Users connect to the network via members, e.g., by using a WiFi hotspot offered by a member.

As already mentioned, users are able to offer and request services. This is the main communications paradigm in Car4ICT. All communication is built around service offers and service requests. A service is identified by an *Identifier* consisting

[1] see `http://www.ccs-labs.org/projects/car4ict/`

of a hash and a list of metadata items. The hash can be a human-readable word like, e.g., “temperature” for a sensor offering temperature readings or “cpu” for a car offering computing power or it can be an actual hash of the content, e.g., when offering a file. The metadata is an arbitrary list of key-value pairs. Typical entries could be `location=Paderborn`, `type=video`, `size=500MB` and the like. An update on the Car4ICT concept [11] introduces *Position* and *Validity* as mandatory metadata elements[2]. *Position* is mainly needed to make Store-Carry-Forward (SCF) possible, which is needed for routing over long distances, and *Validity* gives the providing side of a service more control over the expiration time of the entry instead of relying on default values for discarding service table entries. Table 2.1 shows an example of how a service table could look.

hash	metadata	providing user
hash(file1)	type=video, size=500MB, topic=sports	5
hash(file1)	type=video, size=500MB, topic=sports	48
storage	type=hours, size=100GB	8
cpu	architecture=ARM, mhz=800	13
chat	rooms=cooking,technology	8

Table 2.1 – Example Car4ICT service table containing hashes, metadata and ids of the respective users offering the service.

For offers the hash field is mandatory and can optionally be extended with metadata to filter the results (e.g., filtering by position to get only nearby offers). For a request, however, the hash is optional and the identifier can consist only of metadata items. The result will then contain all services matching these items, regardless the hash.

The process of a user requesting a service generally looks like this:

1. The user authenticates against a member of the network by sending security credentials.
2. If the car is able to successfully verify the credentials, it sends an access grant to the user.
3. The user sends a request to the car, including an identifier.
4. The car checks its service table looking for services matching the request. If matches are found, it sends an answer to the user containing a list of identifiers of the services including ids for the users offering them. If there is no matching service in the local service table of the member-car then the car may start a search among other members that are reachable via DSRC, using an expanding

[2]Since the used Car4ICT prototype by Merschjohann [6] was written before this change I will continue to treat these elements as non-mandatory.

ring search algorithm, or via another communication medium like e.g., LTE. to find more distant services.

5. The requesting user decides which service he wants to use and initiates communication with the offering user. The Data packages needed for this communication are routed by the members.

Figure 2.1 visualizes this process of a user requesting a service.

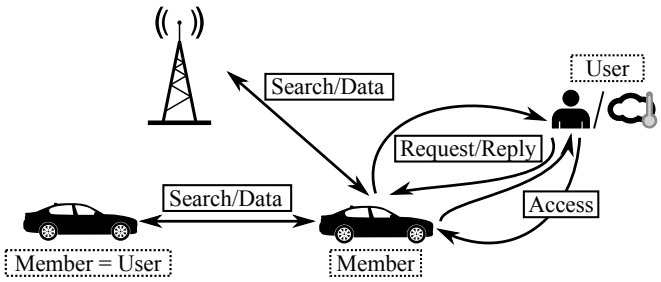

Figure 2.1 – Communications taking place between users, members and external infrastructure when a user requests a service. – based on [1, Figure 2] – ©2015 IEEE

2.1.2.1 Existing Prototype

Simon Merschjohann developed a prototype implementation of Car4ICT as part of his master thesis "Prototype Implementation of the Car4ICT Framework" [6]. This prototype consists of several separate applications of which the *Car4ICT controller*, available as a `controller` binary, is one of the main components. Each running instance of the *Car4ICT controller* represents exactly one member car of the Car4ICT network. It is responsible for service discovery (it keeps a local service-table in memory), message routing and communication with users. Additionally it is able to run so-called *Car4ICT applications* directly as part of its own process. A *Car4ICT application* is what takes the user role in the Car4ICT prototype. These Applications can either be controlled by humans (e.g., via a smartphone) or by machines to provide or use services.

There are two examples of such applications available in the prototype. A *WideAreaStorageApp* and a *TextCharApp*, both consisting of a server part that offers the service and a client part that requests and uses the service. The *WideAreaStorageApp* is an example of a simple file storage service while the *TextCharApp* offers a chat service with multiple chatrooms. Instead of running directly as part of the *Car4ICT controller*, apps can also use the *Car4ICT gateway* component, usable via the `apploader` binary, to run in a separate process and connect to a controller for sending and receiving messages. Apps using the *Car4ICT gateway* can even run on a different machine than

the controller they are using since the gateway only needs to be able to communicate with the controller via a UDP connection.

The last component of the prototype is the *Topology Manager*. It allows the simulation of complex scenarios involving multiple *Car4ICT controllers* running on the same machine. It does so by simulating the quality of connections between cars based on given success-probabilities and is therefore able to mimic complex real-world scenarios that would otherwise require lots of hardware to be successfully tested.

2.2 Distributed Hash Tables

A Distributed Hash Table (DHT) is a distributed system that provides data access in a manner similar to that of a hash table. Probably the best known implementations of DHTs are Tapestry [12], CAN [13], Chord [14], and Pastry [15].

Data in a DHT is organized in `(key, value)` pairs and data items are accessed via their respective key. Each DHT has to define an address space. Each node is responsible for a certain range of these addresses so that the whole address space is covered.

The general process of storing a data item is to take the *key* and put it through a hash function that returns an address from the address space. The key in this case can be the value itself, a file name or an arbitrary key that is somehow attached to the value. The node responsible for the address that the hash function returned is then used as the location to store the `(key, value)` pair. Assuming a good uniformity of the chosen hash function, this leads to quite evenly distributed storage of the data items.

Data access is done similarly. The key is hashed onto the address space and the node responsible for the resulting address is asked for the value that corresponds to the key.

In summary a DHT provides the two basic functions

- `put(key, value)` and
- `get(key)` $\rightarrow$ `value`.

If a distributed system is able to provide these functions it can be called a DHT or characterized as DHT-like.

2.3 The Virtual Cord Protocol

The Virtual Cord Protocol (VCP) is a routing protocol proposed by Awad, German, and Dressler [16], that also provides data management functionality as known

from DHTs. It is mainly designed for Wireless Sensor Networks (WSNs) which are characterized by very limited resources and high mobility. It works by assigning each node a unique, virtual address based on its relative position and organizing these addresses on a virtual cord. It then utilizes this cord for routing and uses the virtual addresses to provide DHT functionality. A key feature is that instead of establishing a base network using one of the routing protocols typically used in Mobile Ad Hoc Networks (MANETs) [3] and building the Distributed Hash Table (DHT) on top of it, like many widely known DHT implementations [4] do, it combines both layers and is thereby able to reduce overhead significantly. Another key feature is that it uses physical proximity information to make sure that neighbors on the virtual cord are also physically close unlike e.g., in Virtual Ring Routing (VRR) [20].

2.3.1 Structure

Addresses in VCP are values within an interval $[S, E]$. Usually, a good choice is to use floating-point numbers between $S = 0.0$ and $E = 1.0$ as this enables a theoretically unlimited number of addresses. In the following I will stick with this choice of interval for reasons of clarity. These addresses are arranged on a virtual cord in ascending order, the first node having the address 0.0 and the last node having the address 1.0.

Each node has one address, but depending on the order of nodes joining, it might be necessary for a node to create a so-called *virtual node* (see Section 2.3.2). Creating such a virtual node is functionally equivalent to assigning another address to the creating node. Figure 2.2 depicts a simple VCP network containg both types of nodes and the virtual cord connecting them.

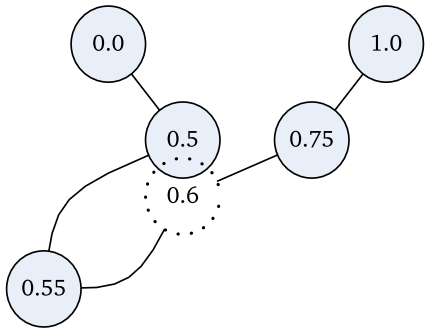

Figure 2.2 – Example of a simple VCP Network. Node 0.6 is a virtual node created by node 0.5. The line shows the formed virtual cord.

Each node is responsible for a certain range of addresses on the cord. The exact range is not explicitly defined in any of the proposing papers for VCP [16], [21],

[3] e.g., DSDV [17] , DSR [18] or AODV [19]
[4] e.g., Tapestry [12], CAN [13], Chord [14] and Pastry [15]

[22]. A valid definition would be that every node with address A is responsible for all addresses in the range $[A, succ(A))$. Using this definition, the node 0.5 in Figure 2.2 would be responsible for the address range $[0.5, 0.55)$ and the virtual node 0.6 would be responsible for $[0.6, 0.75)$. *Responsible* means that the node receives and processes all messages with a destination address in its responsibility interval. In addition to its own position, each node stores the addresses of its predecessor and successor on the cord as well as a table of physically reachable nodes which are simply called *neighbors* [22]. The references to predecessor and successor, as well as the table of *neighbors*, are heavily used in network operation, i.e., joining operations and routing.

