Eberhard Karls
Universität
Tübingen

A Distributed Cooperative Event Data Recorder for Networked Vehicles

Diplomarbeit

Berlin, June 21st, 2005

Horst Rechner

Department of Information and Cognition Sciences
Wilhelm-Schickard-Institute for Computer Science
Eberhard Karls University of Tübingen
Sand 13
D-72076 Tübingen

Betreuer: Prof. Dr. Georg Carle
Assistierender Betreuer: Dipl.-Inf. Andreas Klenk

Hiermit versichere ich, diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Berlin, den 21. Juni 2005

Horst Rechner

Abstract

This thesis describes the design, proof-of-concept implementation, analysis and validation of an event data recorder working in a vehicular ad-hoc networking environment. With this application being a new idea to utilize the capabilities of event data recorders in ad-hoc networks, possible improvements by the merging of these technologies are outlined.
The ad-hoc network enabling distributed recording poses new questions concerning the event signaling and data synchronization between multiple network nodes (vehicles) which are addressed. The design and implementation of a prototype realizing these functions are central part of this work.
Since event data recorders operate in a special domain with respect to data availability and timing accuracy, a validation of the prototype against the IEEE Standard for Motor Vehicle Event Data Recorders is conducted. Finally the performance and accuracy of the prototype operating in a live environment involving two vehicles equipped with an ad-hoc platform is analyzed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With event data recorders and ad-hoc networks being two separate technology domains, this work analyzes the possibilities of their integration to achieve a new operational scope for event data recording.

1.1 Objective and Scope

The objective of this thesis is to show that the integration of an EDR into an ad-hoc network can be achieved and to what extent since EDRs operate under special conditions with respect to timing accuracy and data rates. The IEEE Standard for Motor Vehicle Event Data Recorders will serve as a leveling staff for these attributes.

Paying attention to the above mentioned aspects, the design and implementation of a working prototype and the detailed analysis of its operation will be a result of this work. The operational environment has to be mastered as well as the preparation and conduction of test runs involving two vehicles provided for these tests.

Since this thesis also has to answer the question if a Distributed Cooperative Event Data Recorder (DC-EDR) can be implemented as a software component integrated with future vehicular communication systems, requirements regarding performance and timing accuracy are imposed on the prototype implementation.

Finally a tool for post-crash analysis has to be provided to allow the display of the data collected during the operation of the DC-EDR in a graphical and textual manner.

1.2 Outline

Chapter 2 gives an overview of current EDR technology and describes the limitations of current implementations as well as possible improvements achieved by merging ad-hoc networks and EDRs. These improvements will be illustrated at the end of this chapter by a number of scenarios. This leads to the design of a device realising these aspects, the DC-EDR.

The chapter 3 is about the design of the DC-EDR and introduces the extended trigger as a main concept of the DC-EDR. Because the design is based

on an existing platform, the architecture of this platform is outlined before the different components of the DC-EDR are described.

With a design in place, chapter 4 will go into some details of the implementation which were important for a software system operating with millisecond accuracy. Without providing every detail, central aspects about the implementation like the data format, post-crash analysis method and system optimizations are discussed.

The validation of the DC-EDR in chapter 5 tests the operation of the prototype in several test runs in a lab environment (dra run) and a vehicular environment (test run). Timing and recording accuracy are two aspects that are addressed in depth in this chapter.

The last chapter concludes the work, summarizes the test results and gives an outlook to future prospects.

1.3 Fundamental terms

Event Data Recorders

EDRs, sometimes referred to as "black boxes" have been used in various transportation modes like aircrafts, trains or cars for a number of years. The main unit continuously records data that is collected by a number of sensors inside the transportation and saves it for analysis after a possible accident.

With the evolution of available sensor data and storage capabilities the EDRs recorded more data enhancing the possibilities of post-crash analysis up to the point where video and audio data was included into the recording.

Nowadays the automatic transmission of this data using cellular networks is available for automotive application.

Vehicular Ad-hoc Networks

An ad-hoc network is a form of network that is comprised of a set of nodes which are connected by some form of network link. The nodes are establishing these links on their own, which makes them independent of a centralized structure. With network links forming and changing arbitrarily and every node only having a limited transmission range (one-hop neighborhood), each node has to act as a router for information sent to other nodes (multi-hop neighborhood).

Sensor networks are one application of ad-hoc networks with the goal to miniaturize the nodes forming the network. The nodes themselves are equipped with sensors that provide information for other members of the ad-hoc network.

Vehicular ad-hoc networks have two specialties:

- the nodes are mobile
- the links are wireless

A node in a Vehicular Ad-hoc Network (VANET) is changing its geographic position with higher speed compared to other mobile ad-hoc networks. This has the effect that the link topology of the network is changing very rapidly.

The applications for VANETs are mainly safety applications, communication and informational services which are based on the additional sensor information

available through the network. For an overview of a real life VANET implementation and possible applications, see [CEF02].

Chapter 2

Analysis of EDR technology

2.1 History of Data Recorders

The first practical flight recorder was introduced in 1953 which used a styli to produce oscillographic markings on a tin foil. These recorders were being made mandatory in airplanes[1] since 1957. Since then flight recorders improved their recording capabilities and new recording parameters were made mandatory for certain aircraft types (see [Gro99] for details) by the Federal Aviation Administration.

- 1967: voice recording
- 1988: digital recording

The purpose of current recorders range from crash analysis to maintenance trouble-shooting.

The first airbags appeared in vehicles in the 1970's. To improve the quality of the airbag deployment algorithms, car companies began to integrate data recording devices into the airbag electronics to allow the collection of real world data. The National Highway Transport Safety Agency (NHTSA) realized the need for automatic crash data collection and equipped a fleet of 1000 vehicles in the early 1970's with analog recording devices.

It was not until the 1990's when car manufacturers began integrating devices into their products by default that recorded data mainly for the deployment of occupant protection systems. Since then the usage of this data was possible to crash investigators. However, retrieval and analysis was restricted to car manufacturers because of the proprietary storage and data formats used.

In 1994 General Motors Corporation (GMC) replaced their airbag control module with a device they called Sensing and Diagnostic Module (SDM) and which was the first Original Equipment Manufacturer (OEM) installed EDR which allowed the measuring of longitudinal acceleration rates of the vehicle. Three years later the Ford Motor Company equipped their products with a similar device (Restraint Control Module (RCM)) with which it was possible to record acceleration rates, along with airbag deployment, seat-belt pretensioner and seat position data.

[1] with a weight of over 5700 kilograms that operate above 7600 meters

The primary purpose of these early measuring devices wasn't to provide data in the event of a crash for means of accident reconstruction, but to collect data to improve the algorithms for passive safety systems. EDRs thus can be seen as a byproduct of the airbag industry.

In 1999 GMC rolled out an improved version of their SDM, which now recorded about 5 seconds of pre-crash data in a ring-buffer, including auxiliary data like vehicle speed, engine Revolutions Per Minute (RPM), engine throttle opening and service brake application. In the event of a crash this ring-buffer was frozen and data is preserved for post-crash analysis.

In the same year GMC licensed their data model to the Vetronix Corporation, which allowed the creation of a third party crash data retrieval and analysis software. The CDR System [CDR] allows the analysis of crash data without the consultation of the car manufacturers.

This improved crash reconstruction capabilities dramatically by easing the means by which to recover crash data from the vehicle.

In Europe Siemens VDO Automotive developed a EDR called Unfalldatenspeicher (UDS), which is meanwhile in it's second iteration. The UDS records acceleration rates, current speed, brake application, engine throttle opening, turn indicator and other sensory data. Currently about 40 000 vehicles are equipped with this type of EDR worldwide.

2.2 Current Technology

2.2.1 Crash Relevant Data

EDRs are used to record sensory data that describes the state of a vehicle before, during and after a crash.

This data includes internal sensory data [JHB98] such as

- longitudinal and lateral acceleration rates (which are also used for airbag deployment algorithms) [2]
- speed
- positional data (Global Positioning System (GPS) data)
- car related parameters (temperature, oil pressure, brake fluid status)
- driver related car operating parameters (brake, light and seat belt status)

and external sensory data such as

- audio and video information [MPB98, Bag99, Ray99]
- electronic IDs of vehicles in the vicinity [Sza88, CGTW99]
- other external sensory information (temperature, road condition)

While internal data describes the state of the car itself and information about the driver, external data captures information about the vicinity of the car.

[2] to reduce the amount of data acceleration is often represented by low frequency velocity change samples Δv

2.2.2 Storage

The collected information is stored on non-volatile memory in a tamper-proof fashion to allow post-crash analysis. The storage device is housed in a fire-proof shock-absorbing box that is placed in a statistically safe place inside the car.

In order to allow crash reconstruction the recorded data has to cover pre- and post-crash time. This is achieved through a ring buffer constantly recording data, even if no crash is indicated. After a certain amount of time, the earliest recorded data is overwritten by the most recent data. In the event of a crash (or near-accident, such as sharp breaking) the logging stops thus preserving a certain amount of pre-crash data. In case of the UDS 2.0 from Siemens VDO Automotive the collected data consists of

- 30 seconds of pre-crash data
- 15 seconds of post-crash data

The MACBox from Altius Solutions [Alt] and the DriveCam from DriveCam Video Systems [Dri] record 20 seconds of acceleration data (10 pre-crash and 10 post-crash). Other systems have different recording times. Since the sampling rate of the main signals (acceleration, speed) in a crash situation is between 100 and 1000 Hz, the latter being the sampling rate recommended by the NHTSA [CHR98] and the IEEE Standard for EDRs [IEE04], the recording time of commercially available recorders is below one minute. Other signals (turn signal) will be recorded when they occur.
The number of total captured events can be greater than one. In case of the UDS 2.0 a maximum of 12 events (accidents or manually triggered recordings) can be stored. The storage devices are mainly Programmable Read-Only Memory (PROM)s like flash memory or Electrically-Erasable Programmable Read-Only Memory (EEPROM)s. A typical SDM from GMC holds about 0.5 kilobytes of data [CHMS99].

2.2.3 Data Extraction

Data can be extracted from the EDR in the following ways

- extraction and analysis of the actual memory chip (passive)
- via cable, connected directly to the event data recorder (passive)
- via wireless link (active or passive)

Passive extraction methods require the intervention of a person, whereas active wireless transmission (in most cases a GSM connection, see [MPB98], [Sch01] and [OnS]) are initiated by the EDR. They allow the automatic transmission of crash relevant data to a central database without the interaction of a person. Since crashes put a lot of stress on these devices, telematic services such as wireless data extraction can not be guaranteed. This is why most EDRs implement more than one extraction method.

2.2.4 The IEEE Standard for Motor Vehicle EDRs

In February 2005 the IEEE Standard for Motor Vehicle Event Data Recorders (MVEDRs) [IEE04] was released providing a guideline for all aspects mentioned so far in this chapter. With most of the manufacturers using a proprietary data format the central part of this standard is a data dictionary which describes a list of 86 data elements in detail, providing recording requirements such as accuracy, sampling rate and resolution as well as recommendations for the storage format. Several details from this standard will be used throughout this thesis and will be explained when they are topical.

2.3 Limitations of Current Technology

The following limitations are based on the fact that traditional EDRs do not communicate their state to the surrounding infrastructure at crash time. The communication methods used are post-crash centered and focus on the dispatching of emergency vehicles like in the OnStar System from General Motors [OnS]. This system uses a cellular network connection to contact the OnStar Call Center about the crash and transmits sensory data from GMCs own SDM and additional impact sensors located in the vehicle. The main aspect to keep in mind being that the cellular connection is established as recently as after the crash. Figure 2.1 taken from the OnStar website shows the chronology of this process.

2.3.1 Trigger Event

Even if all cars in an arbitrary scenario are equipped with an intact EDR, only the ones directly involved in the crash would permanently store data to it. This is due to the limitation of the trigger event used to freeze the cyclic EDR data buffer. This trigger event is restricted to local sensory data (the same data or a superset of the data that is stored to it) to determine if a crash occurred. In most cases the variables in this trigger are traversal and longitudinal acceleration rates.

If one of them exceeds a defined threshold (2 gs[3] in case of the SDM), a algorithm decides if the data should be frozen and further actions like airbag deployment should be carried out. This threshold might also be exceeded by sharp breaking[4], but the algorithm will distinguish this event from a crash. Another form to determine a crash is to install shock sensors in the bumpers and doors of the car which, upon impact, send a trigger event to the connected EDR. The EDRs in cars, which are in the vicinity of the crash, but do not experience any of these defined triggers do not store any data. The only additional information is gathered from traditional crash reconstruction techniques. See the Haddon Matrix in [Hai01] for an overview of these techniques.