The concept of nodes being responsible for an address range so that the whole address range is covered makes VCP suitable as a DHT, hence it is also called a "DHT-like protocol" [16]. What this means is that VCP can easily provide the two functions `put(key, value)` and `get(key)` $\rightarrow$ `value` . To implement this, one has to hash the `key` onto the address range used by VCP, i.e., $[0.0, 1.0]$, and then send the appropriate `put` or `get` to the resulting address.

2.3.2 Joining Nodes / Network Establishment and Operation

Nodes in VCP keep track of their neighbors via periodic *hello* messages. Each node that is part of a VCP network sends these short messages with a predefined frequency. If a node wants to join a network, it first of all listens for these *hello* messages for some time to see whether there is an existing VCP network in its range. If it does not encounter any *hello* messages in this waiting period, it creates a new VCP network by starting to send periodic *hello* messages. If the node encounters one or more *hello* messages during its listening phase, it tries to join the network.

The design of the join operation makes VCP networks highly scalable. A joining node causes changes only to its direct neighbors. If the joining node can communicate with an end node only, i.e., a node with address 0.0 or 1.0, it joins the network by overtaking the end node's address. The former end node gets a new address and both nodes update their predecessor and successor references.

If the joining node can communicate with two nodes that are adjacent on the cord it chooses an address in between these nodes. It then informs both nodes of the change so that they can update their adjecency references.

If neither of the aforementioned conditions is met and the node can only communicate with one node (or multiple non-adjacent nodes) it asks this node to create a virtual node. The asked node then creates a virtual node between itself and its predecessor or successor and thereby enables the joining node to take an address between the address of the real node and the address of the created virtual node. Examples for each of these three cases are shown in Figure 2.3.

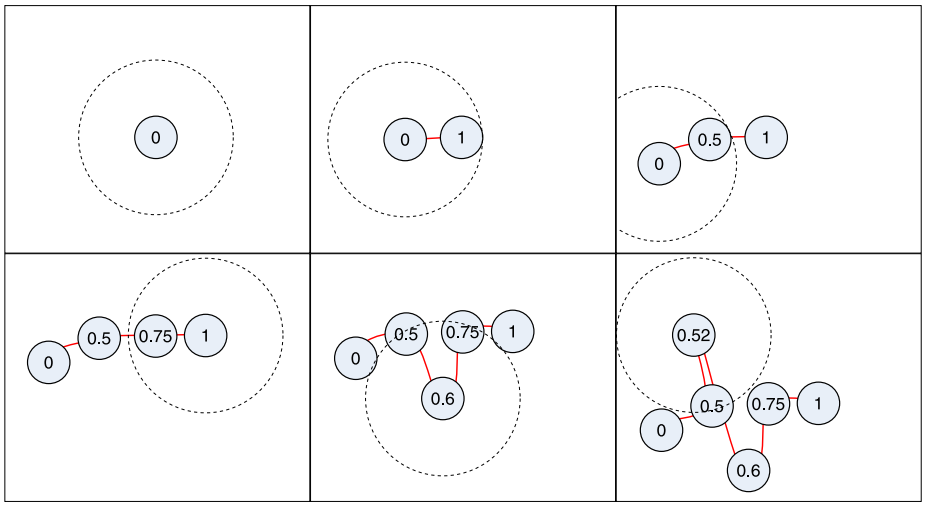

Figure 2.3 – Basic join operation in VCP. The dashed circles mark the communication range of the respective joining node. – Source: [21] – ©2008 IEEE

2.3.3 Routing

As already mentioned, each node keeps a reference to its respective successor and predecessor on the cord. The only exception to this rule are the first node, which does not have a predecessor and the last node, which lacks a successor. These references can already be used for a simple but reliable form of routing. As every node lies on the cord, a sent message can simply travel along the cord until it reaches the node responsible for the destination address of the node. During this, each node has to compare the message's destination to its range of responsible addresses and either keep the message (because it is responsible for it) or pass it to either its successor or predecessor. Assuming an intact cord without failing nodes, this provides an absolutely reliable method of routing without deadlocks.

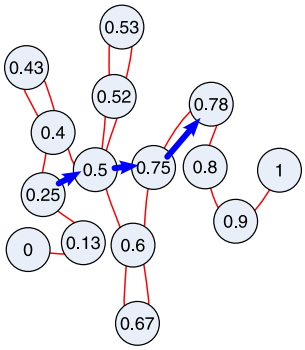

Figure 2.4 – VCP routing path using shortcuts. – Source: [21] – ©2008 IEEE

VCP, however, tries to take additional neighborhood information into account. Each node gets the *hello* messages of its physical *neighbors*. Since these *hello* message contain the address of the sender, each node knows about all addresses that are reachable to it. So a node making a routing decision does not have to stricly follow the cord but can decide to take shortcuts by directly passing the message to the neighbor that is closest to the destination. Figure 2.4 shows such a routing path for a message from 0.25 to 0.78. Simulations have shown that this routing algorithm leads to routing paths with a very low stretch ratio. Even in large networks (up to 225 nodes) the stretch ratio stays below 25% [21]. The stretch ratio is defined as

$$\text{stretch ratio} = \frac{\text{length of path taken by VCP}}{\text{length of shortest path}}. \tag{2.1}$$

Broadcasting a message in VCP is straight forward. The broadcasting node simply has to send the message to the end node positions 0.0 and 1.0. To prevent loops and to guarantee that all nodes are reached, shortcuts are disabled when routing broadcast messages.

2.3.4 Failing Nodes

The above-mentioned network operations work very well as long as all nodes are functioning as expected. However, this does not hold anymore if we assume that nodes can fail. Since nodes in WSNs tend to fail frequently, there has to be a way to deal with such failures. A simple but inefficient way would be to shutdown and rebuild the whole network. This, however, is only feasible in small networks and low frequencies of failing nodes.

A better approach is to try to repair the network [21]. For a failing endnode the respective successor or predecessor can take over the end position. If an intermediate node fails and its successor and predecessor are in each others range, the cord can also be repaired quite easily. If, however, the successor and predecessor of a failed intermediate node cannot reach each other, the network is separated and cannot be repaired. In this case, the successor and predecessor of the failed node have to take over the appropriate end positions for their network segment and thereby create two independent but functional networks.

Awad et al. [22] also extend the initially proposed version of the VCP routing algorithm with failure tolerance so that routing can continue to work even when the network is not yet repaired. If the failed node is the destination node of a message then it might be possible for the last reachable neighbor of the node to handle the request. This is possible, e.g., for DHT-*put*-messages where it is easy for the neighbor to store the data. If, however, the message is a DHT-*get*-request, then the connection counts as failed. A method to prevent such failures of DHT-*get*-messages would be

to introduce data replication onto neighboring nodes. If the failed node is not the destination node but lies on the route to the destination then the routing nodes send back *No Path* messages and try to find an alternative path that circumvents the failed node. The exact details of this process are not important for this thesis and can be found in the proposing paper [22].

2.3.5 Existing Prototype

Bernhard Weber developed a prototype implementation of VCP as part of his bachelor thesis "Experimenteller Aufbau und Validierung von VCP im Rahmen des BATS-Projekts" [7]. This prototype consists of a *VCP-Daemon*, written in C++, as well as a shared library that allows other applications to interact with the VCP daemon. Additional components are a *VCP-Daemon_Client*, written in C++, and a Simulator, written in Java, which were used for evaluation in his thesis.

Using a compile-time switch the VCP daemon can be switched between *simulator mode* and *normal mode*.

- When compiled in *normal mode*, the daemon uses a real network interface to send and receive packets. For sending, it uses a raw socket to be able to skip the transport layer and send packets directly on the network layer. This allows the VCP daemon to define it's own addressing scheme, the VCP adressing, instead of using IP.

- When compiled in *simulator mode* the daemon uses a local TCP port for sending and receiving packets. The packets are, however, the same packets that it would have sent via the raw socket in *normal mode*, i.e., including the ethernet header [7, p.29]. The port must then be connected to an application that handles the correct delivery of the packets. An example of such an application is the Java Simulator also written by Bernhard Weber in the context of his thesis.

The C compatible shared library can be used by other programs to connect to the VCP daemon and retrieve status information as well as to send and receive messages. To receive messages the using application has to register a listener that includes several callback functions for different types of messages and events. The callback messages are then called when the appropriate event occurs.

Although working, the prototype has several limitations which are important to note since they influenced this thesis significantly.

1. The first limitation is the lack of DHT functionality. Although this is an important part of VCP and is also mentioned as such in the thesis by Weber [7] the prototype does not include functionality to save data in the network.

2. The second limitation is in the available modes of transmission for messages. The provided modes are:

   (a) Unreliable transmission: Reception of messages is not acknowledged by the receiving node. The sender has no infomation about the failure or success of the message transmission.

   (b) Transmission with single-hop-sessions: A session, which guarantees and acknowledges the correct delivery of a message, is created for each hop on the path to the destination. The sending node can only know that its message reached the first hop and does not get any information about further hops. Especially the sender does not know if its message reached the destination.

   (c) Transmission with single-hop-sessions and a so-called *nested session*: In addition to the single-hop-sessions mentioned in Item 2b a session is established between sender and receiver. This is the only option if the sending node needs to be sure that its message was delivered successfully.

   The problem here lies in the delays created by the transmission modes using single-hop-sessions. As shown in Table 2.2 single-hop-sessions introduce significant delay.

mode of transmission		min	max	avg
unicast	unreliable	0 ms	1 ms	0 ms
unicast	single-hop-sessions	304 ms	536 ms	334 ms
unicast	nested sessions	307 ms	362 ms	331 ms
broadcast	unreliable	0 ms	10 ms	0 ms
broadcast	single-hop-sessions	748 ms	2292 ms	1405 ms
broadcast	nested sessions	908 ms	2098 ms	1695 ms

Table 2.2 – Transmission times of messages in the original VCP prototype for each transmission mode. 50 messages were transmitted in each mode. In unicast modes the destination was chosen randomly. In broadcast mode all 8 nodes in the test setup had to receive the message. – Source: [7, p.61]

3. Another limitation is the behavior on node failure. As detailed in Section 2.3.4, the simplest but most inefficient reaction to node failure would be a network restart. The prototype unfortunately implements this behavior [7, p.45].

4. The VCP daemon is designed to identify each instance network-wide via the unique MAC address of the WiFi-card that it is using (when compiled in *normal mode*). This limits the number of VCP daemon instances in an experiment to the number of available machines, or, to be exact, the number of available WiFi-cards.

5. Messages in the VCP network are size-limited to one ethernet frame. [7, p.47] This requires additional logic in software that wants to transfer larger amounts of data.

2.4 Macvlan

Macvlan[5] is a linux network driver that allows the creation of multiple sub-interfaces on one physical network interface. Each one of these sub-interfaces has its own MAC address. A macvlan interface can be set into one of the four modes *Private*, *VEPA*, *Bridge*, and *Passthru*, which define different rules for the delivery of packets. The *Bridge*-mode connects all sub-interfaces of a parent interface like a bridge. Packets between such sub-interfaces are delivered locally without being sent out to the physical interface. Packets destined for foreign MAC addresses are sent via the physical interface and broadcasts are delivered locally and also sent out.

2.5 Hard- and Software Requirements

To develop the prototype it was important to be able to test it on real hardware that is as similar as possible to the targeted platform. At the CCS Labs I had three "PC Engines alix3d3" devices at my disposal which I will further refer to as *the alix boxes* or by their respective hostnames *alix3*, *alix4*, and *alix5*. The alix boxes are small single board computers with a 500 MHz *AMD Geode LX800* CPU and 256 MB of DDR DRAM. Each box is equipped with a *VIA Rhine III VT6105M* ethernet card as well as two wireless network cards:

- A Qualcomm Atheros AR5413/AR5414 card (using the ath5k driver)
- A Qualcomm Atheros AR922X card (using the ath9k driver)

The installed operating system is a Debian GNU/Linux 8.5 (jessie). To be able to use 802.11p it has a custom kernel with modifications by Lisovy, Sojka, and Hanzálek [23]. With their modifications to the ath9k driver and some system utilities[6] it is possible to put the AR922X card into OCB mode and onto the 5.9GHz band (see Section 2.1.1).

[5] see linux kernel source: `/drivers/net/macvlan.c`
[6] `https://github.com/CTU-IIG/` search for `802.11p`

2.6 Related Work

2.6.1 Parked Cars in VANETs

Parked cars as support for VANETs and RSUs have already been investigated for several different use-cases.

Liu et al. [24] looked at using parking cars as an alternative for RSUs for relaying messages of driving cars. Parked cars have the advantage of being available in high numbers and without the high costs that deploying and maintaing RSUs implies. They, however, did not include aspects of clustering, storing data or providing services other than relaying of messages. Sommer, Eckhoff, and Dressler [4] also consider single, unclustered cars replacing RSUs, but look specifically at improving intersection safety by using parked cars to forward safety messages that would otherwise be blocked by buildings or other obstacles.

Malandrino et al. [25] look at how parked cars can assist existing RSUs in providing content downloading to cars from a central entity. Their simulations showed that leveraging parked cars in cunjunction with existing RSU infrastructure highly improves download performance as well as coverage and content freshness.

The approach that is most similar to the one in this thesis, is taken by Dressler, Handle, and Sommer [26] in “Towards a Vehicular Cloud - Using Parked Vehicles as a Temporary Network and Storage Infrastructure.” They look at clustering cars using VCP to form vehicular clouds which are then used for storing and retrieving data. The authors, however, focus mainly on the needed extentions to VCP to allow for the needed routing between formed clouds.

The paper this thesis is based on is by Hagenauer et al. [5]. The idea is to extend the already described Car4ICT platform with parked cars. These parked cars form local clusters and are thereby able to work as a large RSU providing stable network nodes and long-running data transfers. Those clusters act as a single Car4ICT node with high computing power, storage capabilities and possibly many offered services from nodes inside the cluster.

Chapter 3

Implementation

As already mentioned previously, the work in this thesis is heavily based on existing prototypes of VCP and Car4ICT which are introduced in Section 2.3.5 and Section 2.1.2.1. As I did not develop the whole system from scratch but was bound to what these existing prototypes offered, I was not able to fully design and plan a coherent system from the ground up but had to plan my work around what was already there. Instead I created a list of changes and new features that would be needed in each of the prototypes to integrate them with each other.

One of the main challenge in the implementation phase was to understand the existing prototypes and the details of their design and to find ways to extend them with the needed functionality in an efficient and extendable way. In the following chapter I am going to describe the concept of the planned system and then proceed to present the changes I have come up with.

3.1 Concept

To integrate parked cars into Car4ICT I planned to connect cars parking within the same parking lot into one network using VCP. So each parking car needs a running VCP daemon. I will call a network of parking cars that are connected via VCP a *cluster*. To keep the WiFi channel free, most of cars within a cluster will disable their Car4ICT radio (but still keep the other Car4ICT functionality running). Only a subset of the cars in a parking lot keep their Car4ICT radio enabled; these cars are called *gateways*. See Figure 3.1 for a visualization of this structure.

A cluster tries to occur as one big node to the Car4ICT network. Therefore all Car4ICT related traffic is routed via the gateways which take over the communication for the whole cluster.

All nodes within a cluster should be able to continue runnig Car4ICT applications, thereby acting as service-consumers and -providers. But since not all of them can be

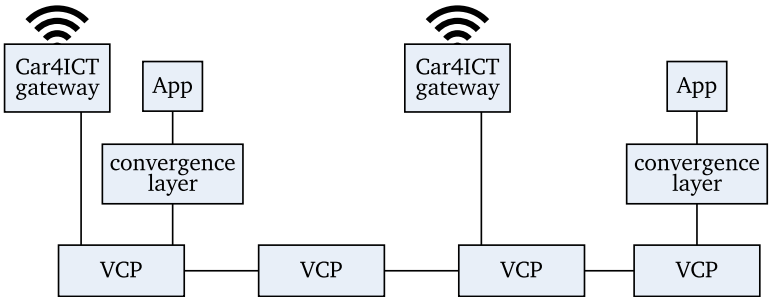

Figure 3.1 – General concept of the planned system. Each parking car has an active VCP module, most cars have disabled their Car4ICT radio but can still run Car4ICT Apps. Only some nodes, called *gateways*, have Car4ICT radio enabled.

expected to have active Car4ICT radio, a *convergence layer* is needed that routes outgoing messages from cluster-internal Car4ICT applications to appropriate gateways and delivers messages received by gateways to the appropriate cluster-nodes that run the applications. In order to do this, the service table needs to be extended with a field noting the VCP address of the cluster-node providing a service. Additionally this service table needs to be kept in sync between all gateways of a cluster. The planned way to accomplish this is to store the service table using the DHT functionality of VCP, evading the need for more complex synchronization schemes.

3.2 Changes to VCP

In this section I will describe the changes to the VCP prototype that were needed and how I approached their implementation.