2.3.2 Location of Sensors

All sensors that are feeding data into the EDR including the ones that determine if the data should be permanently stored, are located at the vehicle thus limiting

[3] acceleration due to gravity with a standard value of $g_n = 9,80665 \frac{m}{s^2}$

[4] normal breaking rarely exceeds $\pm 1g$ for more than a few milliseconds

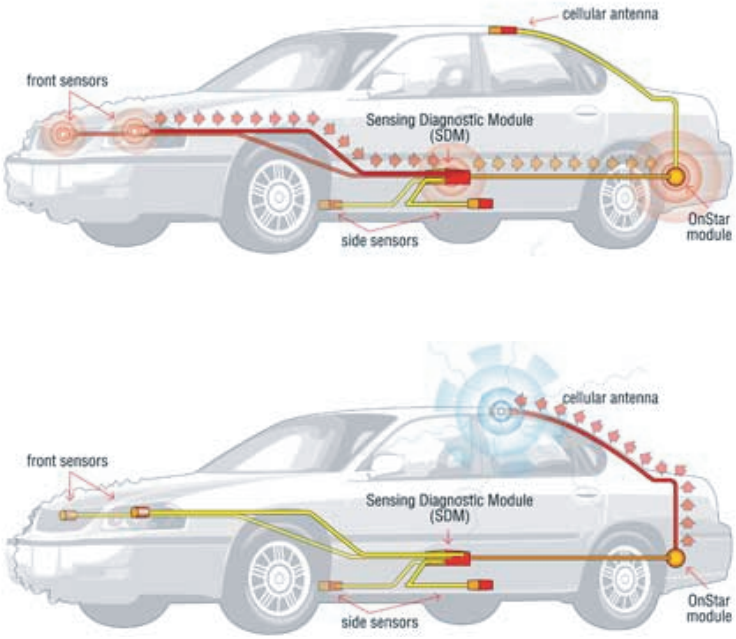

Figure 2.1: OnStar System: mode of operation

the scope of the collected data. It is impossible, for instance, to get a view of the accident from an uninvolved non-human observer if it is not captured by traffic-cameras. Such additional data could help to reconstruct a more complete picture of the crash situation in post-crash analysis e.g. in the event of a mass collision.

Additionally, the events that trigger the EDR could be gathered from a remote location such as a traffic light or another vehicle involved in a crash.

2.3.3 Data Loss

Another problem intrinsic to the centralized nature of EDRs is data loss. If the data store is damaged in the crash and cannot be accessed during post-crash analysis, the reconstruction of the crash is limited to traditional crash reconstruction techniques like thread-mark analysis. If the data could be distributed before the EDR is severly damaged the information could also be gathered from the receivers of this data.

2.3.4 Synchronization of EDR Logs

Inherent to the restriction of triggers to the direct sensors connected to the EDR is, that data logs from multiple EDRs can only be synchronized if the vehicles carrying them were in direct relationship to each other during the event (e.g. in a rear-end collision). In this case, an acceleration registered in one vehicle also registers as an opposite acceleration in the other one. If this condition is not met, it is difficult to synchronize more than one data log by time and establish a linear timeline for the data collected from multiple EDRs. Even if the EDRs store the time of an internal clock in the log files it cannot be made sure that these clocks have recently been synchronized to a central time service.

Time synchronization could help in the reconstruction of multi-car accidents.

2.4 Improvements Achieved by the Use of Ad-hoc Networks

With the involvement of a vehicular ad-hoc network we can relax these limitations by introducing a trigger which is broadcasted by the crashed car to the cars in its vicinity. Subsequently the data stored by the crashed car is transmitted to them.

2.4.1 Additional Sensory and Network Data

By means of the broadcasted crash trigger which acts as a secondary trigger for the EDR, data is also captured by cars outside the immediate vicinity of the crash. This data can improve crash reconstruction through additional environmental data. This aspect gains special importance if video data is included in the EDR logs, since every camera can only capture a limited angle of the overall picture.

With the additional data it is possible to perform plausibility checks to see if the data from one car can be supported by the data recorded by cars in the vicinity presuming that they received the trigger event.

Besides the additional sensory information, the network itself continuously transmits data. The safety applications for instance generate warning and informational messages which can be used to determine if the driver was warned about possible problems ahead (weather condition, road condition, traffic-jam warnings) and acted appropriately (turning on fog tail lamp, slowing down, etc.).

The surrounding infrastructure might also generate information which is of importance to the accident (e.g. the state of traffic lights, information from ramp management and traffic management systems).

The recording of this data would allow reconstruction of the network state as well as the analysis of the behavior of applications. In this case, the EDR would serve as a network probe.

2.4.2 Time Synchronization

Since we deal with a distributed system, the problem of log synchronization across different instances of the EDR can be achieved with respect to the required accuracy using the ad-hoc communication network.

Even though the internal clocks of the systems on which the EDR is running might be synchronized using Network Time Protocol (NTP) or GPS, we cannot rely on them being in perfect sync during the time of a crash. This is the case, if e.g.

- the vehicle has not been in contact of a acNTP time server for synchronization
- the vehicle has been parked in a location without GPS availability, like a garage.

In the event of a crash instant synchronization has to be achieved along with the transmission of crash data. Since NTP and GPS rely on a significant amount of connection time to their master time sources (being a central time server or a satellite) before the clocks are synchronized, they are not ideal in this application domain.

The method introduced in chapter 3 is used to achieve log file synchronization through simple difference time measurement even if the absolute time (which nevertheless will also be recorded) is off by an unknown amount. This information is sent along with the actual crash trigger, which will serve as a global sync event for the different data recorder logs. Since the links in a VANET change rapidly, the combination of synchronization information along with EDR data in one packet guarantees the integrity of the data.

The goal is to achieve millisecond accuracy for log synchronization.

Other methods for time synchronization can be found in [EGE02] and [ER03].

2.4.3 Distributed Storage

EDRs located in vehicles outside the hazard zone that receive the data of the crashed car are comparably safe against the physical effects of the crash, therefore providing an appropriate storage of crash data.

Another safety aspect concerns the manipulation of data. Since the data is now stored on multiple vehicles, manipulation of the data is even more difficult.

The manipulation of the data of one EDR might not be sufficient, because the data would no longer correlate with the data stored on the other vehicles. A scenario that anticipates the altering of all existing data of a crash is feasible, but reminds of the "Murder on the Orient Express", where all involved drivers of the vehicles would change the data together.

2.4.4 Usage Scenarios

The following scenarios will motivate different aspects of distributed cooperative EDRs.

Multiple Perspectives

This scenario will focus on the recording of video data by a passing car to allow an overview of the crash situation.

The pedestrian in figure 2.2 is crossing the road and is not seen by the driver of the car. A crash occurs and the car is sending out a trigger that is received by the passing car. The passing car is saving the crash data (including video information).

The benefit of this additional data is, that the video camera is recording the crash site from a different angle, allowing an overview of the site which can e.g. be sent along with an emergency call to paramedics to get a first impression of the situation they will encounter. This scenario is simulated with the prototype of the DC-EDR in chapter 5.

Infrastructure and Network Information

With the DC-EDR being able to query additional information from surrounding infrastructure integrated into the network (e.g. traffic lights) this information can also be incorporated into the logs. The following scenario illustrates the advantage to crash analysis if such information would be included.

A typical intersection situation is shown in figures 2.5 and 2.6: One car is waiting on a red light while another car is approaching the intersection. The driver of the approaching car is accelerating the vehicle instead of decelerating when the green light turned to yellow, therefore risking a crash with the waiting car. The Active Safety (AS) system in his car is also fed by the information from the traffic lights advising him to decelerate to come to a stop at the intersection.

After the crash the DC-EDRs of both cars stored local sensory data along with information from the traffic lights. In addition, information from the active safety system was stored inside the DC-EDRs allowing the post-crash reconstruction of warning messages issued to the driver. In this case the speeding driver has no chance to wind himself out, because the DC-EDR log shows that

- the AS system reported the traffic light change in time
- the driver accelerated during the approach of the intersection
- the state of the traffic lights was saved pre-crash, showing that the driver ran a red light.

Transmission of Log Data

In case of a severe crash with the risk of data loss the immediate transmission of data through the VANET could save it from destruction. In scenario 2.7 the passing car receives the data from the DC-EDR of the crashed car, which is compromised after the crash, thereby preserving it from destruction by subsequent events of the crash.