3.2.1 Multiple Daemons per Machine

To run and evaluate reasonably sized vehicular networks on limited hardware it is necessary to be able to simulate multiple cars per machine. The Car4ICT prototype was already developed with this requirement in mind and has features for this, like the *topology manager* mentioned in Section 2.1.2.1. The VCP prototype on the other hand is designed to identify each daemon network-wide via the unique MAC address of the WiFi-card that it is using. It uses the MAC address to identify the respective next hop of a message that is being routed through the VCP network. This limits the number of VCP daemon instances to the number of available machines. As VCP is used as the networking technique for parking cars it is especially important to be able to run larger numbers of VCP daemons. It was therefore necessary to extend the VCP daemon with this feature.

One way to run multiple VCP daemons on one machine is the Java Simulator mentioned in Section 2.3.5. This simulator is, however, unsuitable for the intended purpose for the following reasons:

1. It is restricted to running locally and does not allow communication via WiFi between machines. It is therefore only useful for simulating VCP networks on one machine but not for simulating larger networks with multiple daemons per machine that span across multiple machines.

2. It is a Graphical User Interface (GUI) based application that depends on a running X-Server. These dependencies are usually not available on embedded systems.

3. It is very resource hungry which is not suited for embedded/low-power systems. Table 3.1 presents a rough view of the RAM and CPU usage on my development-notebook which has a dual-core Intel Core i5-4310U CPU running at a base frequency of 2.00GHz and 12GB of RAM. The used simulation consisted of 9 VCP nodes arranged in a 3x3 grid and sending hello-messages in intervals of 5 seconds. Especially the RAM usage of 350MB would already exceed the available memory in many embedded systems.

Situation	RAM usage (RSS[7])	CPU usage
After startup	135MB	3%
Simulation loaded but not running	150MB	5%
Simulation running (idle)	500MB	30%

Table 3.1 – Approximate resource usage of the Java Simulator as reported by the `ps` command. Measured on a system with an Intel Core i5-4310U CPU running at a base frequency of 2.00GHz and 12GB of RAM.

I therefore tried extending the code to use an additional local ID to tell daemon-instances on the same machine apart but this showed to be rather complex as the daemon was designed from the ground up with the assumption that the MAC address is a unique identifier of a daemon. I would have needed to heavily modify the message format, the routing, and many other aspects of the system. While trying this I came to the conclusion that the code is not easily extendable. Therefore I decided to implement the needed feature using other methods. I first attempted to use the `macvlan` driver of the linux kernel but failed due to packet losses which I could not explain and so I decided to develop a custom system. The failed attempt as well as the custom system are explained in the following two subsections.

[7] Resident Set Size (RSS)

3.2.1.1 Failed Attempt with Macvlan

As described in Section 2.4 the linux kernel's macvlan driver allows the creation of multiple virtual network interfaces, each having a separate MAC address. To use this feature I had to compile the ocb-enabled kernel mentioned in Section 2.5 with activated macvlan driver. I then ran a test using two alix boxes. I will refer to them as *machine A* and *machine B* in this section. On both machines I created macvlan interfaces and started multiple VCP daemons/nodes, assigning each daemon one interface and therefore its own, unique MAC address. I then started the VCP network and observed the following behavior:

- All nodes could successfully receive hello-messages of all other nodes.
- Unicast packets – which are used for all normal communication in VCP apart from hello messages – between nodes running on the same machine arrived successfully.
- Unicast packets between nodes running on different machines did not arrive.

To investigate the problem I wrote a small C program that could send custom ethernet packets using a given network interface and destination MAC address. I then followed the following steps:

1. On *machine B* I created two macvlan interfaces, in bridge mode, using the commands seen in Listing 3.1.
2. On both machines I set the ath9k network interface into OCB mode and configured them to use the same channel. The wired network interfaces (eth0) were connected to a switch.
3. For each of the interfaces wlan1-ath9k, mac_wlan, eth0, and mac_eth, I started `sudo tcpdump -e -i $INTERFACE ether proto 0x1234` on *machine B*. This shows all incoming packets on the given interface and filters them (using my custom protocol type `0x1234`) so that I only see the packets sent using my small program.

```
1 ip link add mac_eth link eth0 type macvlan mode bridge
2 ip link add mac_wlan link wlan1-ath9k type macvlan mode bridge
3 ip link set mac_eth up
4 ip link set mac_wlan up
```

Listing 3.1 – Commands to set up macvlan interfaces

4. On *machine A* I used the small C program to send multiple packets for each of the network interfaces on *machine B*. Packets were sent using the wireless interface `wlan1-ath9k` and the wired interface `eth0` to the following destination addresses:

   - The MAC address of the target network interface
   - A random, non-existent, unicast MAC address (e.g., `00:00:00:00:00:00`)
   - A random, non-existent, multicast MAC address (e.g., `ff:00:00:00:00:00`)
   - The broadcast MAC address (`ff:ff:ff:ff:ff:ff`)

destination MAC address	frame arrived at			
	wlan1-ath9k	mac_wlan	eth0	mac_eth
NICs MAC, unicast	yes	**no**	yes	yes
random MAC, unicast	no	no	yes	no
random MAC, multicast	yes	yes	yes	yes
broadcast	yes	yes	yes	yes

Table 3.2 – Table showing which packets arrived at which network interface on *machine B*. The entry in bold print was responsible for the failed communication between VCP daemons running on different machines.

As seen in Table 3.2 the macvlan interface created on top of a wireless interface did not receive unicast packets from *machine A*. This was the reason for the failure of communication between VCP daemons running on different machines.

To make sure that the lost packets were not caused by the modifications Lisovy, Sojka, and Hanzálek [23] made to the ath9k driver, I ran the experiment multiple times, changing the setup a little bit each time. The following changes to the setup described above were tested:

1. The wireless network interfaces on both machines were put into managed mode and connected to an unencrypted access point, instead of using the OCB mode (compare Item 2).

2. I used ath5k cards in managed mode instead of the ath9k cards.

3. I used a newer linux kernel version (version 4.6.4) without any modifications. As this kernel does not support OCB mode, the tests were restricted to managed mode. This kernel version was, again, tested with the ath9k as well as with the ath5k cards.

The results in all of these experiments were the same. Despite extensive debugging I could not find the reason for these packet losses and therefore dropped the macvlan approach.

3.2.1.2 VCP routing layer

Since I could not solve the problem of lost packets when using macvlan I decided to implement the functionality myself. To do this I developed a system similar to the Java Simulator developed by Weber [7] but with reduced resource usage and without the overhead of a graphical user interface. The result was a program written in C++ that I called *VCP routing layer*.

When compiling the VCP Daemon in *simulator mode* (compare Section 2.3.5) the daemon opens a TCP socket and sends all packets via this TCP socket instead of using a raw socket like when compiled in *normal mode*. I made use of this *simulator mode* and made my *VCP routing layer* connect to the provided TCP socket. When started in *simulator mode* the VCP daemon requires a MAC address as a parameter. This address is used for identifying the daemon in the network. As the daemon is not assigned a real network interface with a MAC address this MAC address can be arbitrarily chosen but has to be unique in the formed VCP network.

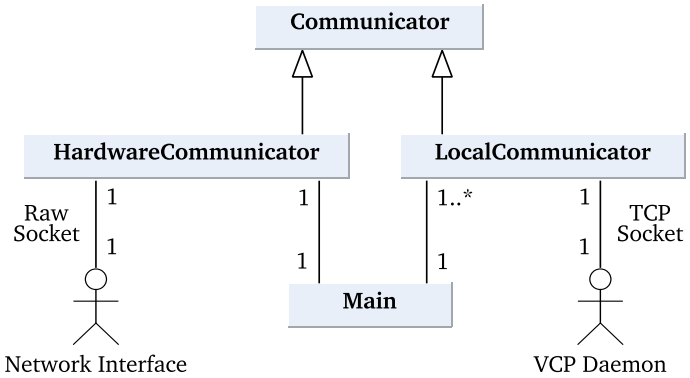

Figure 3.2 – Internal Structure of the VCP routing layer

As seen in Figure 3.2 the *routing layer* internally creates multiple instances of a *Communicator* class. An instance of the *routing layer* has exactly one *HardwareCommunicator* and a theoretically unlimited number of *LocalCommunicator* objects.

- A *LocalCommunicator* is responsible for the communication with a locally running VCP daemon via the provided TCP socket. It delivers messages destined for the daemon and receives messages sent by the daemon to forward them to their destination. All *LocalCommunicator* objects are stored in a map and identified via the MAC address of their assigned VCP daemon.

To be able to deliver messages to their correct destination a *LocalCommunicator* has to parse the ethernet header of the message received from the daemon and extract the destination MAC address. It then executes one of the following actions:

  - If the destination MAC address is found in the map of *LocalCommunicators* the message is handed to the corresponding *LocalCommunicator* which sends it to the attached daemon.
  - If the destination address is a unicast address but is not found in the map of locally managed MAC addresses it is handed to the *HardwareCommunicator* which sends it via its attached network interface.
  - If the destination address is the broadcast MAC address the message is handed to all other *LocalCommunicators* as well as to the *HardwareCommunicator*. This way the message is delivered to the whole VCP network.