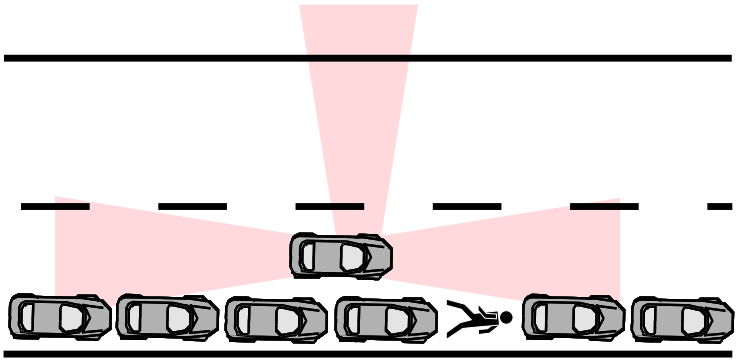

Figure 2.2: Multiple perspectives: pre-crash situation

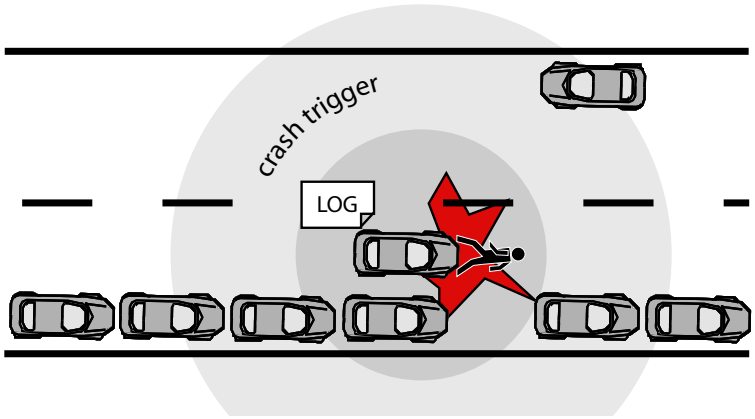

Figure 2.3: Multiple perspectives: crash situation

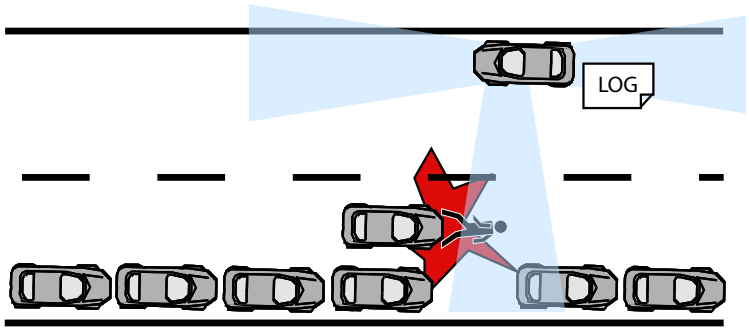

Figure 2.4: Multiple perspectives: post-crash situation

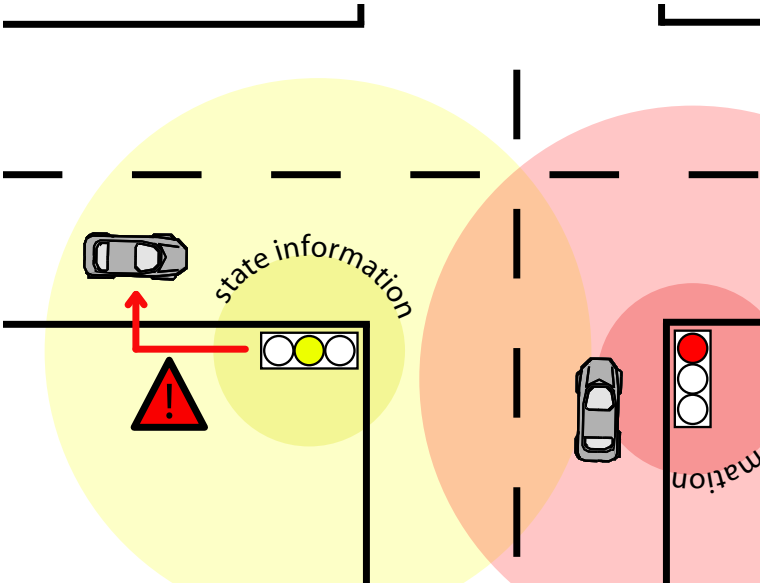

Figure 2.5: Infrastructure information: pre-crash situation

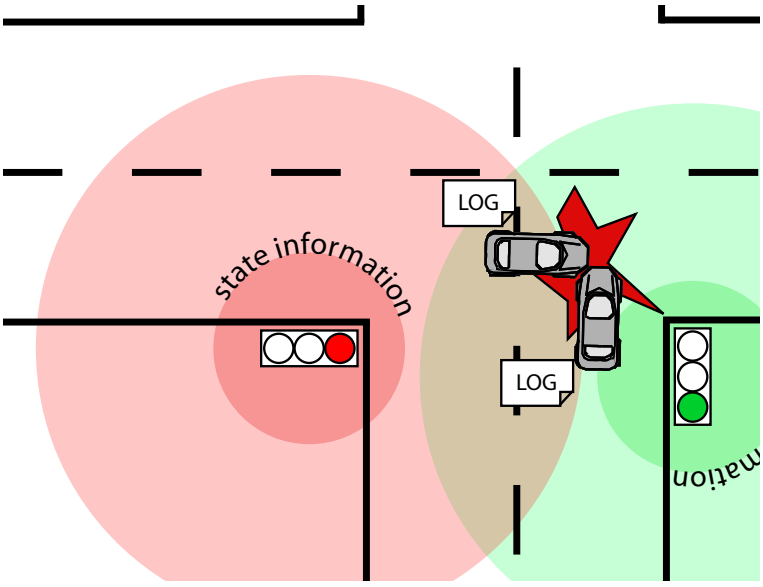

Figure 2.6: Infrastructure information: crash situation

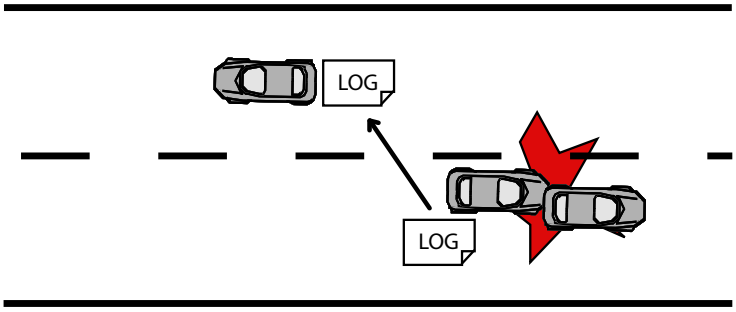

Figure 2.7: Data log transmission: crash situation

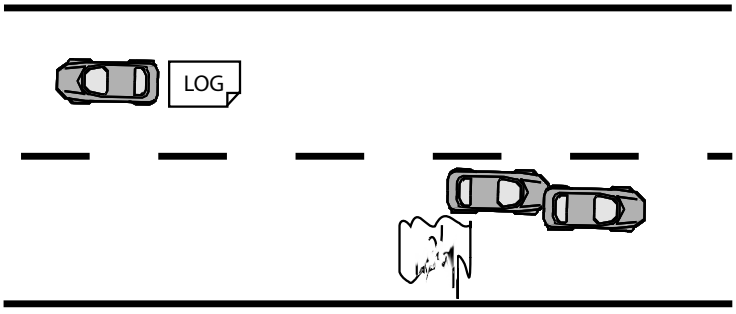

Figure 2.8: Data log transmission: post-crash situation

Chapter 3

Design

This chapter describes the triggering mechanism enabling the distributed cooperative operation of the DC-EDR and the method used to synchronize multiple logs. With the components of the DC-EDR being integrated into an existing platform architecture, the layout of this platform is outlined before going into details of the design of the DC-EDR itself.

3.1 Extended Triggering Mechanism

The main concept of the DC-EDR is an extended triggering mechanism. The primary trigger will still be fired by a car directly involved in an accident or near-accident through traditional data analysis of locally collected data (e.g. acceleration rates). Furthermore the DC-EDR sends out a crash trigger to other cars, which triggers secondary event data recording on vehicles receiving this trigger. The ring buffer of these cars will be frozen, like they were directly involved in the accident. These cars can be seen as "remote sensors" of the directly involved cars.

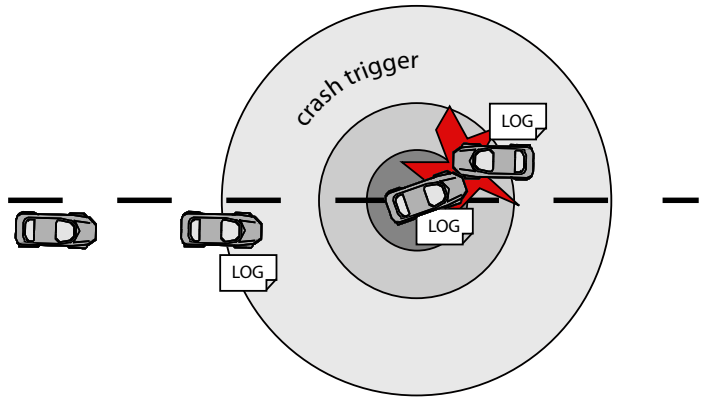

Figure 3.1: Crash trigger sent out to start event data recording on other cars

This thesis will focus on the broadcast of a crash beacon during the event of a crash. In general the broadcast is not limited to crash related events.

With multiple logs describing one event, we have to correlate the logs to allow a correct ordering of the events on a linear timeline. Therefore a synchronization mechanism for the different log files has to be designed providing millisecond accuracy.

Time Difference Method

Figure 3.2 shows the schematic flow of the communication between two instances of DC-EDRs in the event of a crash. Every marker carries a timestamp which allows the calculation of time differences for each of the timestamps on the local instance in respect to a previous timestamp.

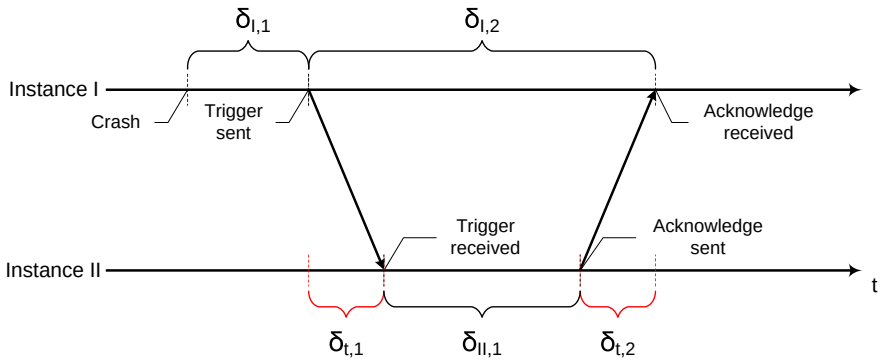

Figure 3.2: Schematic timeline of trigger communication in the event of a crash

Instance I generates a crash trigger and broadcasts it to Instance II, which in its turn extracts the senders address, saves the data of the local DC-EDR to non-volatile memory and returns an acknowledge packet to the sender.

The timeline on Instance I of the car is divided into the following parts:

- $\delta_{I,1}$ represents the time between the generation of the the crash event and the time the trigger was actually sent to the network
- $\delta_{I,2}$ represents the time the car has to wait for an acknowledge of the triggering of the remote DC-EDR

During the wait period $\delta_{I,2}$ the crash trigger is transmitted by the communication infrastructure of the system, processed by Instance II of the DC-EDR and the acknowledge is sent back.

- $\delta_{II,1}$ represents the time between the reception of the trigger and the transmission of the acknowledge packet

The only intervals that do not register in any of the timestamps (and the ones we are interested in) are the transmission times $\delta_{t,1}$ and $\delta_{t,2}$ of the packet. This includes the passage of the data through the IP-stack of the systems and the actual radio transmission delay from instance I to instance II and back.

As can be seen in figure 3.2 we have a correlation between the two timelines with

- $\delta_{I,2} = \delta_{t,1} + \delta_{II,1} + \delta_{t,2}$

With the assumption that we encounter similar conditions regarding the computing and networking environment on each Instance of the DC-EDR we have a symmetric and quasi stationary problem, since transmission times are much smaller than changes to vehicle and network state.

- $2 * \delta_t \approx \delta_{t,1} + \delta_{t,2}$

If we combine the two formulas, we get

- $\delta_t \approx \frac{\delta_{t,1}+\delta_{t,2}}{2} = \frac{\delta_{I,2}-\delta_{II,1}}{2}$

Therefore the difference of the two logs $\delta_{I,II}$ can be computed in the following manner:

- $\delta_{I,II} = \delta_{I,1} + \delta_t \approx \frac{\delta_{t,1}+\delta_{t,2}}{2} = \delta_{I,1} + \frac{\delta_{I,2}-\delta_{II,1}}{2}$

With $\delta_{I,II}$ being the offset from crash registration on instance I (the crashed car) to instance II (the car receiving the trigger) we can simple subtract $\delta_{I,II}$ from the time when the trigger was received on instance II to synchronize the log files.

This method will be evaluated in section 5.4 using the prototype of the DC-EDR.

If this method can be used for two instances of data recorders an iteration for three or more instances can be done in a similar fashion. If instance I is only sending the trigger to its one-hop neighbors the method can be used as is. In the case of a store and forward transmission through multiple instances of DC-EDRs, the difference times $\delta_{I,II}$ have to be added up to compensate for the introduced delay added by every step in the transmission chain.

This method reminds of ping. So why not use ping to measure the delay between two instances of the DC-EDR? One reason was that ICMP was compromised in the ad-hoc protocol making ping times unreliable. The second reason was that the implemented method should measure the actual difference in crash trigger registration times between two DC-EDRs (on OSI/ISO layer 7) and not the run-time of a packet in the underlying network stack (on OSI/ISO layer 3) which would be much smaller.

3.2 Platform Architecture

The DC-EDR is based on an existing ad-hoc network infrastructure developed in the Bundesministerium für Bildung und Forschung (BMBF) FleetNet project [Fle] during 09/2000 and 12/2003. Part of this infrastructure are two test vehicles which are equipped with special hard- and software components enabling ad-hoc network operation. They were provided by the Fraunhofer Institut für offene Kommunikationssysteme (FOKUS) and used during the test runs in chapter 5. In the following section the components of the test vehicles important for this work are outlined.

3.2.1 Hardware

Each of the vehicles is equipped with a low-power VIA C3 Nexcom board which runs a Red Hat Linux distribution. The board is connected with several car systems like the navigation system or webcams over standard issue interfaces. Diagram 3.3 show the hardware components and their interfacing methods.

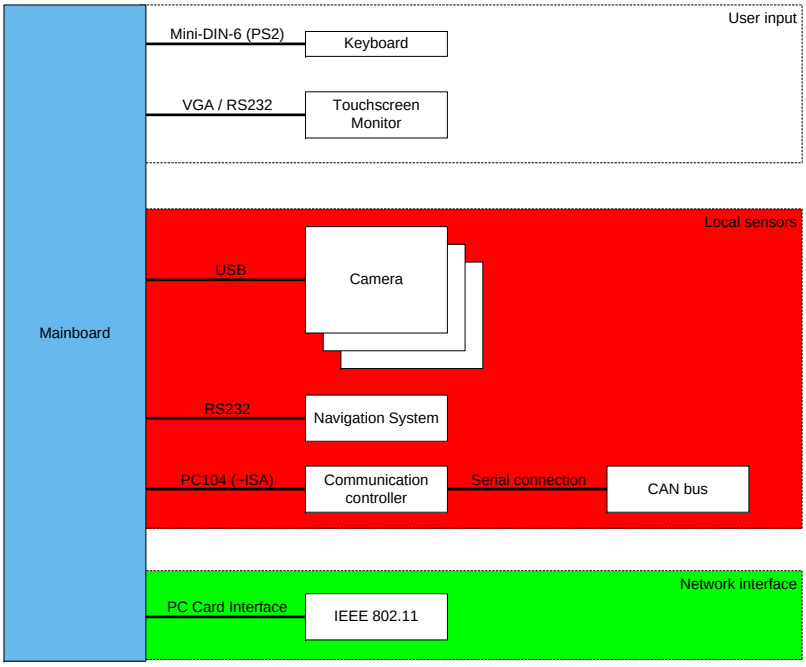

Figure 3.3: Mainboard with interfaced components

The Controller Area Network (CAN) bus interface is based on Philips SJA 1000, Intel 82527 and Siemens/Infinion 81C91 controller chips, for which a Linux driver was implemented. It delivers the CAN data via IP packets[1] to the system. The navigation system is a Blaupunkt TravelPilot DX-N and the cameras are standard USB webcams.

3.2.2 Network Architecture and Routing

The ad-hoc network was used as-is. The component important for this work is the router which was developed by University of Mannheim and NEC Network Labs Europe. The DC-EDR uses the standard IP interface provided by the FleetNet network layer.

The router provides an interface to the ad-hoc routing table describing the current state of the network neighborhood. The following information about network nodes in the vicinity can be obtained:

[1] to provide compatibility to a former implementation using a dedicated embedded controller connected via a 802.3 interface for data delivery.