- The *HardwareCommunicator* is responsible for handling incoming and outgoing messages via the physical network interface. If the *HardwareCommunicator* gets a message from a *LocalCommunicator* it always adds an ethernet header with `ff:ff:ff:ff:ff:ff` as destination address and `0x1234` as ethertype and broadcasts it via the attached network interface. It hereby keeps the pre-existing ethernet header intact. Because the message is broadcasted, all other machines in the network receive the message via their respective *HardwareCommunicator*. If a *HardwareCommunicator* receives a message via its network interface it discards the outer header and, like the *LocalCommunicator*, extracts the destination MAC address from the inner ethernet header. Based on the destination MAC address it then either forwards it to the appropriate *LocalCommunicator*, broadcasts it to all *LocalCommunicators* or discards it if it does not find the target VCP daemon in its list. Note that it does not rebroadcast the message if it cannot deliver it as this would lead to a broadcast storm.

Each Communicator runs in its own thread using the functionality of the `<thread>` header provided by the C++11 Standard Library. For messages that need to be sent to the attached daemon or network interface each communicator has a message queue. Messages can be enqueued by all communicators but only dequeued by the owning communicator after the message was handled. To synchronize the access to the queue and ensure fast handling of new messages without busy-waiting I again relied on functionality[8] available in the C++ Standard Library since C++11.

The VCP routing layer needs a list of TCP ports to connect to and the MAC addresses to be used for the VCP daemons that it manages. But to allow for situations

[8] mostly `<thread>`, `<mutex>`, and `<condition_variable>`

in which not all VCP daemons join the network at the same time, e.g., when a new car enters a parking lot, it does not require all those TCP ports to be served right away. It periodically checks each port for a serving VCP daemon and adds new deamons to the map of *LocalCommunicators*.

Figure 3.3 shows an example setup of two machines each running three VCP daemons using the developed VCP routing layer. An important limitation to mention is that all nodes that are connected to the same routing layer instance have perfect reception. There is, unfortunately, no functionality like the Car4ICT topology manager (compare Section 2.1.2.1) that enables the simulation of unreliable or bad reception between the nodes.

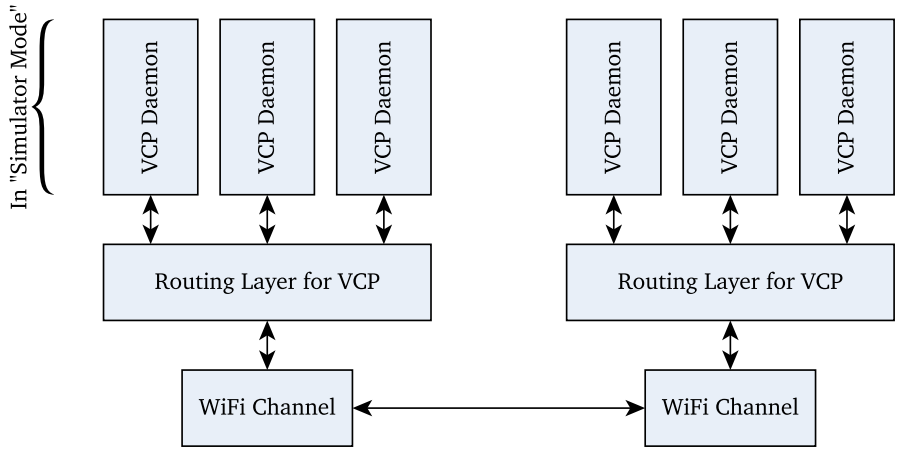

Figure 3.3 – Schematic view of the VCP routing layer running 6 VCP daemons on 2 machines.

3.2.2 DHT Functionality

To be able to store and synchronize data inside a cluster I planned to use the DHT functionality of VCP. Unfortunately, the VCP prototype lacked this feature (compare Section 2.3.5) and therefore I had to implement it myself. As mentioned before, the code of the VCP prototype turned out to be very hard to modify or extend. I therefore decided to implement the needed functionality as a separate layer again. I later merged this layer into the Car4ICT prototype, but as it is still possible to use it without Car4ICT I will continue to describe it separately for the sake of clarity.

The DHT functionality comes in form of a C++ program that can be compiled into an executable binary or be integrated into other software components (as done in Section 3.3). It attaches to a VCP daemon using the provided shared library which communicates with the actual VCP daemon instance via a TCP socket. Besides functions to get status information about the daemon's actual position and

responsible positions, the shared library provides methods to send messages in each of the three transmission modes mentioned in Section 2.3.5 as well as a way to register callback methods which are called when messages are received or the daemon's position changed.

To store a `(key, value)` pair in the VCP network, the source daemon has to calculate the hash of the `key` and convert that hash to a VCP address. It then has to send a message to that destination address containing the key and value to store. For hashing I decided to use the *SuperFastHash* function by Paul Hsieh which promises to be very fast while still having a good distribution of hashes. This makes it well-suited for low-powered systems.

For the actual storage of data on the destination node I developed a simple *StorageManager* interface and built an *InMemoryStorageManager* class implementing this interface. The *InMemoryStorageManager* simply stores all `(key, value)` pairs in a map kept in memory. Due to this design it is, however, easily possible to implement other storage managers to, e.g., store the data on disk.

3.2.2.1 Message Format

For the DHT related communication between VCP daemons I defined a special message format which can be seen in Figure 3.4.

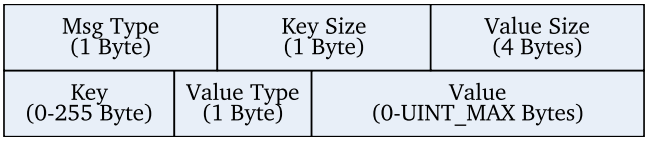

Figure 3.4 – Format of a DHT Message

Message Type: I defined four different message types: `INSERT`, `GET`, `ANSWER`, and `REMOVE`. An `INSERT` requires a key and a value to be present in the message and is used to store the `(key, value)` pair at the receiving node. A `GET` requires only a key and asks the receiving node to answer with an `ANSWER` message containing either the associated value or a *not found* notice if there is not such key stored. And finally a `REMOVE` message requires a key and asks the receiving node to delete the associated `(key, value)` pair from its storage.

Key, Value, Key Size, Value Size: The key and value size fields store the size of the contained key and value in bytes and are needed for correct parsing of the message on the receiving side. The size of both of these fields limits the maximum key size to $2^8 - 1$ Bytes and the maximum value size to $2^{32} - 1$ Bytes. Both sizes may be 0 which indicates that a key or value is not present in the message.

Value Type: This field was introduced to mark values that require special handling apart from the usual storing and retrieving. If no special handling is required this

field is set to a default value. An example for the usage of this field can be found in Section 3.3.

3.2.2.2 Long Messages

The size limitation of messages in the VCP prototype to one ethernet frame per message (compare Section 2.3.5) heavily limits its suitability for DHT. To allow for messages containing larger values I introduced a mechanism for splitting DHT messages into multiple parts if needed. Each message or message part is wrapped in a so-called *DHT packet* which has the format shown in Figure 3.5.

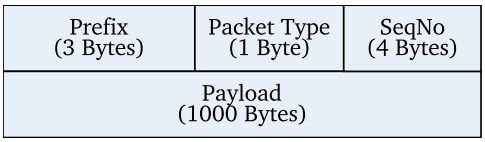

Figure 3.5 – Format of a DHT Packet

Prefix: Each DHT packet starts with a prefix which is used to identify the contents of a VCP message as a DHT packet. In the current implementation the prefix is set to the 3 Bytes `DHT`. If this prefix is not present in a VCP message received from the daemon, the DHT implementation ignores it.

Packet Type: The packet type must be set to `SINGLE` if the DHT packet fully contains a single DHT message. If a DHT message is split into multiple parts the packet type must be set to `INCOMPLETE` for all but the last packet. The last packet of such a multi-packet transmission must have the type `LAST` to signal to the parser that this packet concludes the transmission of a multi-part DHT message.

In addition to these types a special `ACK` type has been defined which is only used for acknowledging the successful reception of a packet. Although the VCP prototype provides dedicated transmission modes for sending messages reliably, these transmission modes introduce a lot of overhead due to the used single-hop-sessions (compare Section 2.3.5). Therefore I decided to only use the unreliable transmission mode and implement my own reliability mechanism which uses acknowledgements only between the end nodes of a communication.

Sequence Number: The sequence number is used to enumerate the packets of a multi-packet message. At the receiver they are used to correctly reassemble the DHT message. For packets of type `SINGLE` the sequence number is always 0. For `ACK` packets the field contains the sequence number of the packet that is acknowledged.

Payload: The payload contains the full DHT message for `SINGLE` packets or a part of the message for multi-packet messages. For packets of type `ACK` the payload is defined to have the fixed length of 8 bytes and has to contain the VCP address to which the acknowledged packet was sent.

This `ACK` payload is needed because each node is responsible for a whole range of addresses. Therefore the node acknowledging a message often has a different address than the one the original packet was sent to. The payload containing the original destination address enables the correct mapping of the acknowledgement to the sent packet.

3.2.2.3 Leaving and failing Nodes

When a VCP daemon fails, the VCP prototype initiates a full network restart. This leads to all nodes getting new positions in the network. For the DHT such a scrambling of addresses would equal a deletion of all data as `GET` all requests would be sent to the wrong nodes (except for the case that a node gets its old address after a restart).

To prevent data loss in these cases, DHT clients detect address changes and changes in the responsibility range of their connected VCP daemon. When such a change happens, the DHT client goes through its storage and moves all `(key, value)` pairs to their new, correct storage location. For the case of a VCP daemon wanting to gracefully exit a VCP network, e.g., when a car leaves a parking lot, I also implemented a function that can be called to copy all DHT data to the node's predecessor. This way the data of the leaving car is not lost but remains available in the DHT.

3.3 Changes to Car4ICT

In this section I describe the changes that were needed in the Car4ICT prototype and how I approached their implementation.

3.3.1 Distributed Service Table

For a VCP cluster to be able to act as one Car4ICT node, all gateways have to have their service tables synchronized. Instead of storing a copy of the service table on each gateway and keeping it in sync between gateways I decided to use the DHT functionality of VCP. Thus there is only one copy of the service table that is stored in the DHT and no further synchronization is needed.

In Car4ICT it is possible that two cars offer services with the exact same hash. A car requesting this hash would then get the IDs of both cars as an answer. It is also possible for a car to update the metadata of an offered service. Further the Car4ICT prototype periodically deletes service table entries that have not received an update (e.g., by re-offering the same service) for a specified amount of time. Implemented with the DHT functionality described in Section 3.2.2 all these features would require frequent read-modify-overwrite operations as the DHT implementation can on store one value per key.

To overcome this inefficiency, I integrated the DHT implementation into the Car4ICT controller and extended it by functionality specifically tailored for managing a distributed service table. I use the *Value Type* field of a DHT message (compare Figure 3.4) to mark the `INSERT` and `GET` messages as containing a value of type *ServiceTableEntry*.

On the storing side I then store the service table entries in a list. If a service table entry is received that matches the service hash and car-ID of an already existing entry, I update only this entry with the new metadata and also update its timestamp. Therefore cars do not need to first request all data regarding a key, update it and then restore it but can simply send their updated service table entry and rely on the receiving side to handle the update.

To integrate the distributed storage into the Car4ICT controller I converted the existing, local *ServiceTable* into an abstract service table interface and made the local *ServiceTable* implement this interface. I then made my distributed version of the *ServiceTable* implement the same interface. This way a controller can easily be switched between using the local or the distributed storage variant or the *ServiceTable*.

3.4 Remaining Work

Unfortunately, due to lack of time, I was not able to implement further functionality on the Car4ICT side of this integration. To complete the concept described in Section 3.1 several features have still to be integrated into the Car4ICT controller.

Specifically, at the current point of implementation, all cars in a VCP cluster still keep their Car4ICT radio enabled. To get the benefit of a large-scale, stable Car4ICT node a gateway selection mechanism has to be developed and all Car4ICT traffic has to be routed through these gateways. Further a handover between these gateways is needed to allow for long-running data transmissions. The needed data synchronization for a handover could build on the DHT functionality described in Section 3.2.2.

Chapter 4

Validation and Evaluation

To validate the correct functionality and evaluate the performance of the developed components I conducted several tests and experiments. The following chapter presents these tests and their results as well as the conclusions that can be drawn from these results.

4.1 Routing Layer

To examine the correctness and performance of the VCP routing layer, I set up an experiment on the alix boxes. On each box I started a VCP routing layer instance and a number of VCP daemons[9] that used the routing layer. The number of nodes per machine started at 5 and went up to 50 in steps of 5. So in the first run I had a network of 15 VCP nodes (3 machines with 5 nodes each) and in the last run I had a network of 150 VCP nodes. The network initiating node (the node that starts the network by sending out hello messages) always was on the alix3 box. Each of the configurations ran for 100 seconds before being terminated. To get more data for the following statistical analysis I ran the whole experiment three times.

To validate the correct function of the routing layer I looked at the neighbor tables of the VCP daemons and especially at the number of distinct network nodes listed in the tables. As the alix boxes stood right next to each other they were in each others range and all nodes should be able to see the whole network. Figure 4.1 shows the average number of nodes in a VCP daemon's neighbor table as a percentage of the network size. These values can be seen as a metric for the connectedness of the network. One can quickly see that with 5 VCP daemons running on each alix box all nodes had 100% of the network (i.e., 15 nodes) in their neighbor tables. The network was therefore fully connected. I conclude that the VCP routing layer is working as intended and is capable of managing at least 5 nodes per machine even

[9] Configured with the default parameters found in [7]

on very limited hardware like the alix boxes. Manual checks of the daemon's log files and neighbor tables confirm this. With 5 nodes per machine the CPU usage of the routing layer was at around 5% (the VCP daemons added another 5%) and its memory usage was approximately 500 kB. This shows that the developed routing layer is well-suited for use on low-resource hardware.

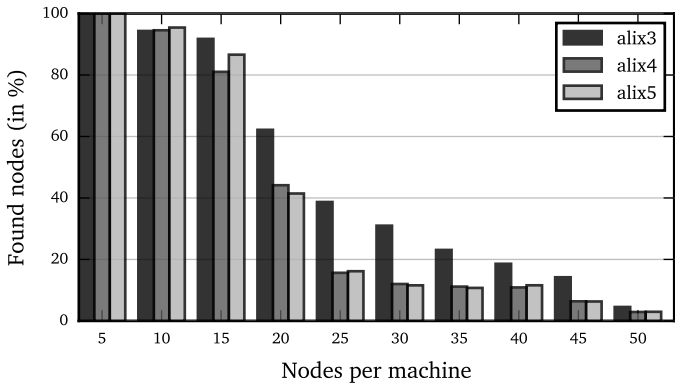

Figure 4.1 – Average network nodes (in percentage of all available network nodes) that each node had in its neighbor table 100 seconds after network start. Nodes with virtual nodes were still counted as one.

With an increasing number of nodes per machine the overall connectedness of the network decreased. This indicates that network sizes with 10 or more nodes per machine were too much for the alix boxes to handle. An important observation here is that the connectedness of nodes running on alix3, the machine with the network initiating node, is almost always significantly higher than on the two other boxes. The same observation can be made when looking at the CPU usage seen in Figure 4.2. While the CPU usage on the alix machine rises with increasing number of nodes, the CPU usage on the other boxes roughly stays the same (starting at 20 nodes per machine). This suggests that the alix3 box is overloaded with such high numbers of nodes and therefore cannot respond to the join requests of nodes from the other machines – at least not before their requests time out (compare [7, p.82]).

To further check this assumption I looked at the reasons that the VCP daemons gave for failed communication sessions. The most frequent reason here was an *ack timeout*. Again the number of reported ack timeouts per node rose quickly on the alix4 and alix5 while staying comparably low on the alix3 (see Figure 4.3).

So I come to the conclusion that the VCP routing layer does not treat nodes running on other machines fairly compared to nodes running on the same machine. This lead to the observed effect of higher connectedness and significantly higher load on the machine with the network initiating node. Meanwhile most of the nodes on the other machines could not communicate and spent a lot of time waiting for

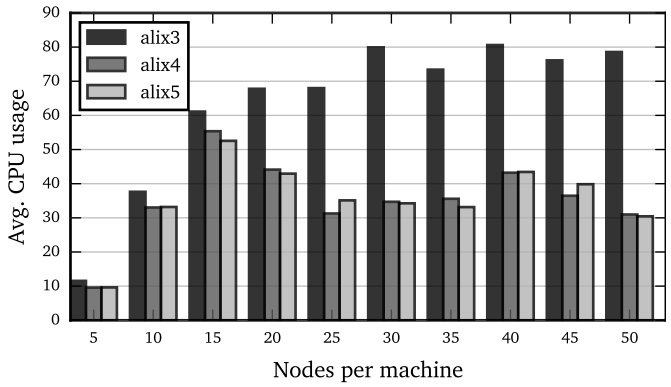

Figure 4.2 – Average CPU usage on the alix boxes in the first 100 seconds after network start.