- ID (which resolves to an IP address)
- Geographic location
- Direct one-hop relationship
- Validity

This information is accumulated through beacons broadcasted by every node in a fixed interval. In case of the FleetNet router the beacon is sent every 3 seconds.

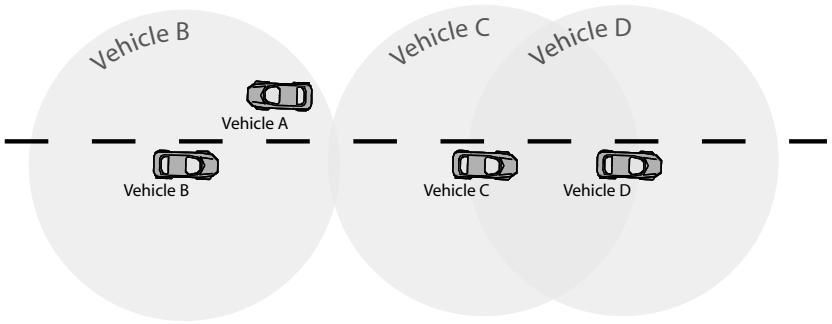

Figure 3.4: Beacon packets sent out by the FleetNet router

In figure 3.4 vehicle A had direct contact with vehicles C and D at some point t_0. Now we look at $t_1 = t_0 + t_b$, with t_b being the beacon interval.
The routing table of vehicle A holds the following entries:

Vehicle	Direct relationship	Validity
B	1	1
C	0	1
D	0	0

Table 3.1: Routing table entries of vehicle A

The beacon packet of vehicle B was received by vehicle A putting it in direct (one-hop) relationship with vehicle A. The beacon of vehicle C on the other hand was not received since t_0 therefore making it a valid but not directly addressable neighbor of vehicle A. Vehicle D was not seen for $t \gg t_b$ making it unlikely to come into contact with vehicle A again. The entry is simply stored to provide information of the network topology to the routing protocol.

3.2.3 Middleware Components

The middleware provides an API for applications to access interfaced components in a standardized way. In case of the FleetNet middleware these components are real hardware like the CAN bus or the navigation system as well as software like the FleetNet router.

The central component of the middleware is the event bus. This bus integrates different manager components which provide persistence functions and regulate the flow of data to and from the bus thereby adding additional functionality to the raw event bus. The hardware components are connected via `Adaptors` (similar to device driver functions) whereas `Servers` provide an interface to the outside world. Figure 3.5 gives an overview of the components. The middleware is also called `InCarMiddleware`.

Managers

The managers along with the `PropertyBroker` make up the intelligent event bus. They provide functions for persistence and for handling data objects travelling the bus. These objects are called `Properties` and define the standard data format used throughout the system.

The `PropertyBroker` receives changes of `PropertyObjects` and keeps track of their current value while providing notification events for components that have registered themselves for property changes. Every component of the middleware interfaces either directly (trusted components) or indirectly via an access control component (untrusted components) with the `PropertyBroker`. In addition the broker provides functions to describe the retrieved `Properties`. The event bus is implemented as an optimized hash table with handles.

Two other components of the event bus are `ManagerNonVolatiles` and `ManagerPersistence`. These two managers are used upon initialization of the middleware. With the vehicle having certain fixed attributes like e.g. its dimensions or chassis number `ManagerNonVolatiles` restores these values into the middleware.

After initialization of the fixed attributes the `ManagerPersistence` restores the last known state of non-static attributes into the middleware. The difference between these two types of variables is that an updated value of a persistent variable will be saved upon shutdown of the middleware by the `ManagerPersistence` back to the file system while non-volatiles will not. Variables that qualify for this category are e.g. the current position of the car or the mileage of the car.

`ManagerAccess` provides access control filters for the `Server` components. Clients connecting to a `Server` register themselves with their current security level. With every `Property` in the `PropertyBroker` being associated with distinct permissions for every security level, the `ManagerAccess` filters out data not cleared for operations of that specific client. In its current implementation four security levels are defined which have non-inclusive permission settings[2].

- World is the default access group
- Trusted is reserved for local or remote clients with a trusted status
- Owner access is given to the owner of the car.
- Manufacturer access.

Every group defines distinct access permissions for read and write operations and the visibility of every `Property` to members of the group.

[2] comparable to the unix file permission scheme

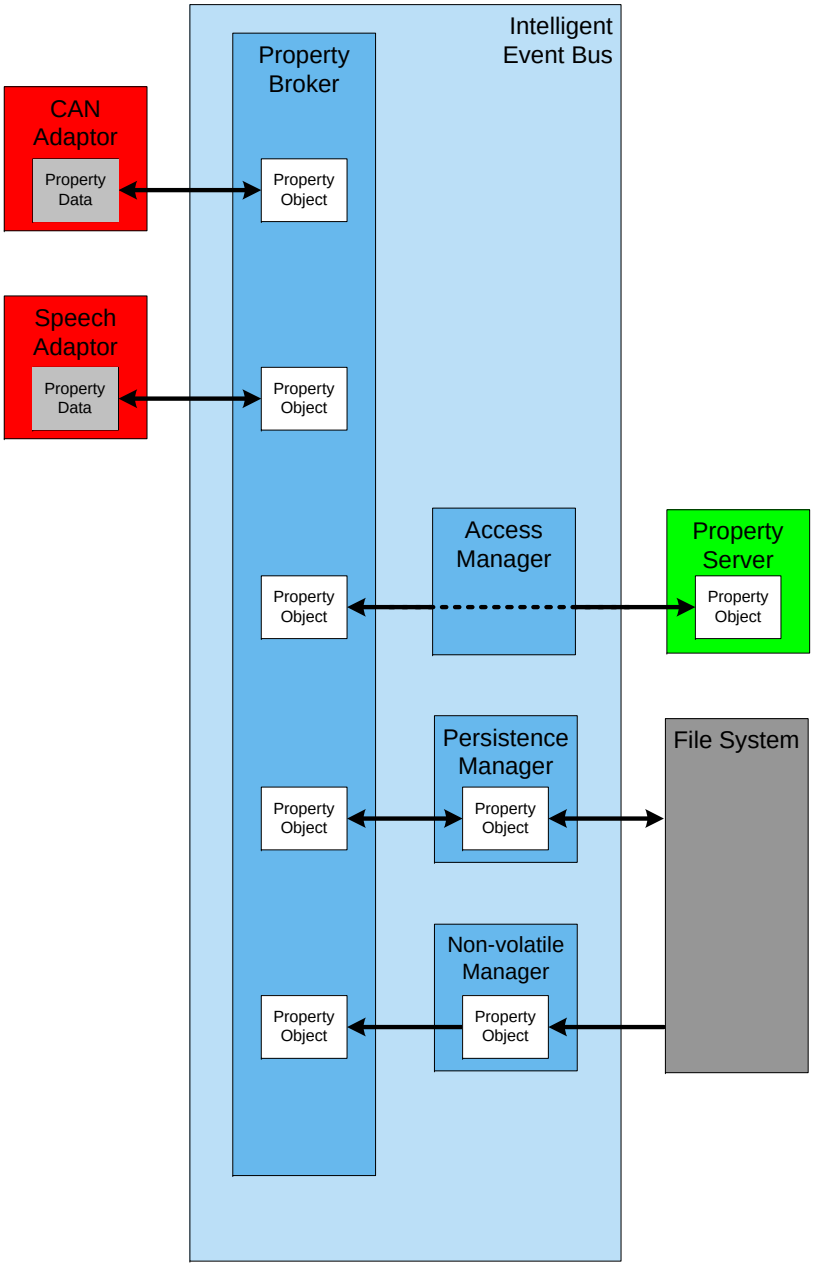

Figure 3.5: The core components of the middleware

Properties and Naming Scheme

As mentioned before all data traveling the event bus is encapsulated in `Properties`. These properties hold name, value and attributes of all information provided by the `Managers`, `Adaptors` and `Servers` of the system. Every `Property` is divided into two objects:

The PropertyObject (PO) holds the current value of the `Property`, whereas the PropertyInfoObject (PIO) provides descriptive information about the `Property`. The following example illustrates this for one `PropertyObject` provided by the CAN bus `Adaptor` for `car.speed.current`:

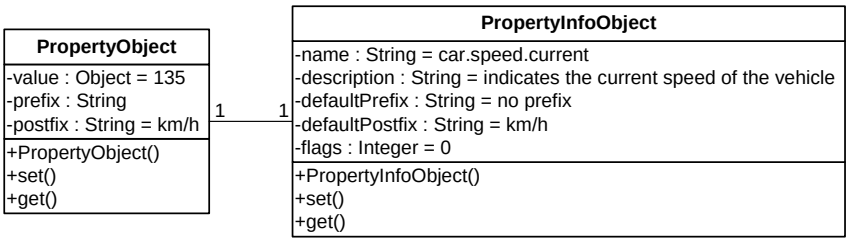

Figure 3.6: Example of a PropertyObject and its acoompanying PropertyInfoObject

The information provided by the PO only makes sense when combined with its PIO which provides the semantics for the value held by the PO. In addition to that, the PIO defines the access permissions for that `Property`. This separation is necessary since the `PropertyBroker` has to handle a high volume of `Property` value changes which should have the least possible performance impact.

The properties are organized into a tree structure. The used naming scheme has the following top level elements:

Root	Usage and example
service.*	Holds information about the `Servers` e.g. service.edr.description, service.edr.port
adaptor.*	Holds information about the `Adaptors` e.g. adaptor.edr.state.current
car.*	Holds information about the car, the car environment, passengers and payload e.g. car.passenger.max, car.navigation.position.current.longitude

Adaptors

`Adaptors` serve as an abstraction layer for hardware components or software subsystems. They have unrestricted bidirectional access to the event bus as they implement interfaces to trusted components such as the CAN bus or the navigation system.

The `CANAdaptor` implements the interface to one or more CAN busses of the vehicle. All information produced by CAN connected components such as the body electronics (e.g. turn signal) are pushed onto the event bus by this

Adaptor. With all Adaptors providing bidirectional access to their attached hardware, it would be possible to write data to the CAN bus. In contrast to the CANAdaptor which mostly pushes information onto the event bus, the AdaptorSpeech queries information for voice output. The basic idea is that it listens to certain PropertyObjects on the event bus and reports them via a text-to-speech subsystem to the driver of the vehicle.

Another important Adaptor is the AdaptorFleetNet which interfaces to the router and translates information about the vicinity of the car (see section 3.2.2).

Servers

Servers are used to enable communication with remote clients. Since the access level of a client accessing a specific Server can vary, each Server is accessed through the ManagerAccess.

The PropertyServer is a perfect example to illustrate a typical Server behavior.
When a remote client is interested in the current state of the vehicle it registers itself to the PropertyServer service and subscribes to certain Properties to receive updates on a value change. When the Property value changes, the PropertyServer pushes the new value to the registered clients. The role of ManagerAccess is clear in this scenario since specific information has to be restricted to certain groups. The world group for example should not be able to access the place of departure or travel destination provided by the navigation system of the vehicle, whereas this information could be anonymously queried by traffic management systems which might be members of the trusted group.

3.3 EDR Components

With the middleware being the central hub for all information from the various car subsystems one requirement of the work was to integrate the prototype of the DC-EDR into the existing middleware, therefore the architecture of the DC-EDR follows the categories laied out in the previous chapter.

3.3.1 Manager

The first component that gets called upon initialization of the DC-EDR is the EDRManager. Its purpose is to enumerate the properties for which recording will be enabled. For this purpose the PIO format has been extended by an DC-EDR flag which is read by the manager. The advantage of integrating this information directly into the PIOs is, that every component introducing a new Property to the middleware can decide whether it should be recorded by the DC-EDR upon crash or not. This mechanism enables the automatic recording of new information without the need to change the configuration of the DC-EDR.

After the information, which POs to record is gathered, the manager hands over the control to the EDRAdaptor.

3.3.2 Adaptor

The `EDRAdaptor` implements the main functionality of the DC-EDR. This includes the ring buffer and the local and remote crash notification of the DC-EDR.

Ring Buffer

The ring buffer is the central component of the DC-EDR. Its purpose is to continually store data into a volatile buffer that will be dumped upon a crash event into non-volatile storage. This ring can easily be simulated by a linear datastructure with a pointer advancing with time and being reset to position 0 when the maximum recording time has been reached. That way the last x seconds of data are preserved since the earliest recorded data is always overwritten by the most recent one. This continual recording is started upon instantiation of the `EDRAdaptor` and kept up until the shutdown of the middleware.

Figure 3.7 shows the schematic flow of data from the source to the ring buffer.

The `EDRAdaptor` registers a listener for every event that was identified by the `EDRManager` for recording by the DC-EDR. When an `Adaptor` pushes a new PO onto the event bus, the `PropertyBroker` notifies all subscribed listeners of that change including the `EDRAdaptor`. The `Adaptor` retrieves the PO, creates a new DatedObject (DO) wrapper and stores the PO inside. The wrapper object is used to hold timing information about the data. Then the DO gets pushed into a queue. While new data is being stored in one queue the second queue is being copied into the ring buffer. This mechanism allows decoupling of the event bus from the ring buffer.

Data Dump

When a crash event is detected by the `EDRAdaptor` the current available data (ring buffer and video data) is copied to non-volatile memory.

The recording of events takes place until a predetermined time after the crash (post-crash time). Therefore the dump thread will begin saving pre-crash data. When he reaches the position of the recording thread it has to lag behind this thread which deferrs the completion of the dump until the post-crash recording time has been reached. A schema of the dump thread time-lag can be seen in figure 3.8.

In addition to the data from the ring buffer, video data from the webcams is stored on non-volatile memory. The video buffer is implemented as an external software component and is not an integral part of the DC-EDR.

Figure 3.9 shows the schematic flow of information when a crash is signaled to the middleware by putting the PO `car.edr.event` with the value "crash" on the event bus. A dump signal is sent to the ring and video buffers and a remote notification is sent out to all available DC-EDRs in the current one-hoc neighborhood provided by the FleetNet router.

Remote Notification

The network component of the DC-EDR consists of two functional parts: sender and receiver. The sender is attached to the `EDRAdaptor` and transmits a trig-

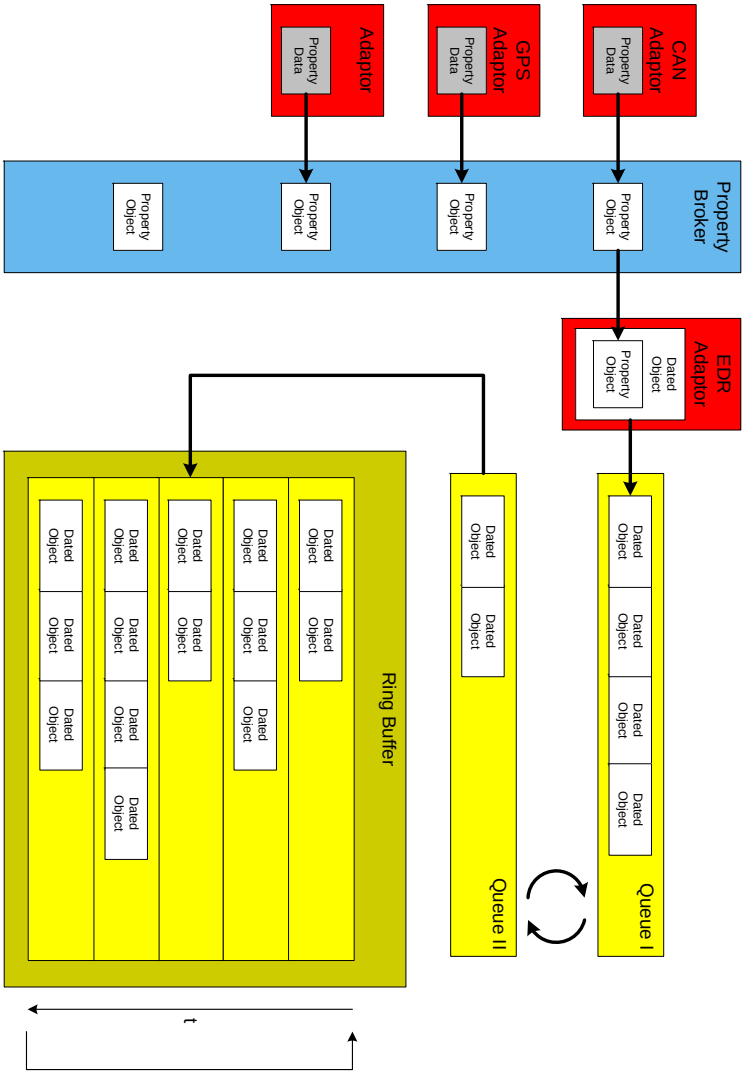

Figure 3.7: Data flow from the adaptors to the ring buffer

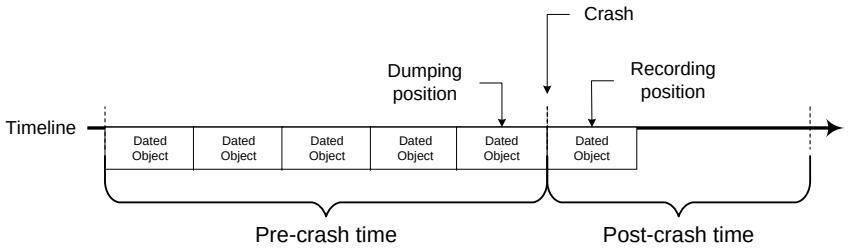

Figure 3.8: Dump thread lagging behind the recording thread

ger to remote DC-EDRs when a local crash condition is met. The receiver is implemented as a `Server` component and will be outlined in the next section.

For the sender to know the state of the current ad-hoc neighborhood it queries the middleware for the latest router information. The retrieved table (see figure 3.1) indicates if a node is currently in the one-hop neighborhood of the vehicle. If this is the case, that node is notified of the crash.

3.3.3 Server

The receiving side of the DC-EDR is located inside the `EDRServer`. This component was separated from the `EDRAdaptor` because it provides an open connection to the VANET therefore allowing attacks from malicious nodes. Access to the `PropertyBroker` through this `Server` is restricted by the `ManagerAccess`.

The `EDRServer` waits for crash triggers sent from remote DC-EDRs. When a trigger is received it is sent to the local event bus. From here on the remote trigger behaves like a local trigger firing up the crash cycle of the local DC-EDR. In addition, an acknowledge is sent back to the originating vehicle notifying it of the reception of the trigger and the storage of the data.

Crash trigger and acknowledge packet are wrapped in DOs that provide local timing information. Upon arrival on the other side they are wrapped into DOs once again. This process is important, because it enables the log synchronization outlined at the beginning of this chapter. In addition in allows the mutual identification of the vehicles involved in the transaction, since the DOs hold information about their point of origin. In figure 3.11 a complete notification cycle is shown along with the wrapping process.

After the acknowledge from the remote DC-EDR is received we have two synchronizable logs on the local and remote vehicles.

3.3.4 DC-EDR States

The DC-EDR cycles through different states during its operation. Each state defines the activity of certain components of the system.

During initialization of the system the `EDRManager` determines the variables to be recorded and prepares this information for the `EDRAdaptor` which gets initialized when the manager gives the read signal to the middleware. The *recording* state is the normal operation state for the system. In this state data is continually recorded into the ring buffer.

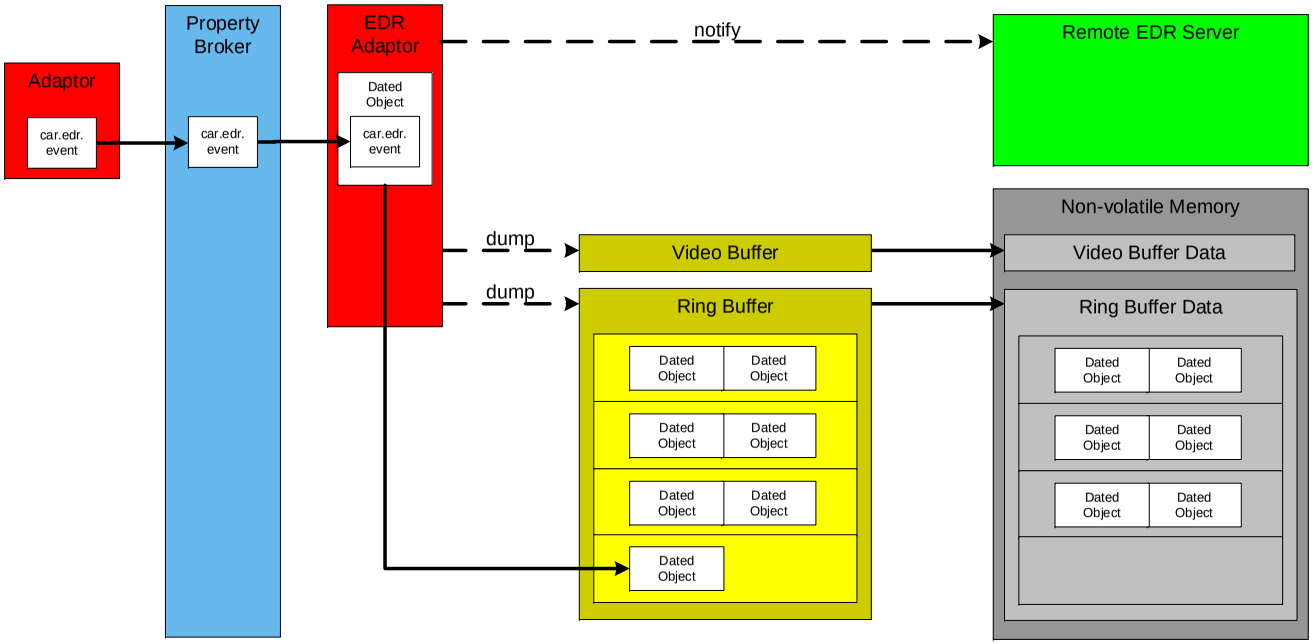

Figure 3.9: Data flow in a crash event

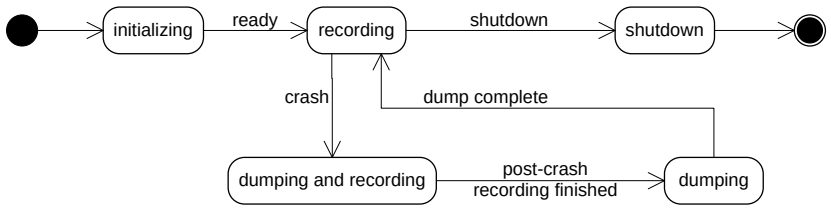

Figure 3.10: Operating states of the EDR

When a crash trigger is detected a state change to *dumping and recording* is issued which starts up the `DumpThread` to store the contents of the ring buffer into non-volatile memory with the recording thread still running. When the recording has reached the defined post-crash recording time the state is advanced to *dumping* and the recording thread is put into queuing mode, which stores new data from the `Adaptors` into temporary memory and leaves the ring buffer untouched. The dump thread finishes storing the data, and the recording thread is put back into normal operations mode. This returns the whole system back to its normal operation state *recording*.

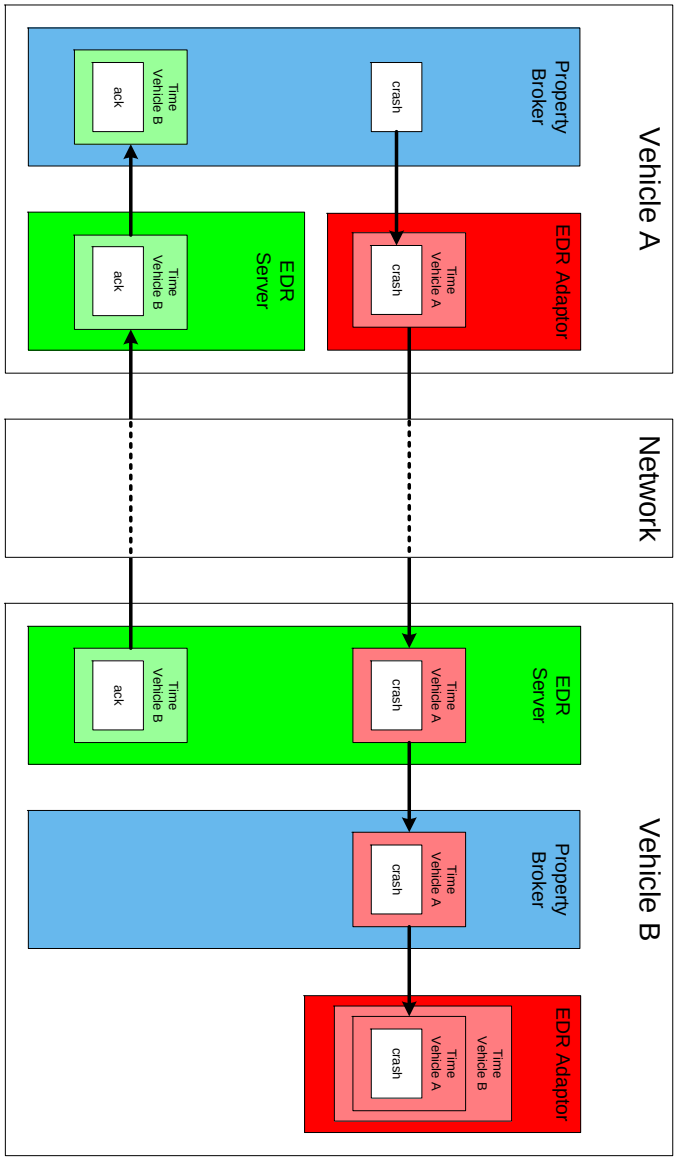

Figure 3.11: Crash notification cycle and PO wrapping

Chapter 4

Implementation details

This chapter will not dive into every detail of the implementation, but highlight some features which were crucial to the implementation of a system operating under millisecond timing requirements. The DC-EDR was completely written in Java[1], because it had to be integrated into the module structure laied out by the middleware.