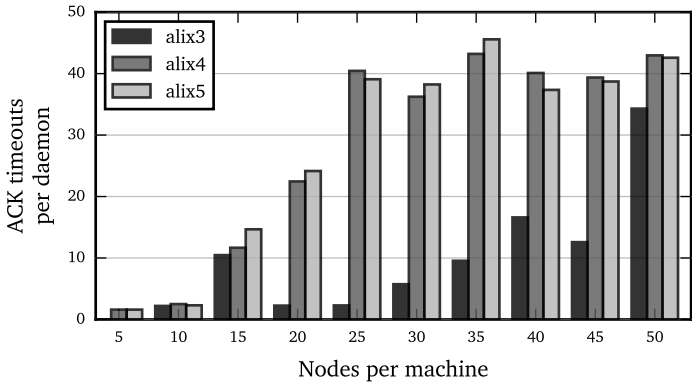

Figure 4.3 – Average number of Session ACK timeouts per daemon.

ACKs for their join requests. This waiting without possible communication explains the lower CPU usage on these machines.

During all experiments the memory usage of the VCP routing layer lay below 2 MB and the overall memory usage of the routing layer and the daemons was below 120 MB. Therefore the low amount of memory did not pose a problem for the experiment.

4.2 DHT

To validate the correct functioning of my DHT functionality I built a small `dht_test_client` which allowed me to manually generate and send DHT messages and to display status information of the VCP daemon and the attached DHT client. I also, temporarily, replaced the *SuperFastHash* function with a simple function that allowed me to directly use VCP addresses as DHT keys and thereby directly choose

the location at which a key-value pair is stored. This change was very minimal and should not influence the results of the following experiment in any way.

I started two VCP daemons on each of the alix boxes. The resulting network of 6 nodes can be seen in Figure 4.4. Then I performed several DHT related actions and documented their effect.

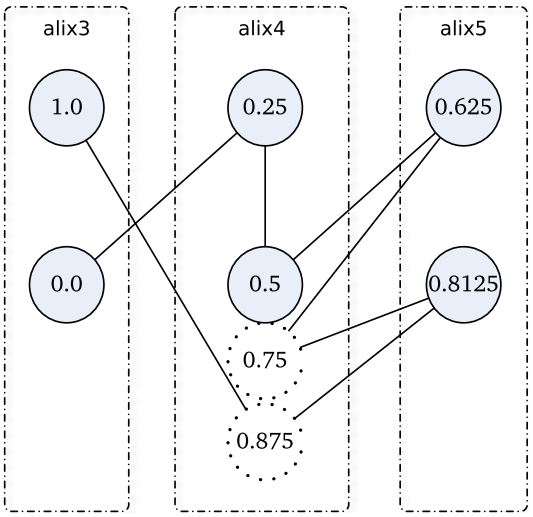

Figure 4.4 – The test network right after startup.

Insert: From node 0.0 I sent three `INSERT` messages with the keys 0.6, 0.75 and 0.9 to their appropriate storage destination. All these addresses fall into the responsibility range of node 0.5 or one of its virtual nodes. On all three messages the daemon running node 0.5 and its virtual nodes reported successful storage of the key-value pair. Each of the transmissions took approximately 650ms.

Get: From node 0.625 I retrieved all three key-value pairs by sending a `GET` message to each of the addresses. Again, the daemon responsible for 0.5 and its virtual nodes responded and sent the requested values to 0.625. To check if there were any errors during storage or retrieval, I compared the values that arrived at 0.625 with the original values stored from 0.0. The comparison showed, that there were no errors in the retrieved data.

Multi-Packet Messages: The values I stored were randomly generated, 10 kB long strings. I chose 10kB long messages because 10 kB are too large for a single DHT packet and therefore the messages each got split up into 10 packets. Each of these packets was successfully sent by the sender and acknowledged by the receiver. This ensured that not only the storage capabilities of the DHT implementation were tested but also the ability to split and transfer large messages.

Leaving Nodes: To test the function for a graceful exit of a node as well as the network behaviour on node failure I made the daemon running nodes 0.5, 0.75

and 0.875 leave the network. For each of the virtual nodes the daemon transferred the stored key-value pair to the respective predecessor on the cord. The affected predecessor nodes were 0.25, 0.625, and 0.8125.

After successfully copying its data to the predecessor nodes, the daemon exited the network. After 30 seconds, which is the configured Node TTL, the other nodes registered the failure of a network node and initiated a network restart. As a result each daemon got assigned a new VCP address and the data items, distributed over 3 nodes before, were successfully moved to their destination in the new network. The new network, including the storage location of the data items is depicted in Figure 4.5.

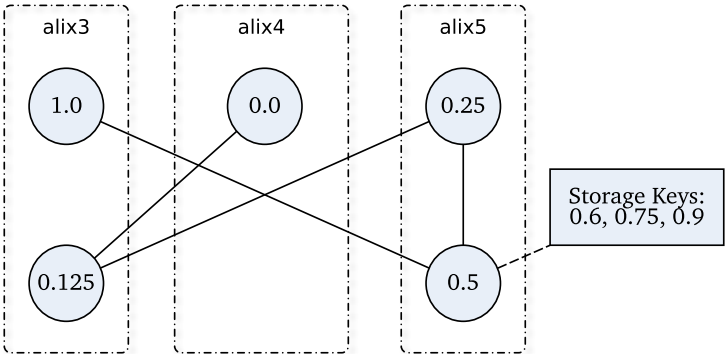

Figure 4.5 – The network, including the storage location of data, after a network restart.

4.2.1 Speed measurements

As described in Section 3.2.2.2 I built a new reliability layer for VCP messages instead of using the already provided `sendReliable` methods provided by the VCP daemon. This choice was made because the provided method uses so-called nested-sessions combined with single-hop-sessions to ensure reliable transmission of messages, which is a lot of unnecessary overhead. My implementation, however, only sents one acknowledgement per packet from the destination node to the source node of a data transmission.

To evaluate the provided performance improvement of this feature, I planned to send messages over multiple hops and measure the time until an acknowledgement is received. Since VCP's routing makes use of shortcuts, the maximum hop count that I could establish with the alix boxes was 2. This could be done by placing the alix boxes in a row with gaps that are big enough for the first box to be out of the transmission range of the last box. A hop count greater than two could not be realized using the alix boxes.

I therefore decided to run this evaluation using the Java Simulator written by Bernhard Weber. Using this simulator I could setup a chain of 11 VCP nodes so that each node could only communicate with its predecessor and successor. I then connected to the end nodes of the chain and sent messages from node 1.0 to the address 0.0. 50 messages were sent using the `sendReliable` method of the VCP daemon, i.e., using nested-sessions as reliability mechanism, and 50 messages were sent using my new end-to-end acknowledgements. To ensure that the message would not be split, I chose a message size of 500 bytes.

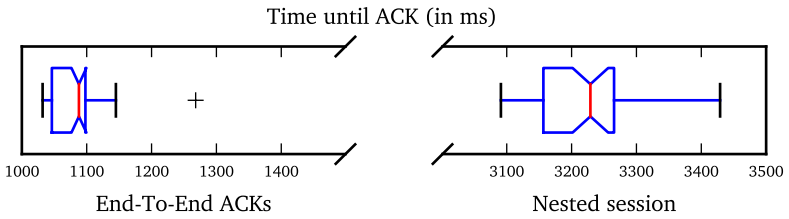

Figure 4.6 – Measured time until an acknowledgement was received for VCP messages sent via 10 hops.

As can be seen in Figure 4.6 the end-to-end acknowledgement system was, in most cases, more than three times faster than the nested-session system. Since all data transfers in the DHT implementation use end-to-end acknowledged transmission, this greatly benefits its performance.

4.3 Car4ICT distributed ServiceTable

As the integration into Car4ICT is not finished yet it was difficult to thoroughly test the implemented changes. I set up two simple tests which I am going to describe in the following.

The first test I performed used the `dht_test_client` that I already used in Section 4.2. I extended this test tool with functionality that allowed me to manually construct service table entries and store them in the DHT network using the special value type for the DHT `INSERT` message. By checking the local storage on the destination node of these storage requests I verified that the service table entries were correctly stored. I then generated a service table entry using the same hash as before but with different metadata items. The `INSERT` message using this entry lead to a successful update of the service table entry on the destination node. `GET` messages asking for the hash were successfully answered with the correct service table entry. This test verified that the DHT implementation applied the correct special behavior specified for service table entries.

For the second test I started two Car4ICT controllers and connected them to a small VCP network. The first controller was configured to offer a Car4ICT service

with the hash *CHAT*. The second controller was configured to request the same hash. The controllers' log files showed that they were able to successfully offer and request their specified service by inserting the related service table entry into the DHT and reading it back. Due to the not implemented functionality further communication between the Car4ICT controllers was not possible.

Chapter 5

Conclusion

During the course of this thesis I became familiar with the existing prototype implementations of VCP [7] and Car4ICT [6]. I developed a concept of how parked cars, using the VCP prototype, could be integrated into Car4ICT. I continued by identifying the changes that needed to be implemented in each of the prototypes.

To enable the creation of large parking lots in form of VCP networks, even on limited hardware, a layer had to be developed which allowed multiple VCP daemons to run on one machine. In the process I discovered and investigated a problem with the macvlan kernel driver in combination with wireless network cards. These problems made macvlan unsuitable as a solution to the task which is why I wrote a separate layer that made multiple VCP daemons per machine possible. Tests of this layer showed that it was able to run at least 5 VCP daemons per machine, even on machines with a very low-powered 500 MHz CPU.

To enable data storage and synchronization between cars in a parking lot, I extended the VCP prototype with a DHT functionality that was missing before. In the course of this development I enabled the VCP prototype to handle large messages that need to be split for transmission. I also developed a new mechanism for reliable data transmission which speeds up transmission by factor 3 when compared to the original reliability mechanism of the VCP prototype.

The Car4ICT prototype was modified to be able to use the DHT implementation to store a distributed version of its service table. This makes it unnecessary to implement complex synchronization strategies between gateways of a VCP cluster.

Unfortunately I was not able to fully complete the integration of parked cars into the Car4ICT platform. I was, however, able to make important steps on the path to a working integration by realizing many of the needed changes. To finalize the integration of parked cars into Car4ICT, some work still remains to be done. A gateway selection algorithm has to be implemented and gateways need the ability to perform a handover to allow for long-running data transfers. Additionally, messages

from and to cars running Car4ICT applications inside a VCP network need to be routed via these gateways.

List of Abbreviations

BSS	Basic Service Set
DHT	Distributed Hash Table
DSRC	Dedicated Short-Range Communication
GUI	Graphical User Interface
ICT	Information and Communications Technology
ITS	Intelligent Transportation Systems
IVC	Inter-Vehicular Communication
MANET	Mobile Ad Hoc Network
RSS	Resident Set Size
RSU	Roadside Unit
SCF	Store-Carry-Forward
VANET	Vehicular Ad Hoc Network
VCP	Virtual Cord Protocol
VRR	Virtual Ring Routing
WLAN	Wireless Local Area Network
WSN	Wireless Sensor Network

List of Figures

List of Tables

Bibliography

[1] O. Altintas, F. Dressler, F. Hagenauer, M. Matsumoto, M. Sepulcre, and C. Sommer, "Making Cars a Main ICT Resource in Smart Cities," in *34th IEEE Conference on Computer Communications (INFOCOM 2015), International Workshop on Smart Cities and Urban Informatics (SmartCity 2015)*, Hong Kong, China: IEEE, Apr. 2015, pp. 654–659. DOI: 10.1109/INFCOMW.2015.7179448.

[2] IEEE, "Wireless Access in Vehicular Environments," IEEE, Std 802.11p-2010, Jul. 2010. DOI: 10.1109/IEEESTD.2010.5514475.

[3] F. Hagenauer, C. Sommer, S. Merschjohann, T. Higuchi, F. Dressler, and O. Altintas, "Cars as the Base for Service Discovery and Provision in Highly Dynamic Networks," in *35th IEEE Conference on Computer Communications (INFOCOM 2016), Demo Session*, San Francisco, CA: IEEE, Apr. 2016, pp. 358–359. DOI: 10.1109/INFCOMW.2016.7562101.

[4] C. Sommer, D. Eckhoff, and F. Dressler, "IVC in Cities: Signal Attenuation by Buildings and How Parked Cars Can Improve the Situation," *IEEE Transactions on Mobile Computing*, vol. 13, no. 8, pp. 1733–1745, Aug. 2014. DOI: 10.1109/TMC.2013.80.

[5] F. Hagenauer, C. Sommer, T. Higuchi, O. Altintas, and F. Dressler, "Using Clusters of Parked Cars as Virtual Vehicular Network Infrastructure," in *8th IEEE Vehicular Networking Conference (VNC 2016), Poster Session*, Columbus, OH: IEEE, Dec. 2016, pp. 126–127.

[6] S. Merschjohann, "Prototype Implementation of the Car4ICT Framework," Master's Thesis, Department of Computer Science, Aug. 2016.

[7] B. Weber, "Experimenteller Aufbau und Validierung von VCP im Rahmen des BATS-Projekts," Bachelor Thesis, Institute of Computer Science, Mar. 2014.

[8] "Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," IEEE, Std 802.11-2007, 2007. DOI: 10.1109/IEEESTD.2007.373646.

[9] IEEE, "Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," IEEE, Std 802.11-2012, 2012.

[10] ETSI, "Broadband Radio Access Networks (BRAN); 5 GHz High Performance RLAN; Harmonized EN Covering the Essential Requirements of Article 3.2 of the R&TTE Directive," ETSI, EN 1.8.1, Mar. 2015.

[11] F. Hagenauer, C. Sommer, R. Onishi, M. Wilhelm, F. Dressler, and O. Altintas, "Interconnecting Smart Cities by Vehicles: How feasible is it?" In *35th IEEE Conference on Computer Communications (INFOCOM 2016), International Workshop on Smart Cities and Urban Computing (SmartCity 2016)*, San Francisco, CA: IEEE, Apr. 2016, pp. 788–793. DOI: 10.1109/INFCOMW.2016.7562184.

[12] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing," Berkeley, CA, USA, Tech. Rep., Apr. 2001.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," *SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 161–172, Aug. 2001. DOI: 10.1145/964723.383072.

[14] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *ACM SIGCOMM 2001*, San Diego, CA: ACM, Aug. 2001, pp. 149–160.

[15] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, Nov. 2001, pp. 329–350.

[16] A. Awad, R. German, and F. Dressler, "P2P-based Routing and Data Management using the Virtual Cord Protocol (VCP)," in *9th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2008), Poster Session*, Hong Kong, China: ACM, May 2008, pp. 443–444. DOI: 10.1145/1374618.1374678.

[17] C. E. Perkins and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers," *Computer Communications Review*, pp. 234–244, 1994.

[18] D. B. Johnson and D. A. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks," in *Mobile Computing*, T. Imielinski and H. F. Korth, Eds., vol. 353, Kluwer Academic Publishers, 1996, pp. 152–181.

[19] C. E. Perkins and E. M. Royer, "Ad hoc On-Demand Distance Vector Routing," in *2nd IEEE Workshop on Mobile Computing Systems and Applications*, New Orleans, LA, Feb. 1999, pp. 90–100.

[20] M. Caesar, M. Castro, E. B. Nightingale, G. O'Shea, and A. Rowstron, "Virtual Ring Routing: Network routing inspired by DHTs," in *ACM SIGCOMM 2006*, Pisa, Italy: ACM, Sep. 2006.

[21] A. Awad, C. Sommer, R. German, and F. Dressler, "Virtual Cord Protocol (VCP): A Flexible DHT-like Routing Service for Sensor Networks," in *5th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS 2008)*, Atlanta, GA: IEEE, Sep. 2008, pp. 133–142. DOI: 10.1109/MAHSS.2008.4660079.

[22] A. Awad, L. '. Shi, R. German, and F. Dressler, "Advantages of Virtual Addressing for Efficient and Failure Tolerant Routing in Sensor Networks," in *6th IEEE/IFIP Conference on Wireless On demand Network Systems and Services (WONS 2009)*, Snowbird, UT: IEEE, Feb. 2009, pp. 111–118. DOI: 10.1109/WONS.2009.4801850.

[23] R. Lisovy, M. Sojka, and Z. Hanzálek, "IEEE 802.11p Linux Kernel Implementation," Industrial Informatics Research Center, Czech Technical University, Praque, Czech Republik, Technical Report, Dec. 2014.

[24] N. Liu, M. Liu, W. Lou, G. Chen, and J. Cao, "PVA in VANETs: Stopped cars are not silent," in *30th IEEE Conference on Computer Communications (INFOCOM 2011), Mini-Conference*, Shanghai, China: IEEE, Apr. 2011, pp. 431–435. DOI: 10.1109/INFCOM.2011.5935198.

[25] F. Malandrino, C. Casetti, C.-F. Chiasserini, C. Sommer, and F. Dressler, "The Role of Parked Cars in Content Downloading for Vehicular Networks," *IEEE Transactions on Vehicular Technology*, vol. 63, no. 9, pp. 4606–4617, Nov. 2014. DOI: 10.1109/TVT.2014.2316645.

[26] F. Dressler, P. Handle, and C. Sommer, "Towards a Vehicular Cloud - Using Parked Vehicles as a Temporary Network and Storage Infrastructure," in *15th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2014), ACM International Workshop on Wireless and Mobile Technologies for Smart Cities (WiMobCity 2014)*, Philadelphia, PA: ACM, Aug. 2014, pp. 11–18. DOI: 10.1145/2633661.2633671.