Despite running on a non-realtime Linux Operating System (OS) the timing requirements for an EDR had to be adhered and measured. The first section will focus on the measurement method used throughout the system. The additional timer data had to be stored in a new container, which is introduced in the next section. Following the storage method and post-crash analysis tool for these objects. The final section talks about optimizations of the code.

4.1 Measuring Method

To provide a millisecond accuracy the measuring method had to be more accurate than that. The use of `System.currentTimeMillis()` did not prove suitable because of its low granularity. The test machines running the Windows OS achieved only an average accuracy of 15 ms, while the test machines with Linux OS (including the OSs on the test vehicles) achieved an average accuracy of 1 ms.

In the first tests conducted with high frequency data this timer returned 0 as the time difference between 2 samples with a sampling rate above 1 kHz.

Nevertheless this timer also called unix or system timer[2] can be used to provide an absolute marker for the DC-EDR log files. It has to be noted, that the system time may be inaccurate if not synchronized with a central time server, but can provide a "macro" identification of crash events (hour and date of the event).

The approach used for sub-millisecond measurement is based on a Java method inside the `sun.misc.Perf` package of Java 1.4.x. It allow the query of the Timestampcounter (TSC)[3] ticks which provide a much higher granular-

[1] currently Java Standard Edition Version 1.4.x.

[2] and delivers the time difference in milliseconds between the current time and January 1, 1970

[3] depending on the platform other available timers will be queried

ity than the system time. The TSC provides the current ticks of the CPU, which, divided by the frequency, gets us a microsecond accurate timer, called high resolution timer in the following chapters.

Since the Java implementation was not officially documented, it was tested against a C implementation using the standard Windows API functions `QueryPerformanceFrequency` and `QueryPerformanceCounter` (JNI developed by Vladimir Roubtsov [Rou]), which proved the granularity of 1 μs.

The downside of this high resolution timer was that it did not provide an absolute marker, since it is started upon instantiation of the Java virtual machine.

Since the method for log synchronization only required difference times, the two timers simply were combined, by associating them with every PO that was put inside the ring buffer of the DC-EDR. This provided an absolute but inaccurate time stamp and relative but accurate time information for every PO.

For further details about accuracy of timers, see [Sti05] and [PB05].

4.2 Data Format and Serialization

In the current implementation of the middleware the POs used throughout the system do not provide any timing information about the data included in them, therefore a wrapper had to be implemented to host the POs as well as the system time and high resolution time (see figure 4.1 for details).

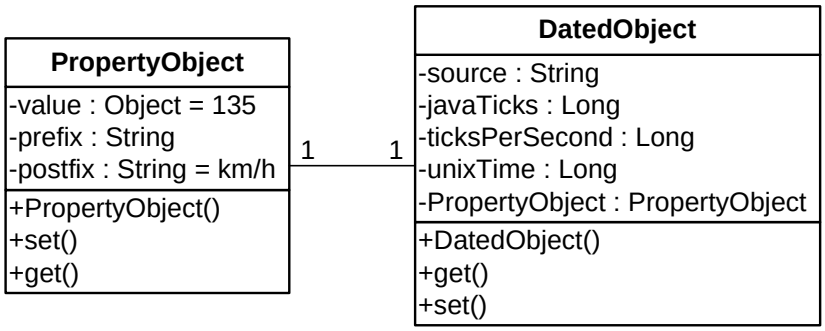

Figure 4.1: DatedObject: A wrapper for timing information

The `ticksPerSecond` holds the frequency of the current high resolution timer, with which the `javaTicks` are divided to get a timestamp which is now based on microseconds rather than ticks.

With the DC-EDR receiving remote crash notifications from other vehicles the origin of the message had to be differentiated. This is the purpose of the field `source` which, in the current implementation simply stores the IP address of the entity which sends information to the DC-EDR. In case of the DC-EDR operating in standalone mode with no vehicles in the vicinity, it is the local IP address.

These DOs are serialized upon a crash notification to the local hard drive using the XML codec provided by the standard Java API. The XML format

was chosen because of the simple customization of the serialized data format (see [Mil] for details), and because of the good readability of the raw data logs. The following XML fragment would represent the DO from figure 4.1.

```
<object class="de.gmd.fokus.vehicle.DatedEventTicks">
  <string>car.speed.current</string>
  <void property="source">
    <string>192.168.17.1</string>
  </void>
  <void property="javaTicks">
    <long>112913442</long>
  </void>
  <void property="ticksPerSecond">
    <long>1000000</long>
  </void>
  <void property="unixTime">
    <long>1117616370636000</long>
  </void>
  <object class="de.gmd.fokus.vehicle.PropertyObject">
    <double>135</double>
    <void property="postfix">
      <string>km/h</string>
    </void>
  </object>
</object>
```

Listing 4.1: PropertyInfoObject serialized as XML

4.3 Post-crash analysis

With the data being stored in log files they can now be retrieved and analyzed post-crash. To simplify the analysis of the data whose XML representation often exceeded several megabytes during one test run, a tool was written that parses the serialized DOs and sorts them by property name and time of occurrence. The data is then displayed graphically[4] as well as in a spreadsheet for detailed analysis.

This tool (see figure 4.2 for a screenshot) was also used to analyze the data for the validation of the prototype.

4.4 Optimizations

The ring buffer being the core component of the DC-EDR had to be optimized for processing speed since it should be able to store PO with a sustainable data rate of 1.5 new objects per millisecond. For this reason an `java.util.ArrayList` was used which provides fast insert times and has native support for conversion into an array of static length for further processing. For details about the performance of different Java collection types see [Eck02] chapter 11.

[4] with the use of the Ptplot package [Ptp]

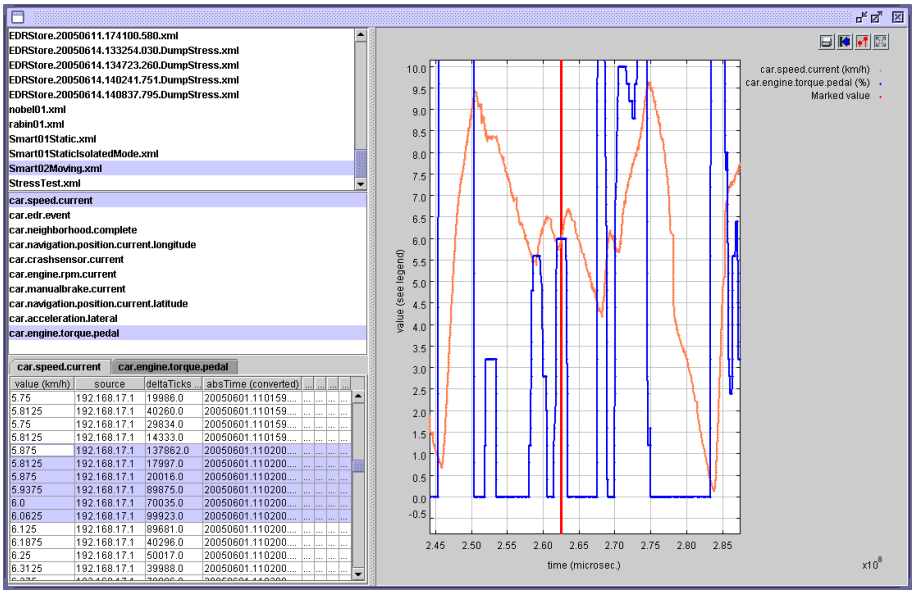

Figure 4.2: Post-crash analysis tool screenshot

The two queues (figure 3.7) are used to decouple the queuing of incoming events from the deposit into the ring buffer. The "bucket" metaphor is suited to describe this method. While the first bucket gets filled with incoming data the second one is emptied into the ring buffer. After a specified time the roles are reversed and the now empty second bucket takes place of the first one. Thus a bucket holds all events occurring within a given time window.

This method provides an additional buffer if the dumping of events is deferred. Due to performance and handling considerations the data structure used for the ring buffer is an array of fixed length. With the two queues having a defined fill time we can instantiate the array upon initialization of the `Adaptor` and leave it untouched during the whole runtime of the DC-EDR. The only variable is the length of the queues which varies depending on the event bus traffic and number of subscribed POs.

Additionally, during normal operation of the DC-EDR no instantiation of new objects or deep copies are made until absolutely necessary. This is the case twice:

- when a PO scheduled for recording is generated, a new DO is instantiated
- when one queue is emptied into the ring buffer

Since the deep copy of a queue into the ring buffer might block the normal operation of the DC-EDR considerably this step is sourced out into a seperate thread called `RecordingThread`. The dumping of the ring buffer upon crash notification is handled by the `DumpThread` and the storage of the video buffer by the `CamThread`. Currently the video buffer is an external component which receives a notification from the ring buffer and saves the current image from the front webcam to the local hard drive.

Recalling the different operating states of the DC-EDR we already said that they define the activity of a certain component of the system. Figure 4.3 maps the main operating states to thread activity while providing an overview of interthread messaging.

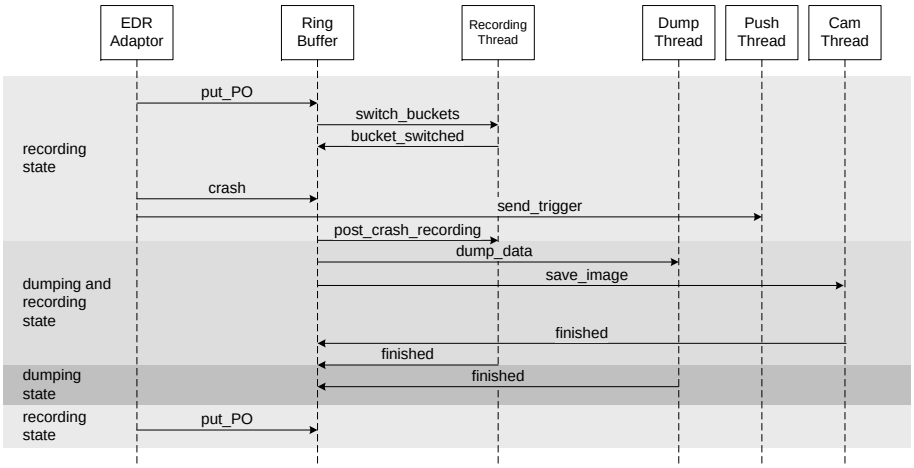

Figure 4.3: Interthread messaging and recording states

During the *recording* state POs are put into the ring buffer and the queues are regularly switched after a fixed amout of time. When a crash occurs, the ring buffer simply issues a couple of commands to the supporting threads and returns to his task of buffering new incoming POs from the `EDRAdaptor`. The `PushThread` is used to send the crash trigger to the nodes in the one-hop neighborhood. When the threads have finished their tasks, they report back to the ring buffer. This concludes a crash cycle and returns the DC-EDR to its normal *recording* state.

Chapter 5

Validation of the Prototype

The validation of the prototype will test different aspects of the implementation.

First the basic operation of the DC-EDR will be checked to guarantee the most basic functions like recording of data and trigger transmission. After that several tests focusing on different timing aspects will be made. Special attention is given to the conformance towards the IEEE Standard requiring a certain resolution and sampling rate from the recorded data. Since the DC-EDR operates in a distributed environment a series of tests shall prove the synchronizability of log files recorded by multiple vehicles during a crash. The chapter will be concluded by an operational test of the image capturing capabilities of the DC-EDR.

The preparation of the test runs proved to be very time consuming, since the complete middleware had to be deployed and functionally tested on both vehicles before test data could be gathered. The preparation of the vehicles and the test runs sometimes took several days to complete, since the middleware and deployment method itself was a prototype developed during the FleetNet project.

In each test case the systems (vehicles or workstations) were based on the middleware and DC-EDR described in chapter 3.

To simulate crashes a custom `Adaptor` was written to generate crash triggers during the test runs. This `Adaptor` called `DataGenAdaptor` was implemented as part of the middleware and connects as a standard `Adaptor` to the event bus. The implementation is basically an array of timers that push different POs into the `PropertyBroker` in a preset interval thereby generating regular event patterns. The most important of these POs being the property `car.edr.event` which is used to signal a crash event.

5.1 Basic Operation

5.1.1 Test setup

The following data was recorded during a test run with 2 vehicles. Vehicle A had a static position (parked in a garage) while vehicle B was parked in a remote location with no radio contact to vehicle A. Vehicle B generated crash triggers in one minute intervals using the `DataGenAdaptor`.

Then vehicle B was driven towards vehicle A and parked beside it. During the approach of vehicle B radio contact was established and the following crash trigger activated the dumping of the DC-EDR log of vehicle A.

The operation parameters of the DC-EDR were changed in this test run to allow long term recording, with crash triggers being sent out every 60 seconds. The network link was provided by a standard 802.11a interface card with a fixed transmission rate of 2 Megabit (Mbit)/s to avoid link rate renegotiation delays during the test. This data rate was used during all vehicle based tests conducted in this chapter.

5.1.2 Results

In the red segment of graphs 5.1 and 5.2 the vehicles had no radio contact. In the green segment they had established radio contact and the routers had an entry of the opposite vehicle in their routing table.

The timelines in both graphs shows the absolute time in seconds counting from the start of the middleware. Since the middleware was not started on both vehicles at the same time, the timestamps in both graph and table are not synchronized across the two vehicles.

The blue graph in 5.1 shows the current speed (`car.speed.current`) of vehicle B. The two vertical markers in this graph indicate the time where a manually generated crash event took place. The events are summarized in table 5.1.

local timestamp (in $10^9\mu$s)	event
vehicle B (moving)	
1.50	starting to drive vehicle B towards vehicle A (speed > 0)
1.56	crash occurred with no vehicle registered in router (marker 1)
1.57	vehicle A registered in routing table (transition of red to green segment)
1.59	vehicle B was parked within communication range of vehicle A (speed = 0)
1.62	crash occurred with radio contact and acknowledge was received (marker 2)
vehicle A (parked)	
1.93	vehicle B registered in routing table (transition of red to green segment)
1.95	remote crash was registered and acknowledge was sent (marker 2)

Table 5.1: Basic operation: Summary of events

Since the x-axis marks the time from the beginning of the recording to the

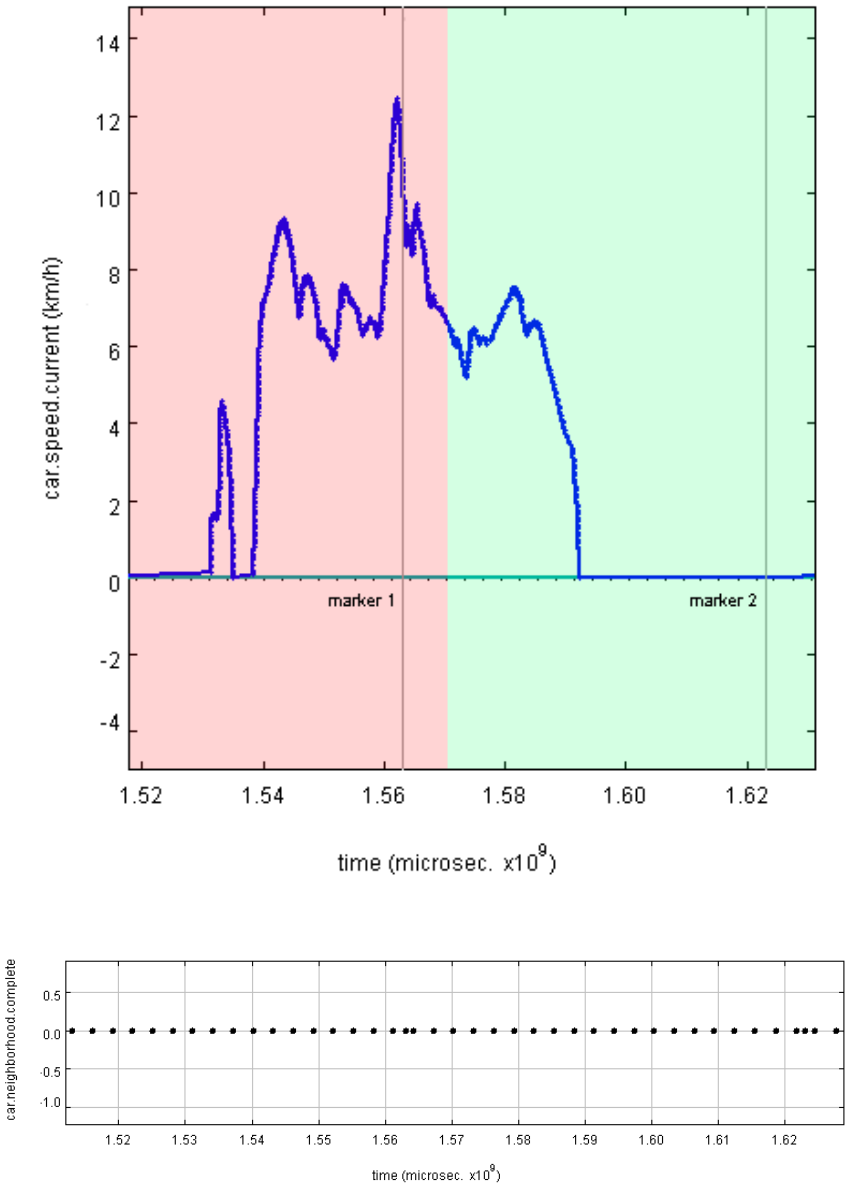

Figure 5.1: Basic operation: recorded events on vehicle B

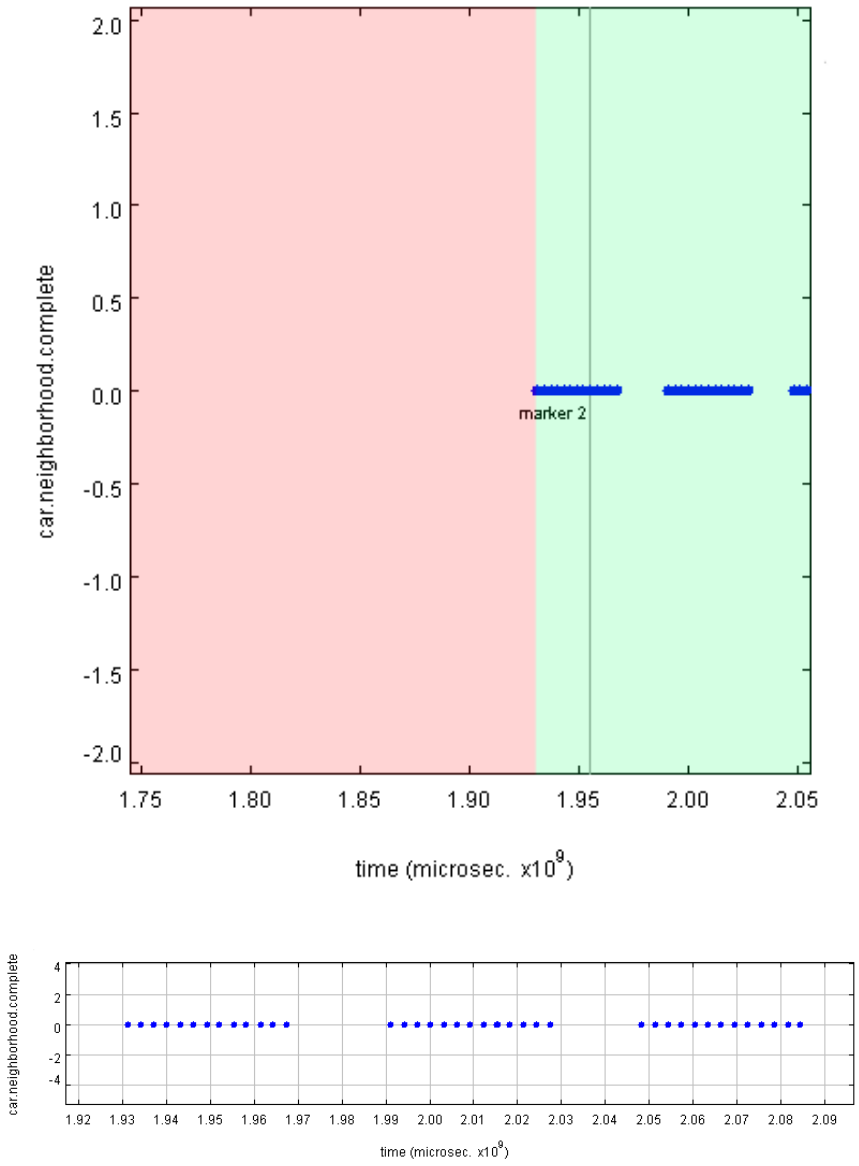

Figure 5.2: Basic operation: recorded events on vehicle A

current event in $\mu sec * 10^9$ the events of vehicle B took place between 1500 and 1630 seconds from the beginning of the recording.

Taking marker 2 as a pivot element, the time from the first appearance of vehicle A in the routing table in vehicle B to the crash is about 53 seconds. This displays the indeterministic behavior of the FleetNet router. On the opposite side this time is about 24 seconds. Subsequent tests confirmed this discrepancy as an effect produced by the router.

To explain the discrepancy, the recorded `car.neighborhood.complete` property comes in handy. This property records the reception of a beacon sent out by a remote router to announce the vehicle to its neighborhood. The beacons of vehicle A registered in the DC-EDR of vehicle B and vice versa.

While the router on vehicle A was sending out a beacon every 2.87 seconds (with a standard deviation of 0.54 seconds) during the whole test run, the router on vehicle B behaved differently. The time between two beacons transmissions was, like on vehicle B, about 2.76 seconds, but this standard interval was regularly broken by intervals with no transmission. These "silent times" had an average length of 25.4 seconds. This explains the deferred registration of vehicle B in the DC-EDR log.

Despite this discrepancy the test run showed the successful basic operation of the DC-EDR with the limitations introduced by the router. The recording of the routing table exemplary showed the application of the DC-EDR as a network probe. The next test will establish if the DC-EDR can handle the amount of data produced during normal operation in a vehicle.

5.2 Operation under Load

5.2.1 Test setup

To verify if the DC-EDR can handle the amount of data that will be generated by the `Adaptors` a dry test was performed using the `DataGenAdaptor`. Every 10 seconds a new high priority thread generating dummy `PropertyObjects` was started until a maximum of 6 threads were running. Every one of these threads generated 1 sample every 4 milliseconds. With all threads running 1.5 elements were generated per millisecond. During normal operation the number of data elements that have to be recorded by an EDR as defined by the IEEE Standard (see sampling rates in table 5.3) does not exceed this threshold.

This test should answer 2 questions:

- Do all generated `PropertyObjects` register in the DC-EDR log?
- In what magnitude does the standard deviation of the time stamps increase when the system is put under stress?

5.2.2 Results

While all generated POs registered in the log, the increase in generated POs had an impact on accuracy when reaching a threshold of about 0.75 objects per millisecond. Input / output operations, like the ring buffer being dumped had an additional negative impact on the accuracy (see table 5.2 for details). Subsequent runs produced similar results.

Threads or $\frac{PropertyObjects}{4ms}$	Mean (in μs)	Standard Deviation (in μs)
1	4008	458
2	4028	930
3	4112	5003
4	4162	5862
5	4178	7418
6	4087	2978
6 (with dump)	5187	17966

Table 5.2: Operation under load results

When the number of threads was decreased to 2 after it has reached 6 the deviation decreased to under 1 ms, with 1 ms being the necessary accuracy for this kind of device.

The result shows that despite multithreading and several optimizations (see chapter 4 for details) the architecture does not provide the necessary accuracy to fulfill the timing requirements outlined by the IEEE Standard if the amount of data exceeds a threshold of about 0.5 POs per millisecond.

The software architecture cannot provide the necessary accuracy required for this application domain, therefore either a dedicated software platform fulfilling real time requirements or a dedicated hardware platform has to be used to handle high volume data. This is especially true if the crash log includes video data, which requires the storage of large amounts of data.

5.3 Data Availability and Resolution Compared to the IEEE Standard

As we have seen in section 2.2.1, traditional event data recorders log a number of sensory data. The tests in this section will validate if the prototype can record this data at all and if so, if it can record the data in the necessary resolution and accuracy set by the IEEE Standard. As we have seen in the previous chapter, the recording accuracy is influenced by the amount of data that has to be handled. These results have to be kept in mind when observing the results of this section.

First these elements are listed along with their data boundaries and then they will be compared against the prototype.

5.3.1 Data Elements for Light Vehicles

The following tables are a compilation of the data elements defined by the IEEE Standard.

Required Elements

The elements in table 5.3 list the data elements required by the IEEE standard along with their recording interval, resolution, sampling rate and sampling accuracy.

Additional Elements

The elements listed in Table 5.4 are not required by the standard but if the vehicle is equipped with measurement methods for these elements, it is recommended to record them. This table is divided into two sections. The first section lists the optional elements while the second part presents data elements which were additionally recorded by the DC-EDR but not optioned by the IEEE Standard.

5.3.2 Comparison with Prototype

Now that the requirements are established they can be compared against the data collected in the test runs. This is done for the recommended data elements and the elements additionally implemented by the DC-EDR. With the computing environment of the DC-EDR being able to store a multitude of the recommended recording interval of 13 seconds, this aspect is of minor interest. As is the data format. Since the format used in the DC-EDR log files is more elaborate, it can be broken down into the format set by the standard. Special attention is payed to resolution and sampling rate of the data elements.

Data Elements Available on the Test Platform

Table 5.5 shows the mapping between the required IEEE data elements, their corrosponding `PropertyObject` names, the resolution as well as their sampling rates. The CAN sampling rate was taken out of the CAN communication matrix[1] for that vehicle while the measured sampling rate is the average recorded sampling rate over several test runs with a pool of 2000 samples for `car.engine.rpm.current` and accordingly more samples for data with a higher frequency.

The last three columns of the table are color coded to allow identification of elements that fulfill the IEEE standard (green) and elements that do not (red).

Unfortunately not all necessary information could be gathered. One important property, the longitudinal acceleration is not provided by CAN attached devices like e.g. the airbag module. On the other hand it must be said that the CAN communication matrix available for this work did not allow complete identification of all data on the CAN bus.

[1] The CAN communication matrix provided by DaimlerChrysler lists detailed information about the CAN data along with the resolution and sampling rate of every message on the CAN bus

Data element	recording interval	resolution	minimum sampling rate	accuracy
Acceleration x-axis (pre crash)	-8s to +5s	0.048 g_n [a]	10 Hz	$\pm 0.01 g_n$
Acceleration x-axis (crash)	-0.1s to +0.5s	0.048 g_n	1000 Hz	$\pm 0.01 g_n$
Maximum Δv	computed	0.048 g_n	once per event	$\pm 0.01 g_n$
Vehicle indicated speed	-8s to +5s	1 km/h	-	$\pm 1 km/h$
Engine RPM	-8s to +5s	0.25 RPM	5 Hz	$\pm 1\%$
Engine throttle	-8s to +5s	0.4%	4 Hz	$\pm 1\%$
Service brake	-8s to +5s	on/off	2 Hz	N/A
Ignition cycle at crash time	N/A	N/A	once per cycle	N/A
Ignition cycle at download time	N/A	N/A	once per cycle	N/A
Driver safety belt status	-8s to +5s	engaged/disengaged	10 Hz	$\pm 10 ms$
Air bag warning lamp (once for every airbag)	-8s to +5s	on/off	N/A	last 10 counts
Air bag deployment level (used for multi-state air bags)	-8s to +5s	by manufacturer	by manufacturer	$\pm 1 ms$
Air bag time to 1-deploy (if the airbag only has 1 level this denotes the time for deployment)	-8s to +5s	1 ms	once per event	$\pm 1 ms$
Event number for multi-events	N/A	1/2/3	once per event	N/A
Time between events	-8 to +5s	1 ms	N/A	$\pm 1 ms$
Complete file recorded	N/A	yes/no	once per event	N/A

Table 5.3: Required data elements defined by the IEEE standard

[a] with acceleration due to gravity having a standard value of $g_n = 9,80665 \frac{m}{s^2}$ we get for $0.048 g_n = 0.471 \frac{m}{s^2}$

Data element	recording interval	resolution	minimum sampling rate	accuracy
Acceleration y-axis		values equivalent to x-axis acceleration		
Acceleration z-axis		values equivalent to x-axis acceleration		
Vehicle roll angle	-8s to +5s	10∘	-	±10∘
ABS activity	-8s to +5s	active/not active	2 Hz	N/A
Stability control status	-8s to +5s	engaged/not engaged	2 Hz	$\pm 10ms$
Steering wheel angle	-8s to +5s	1∘	10 Hz	±1∘
Passenger safety belt status	-8s to +5s	engaged/disengaged	10 Hz	$\pm 10ms$
Air bag suppression switch status	-8s to +5s	on/off/auto	1 Hz	N/A
Air bag time to n-deploy (once for every airbag)	-8s to +5s	1 ms	once per event	$\pm 1ms$
Side air bag time to deploy (once for every airbag)	-8s to +5s	1 ms	once per event	$\pm 1ms$
Side curtain/tube air bag time to deploy (once for every airbag)	-8s to +5s	1 ms	once per event	±1 ms
Pretensioner deployment, time to fire (for driver and right front passenger)	-8s to +5s	1 ms	1000 Hz	±1 ms
Seat position (for driver and right front passenger)	-8s to +5s	1.25 mm	once per event	$\pm 1.25mm$
Occupant size classification (for driver and right front passenger, records weight and size)	-8s to +5s	by manufacturer	by manufacturer	N/A
Occupant position classification (for driver and right front passenger, no description found)				
GPS longitude and latitude	-8s to +5s	0.0001’	1 Hz	$\pm 0.06'$
Trigger event date	-8s to +5s	1 day	once per event	±15 min
Trigger event time	-8s to +5s	1 s	once per event	±1 s

Table 5.4: Optional elements defined by the IEEE standard

Data element	PropertyObject name	resolution	sampling rate CAN	average rate measured
Acceleration x-axis	-	-	-	-
Maximum Δv	-	-	-	-
Vehicle indicated speed	car.speed.current	0.0625 km/h	100 Hz	21,97 Hz
Engine RPM	car.engine.rpm.current	25 RPM	10 Hz	8.7 Hz
Engine throttle	car.engine.torque.pedal	0.4%	100 Hz	36.8 Hz
Service brake	car.manualbrake.current	on/off	50 Hz	0.14 Hz
Ignition cycle at crash time	-	-	-	-
Ignition cycle at download time	-	-	-	-
Driver safety belt status	-	-	-	-
Air bag warning lamp	-	-	-	-
(for every airbag)				
Air bag deployment level	-	-	-	-
(used for multi-state air bags)				
Air bag time to 1-deploy	-	-	-	-
(if the airbag only has 1 level				
this denotes the time for deployment)				
Event number for multi-events	N/A	timestamp	once per event	
Time between events	N/A	1 μs	once per event	
Complete file recorded	N/A	yes/no	once per event	
Acceleration y-axis	car.acceleration.lateral	0.1 $\frac{m}{s^2}$	100 Hz	33.7 Hz
GPS longitude and latitude	car.navigation.position.current.*	0.000001'	N/A	1 Hz[a]
Trigger event date	car.edr.event	1 day	N/A	N/A
Trigger event time	car.edr.event	1 ms	N/A	N/A

Table 5.5: Availability of properties in the test platform middleware

[a]the polling interval for the Blaupunkt GPS system used

With the acceleration rates being the most important data for post-crash analysis the deployment system has to be equipped with the necessary sensors that provide a sampling rate high enough to fulfill the standard. If we assume, that the CAN bus would provide data for longitudinal acceleration with the same frequency as for lateral acceleration[2], the provided sampling frequency of 100 Hz would not suffice anyway.

Though the CAN bus provided a data element named "crash signal from airbag" which was recorded and could be used to trigger the DC-EDR.

The sampling rate for `car.manualbrake.current` was obviously misstated in the CAN matrix since the value only registered either when the brake was released, or with the vehicle in motion with a maximum delay of 30 seconds between each sample.

The measured data rates show that in all of the recorded properties there are severe discrepancies between the sampling rates provided by the manufacturers CAN communication matrix and the measured sampling rates. With the sum of the recorded data being well within the processing threshold determined in 5.2 the middleware should be able to process the average of 0.5 samples per millisecond without losing any samples.

To determine the source of error, the time difference of two consecutive POs was examined. The `car.engine.rpm.current` property should, according to the CAN communication matrix, deliver a sample every 100 ms whereas the test results showed an average time difference of 115 ms. Graph 5.3 shows the measured time differences between two samples n and k with $n = k + 1$.

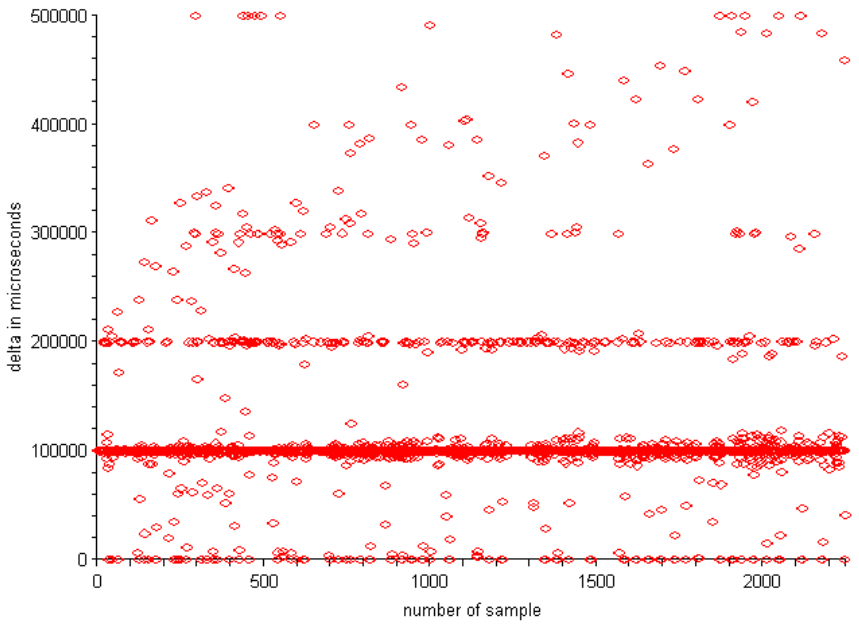

Figure 5.3: recording time stamp deltas for car.engine.rpm.current (all samples)

[2]since the sampling rates of any data on the CAN bus did not exceed 100 Hz, this is most likely the case

As we can see, the majority of samples had in fact the promised time difference of 100 ms to its previous sample. With samples having both positive and negative variances, the errors can be explained by 2 effects. In graph 5.4 we zoomed in on 161 successive samples of graph 5.3.

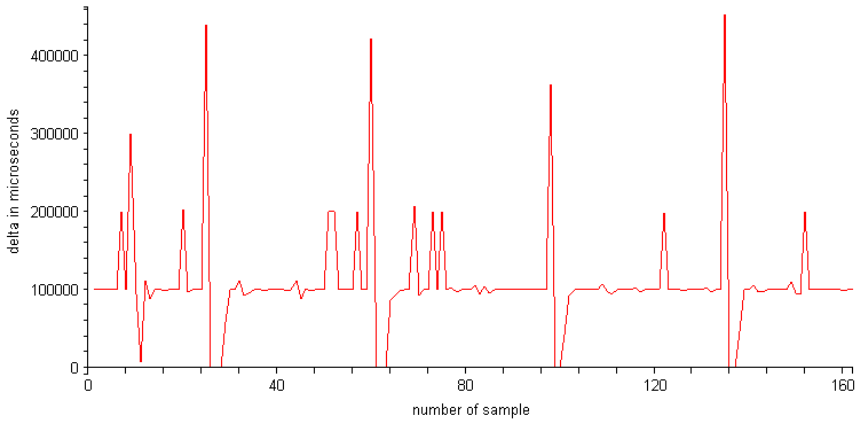

Figure 5.4: recording time stamp deltas for car.engine.rpm.current (samples 1558-1719

The first effect can be seen at sample position 26 or 61. Each time one sample breaks out with a positive deviation several following samples have a negative deviation with a time difference of nearly 0 in respect to each other. This is an effect of the middleware buffering queue inside the `PropertyBroker` which defers the delivery of events to the `EDRAdaptor` if the system is under increased load. When the load has decreased, the broker delivers them with a high frequency to the `EDRAdaptor`.

The second effect can not be explained with this queuing mechanism. It occurs for example at sample position 50, with several samples having a positive deviation from the mean without their following samples compensating for this effect. The graphs show that the error occurs in a characteristic pattern, with samples having a time difference to its predecessor which is a multitude ($100ms * x$ with $x \geq 2$) of the time difference specified by the manufacturer. This effect is observed in all properties that are registered through the `CANAdaptor`, like the property `car.speed.current`, which also shows this characteristic pattern. Since the speed is sampled with 100 Hz the deviation is a multitude of 10 ms. In contrast, other subsystems like the FleetNet router (see graph 5.6) or navigation system are delivering samples without these effects.

The operation under load has already been tested previously, and the DC-EDR did not lose any samples, even under high data load. Therefore the following remaining errors along the pipeline are possible:

- An error occurred on the CAN bus, delaying / dropping the data[3]
- The device drivers delivering the CAN data to the middleware are dropping messages

[3]see [CAN05] about details about the CAN bus and protocol.

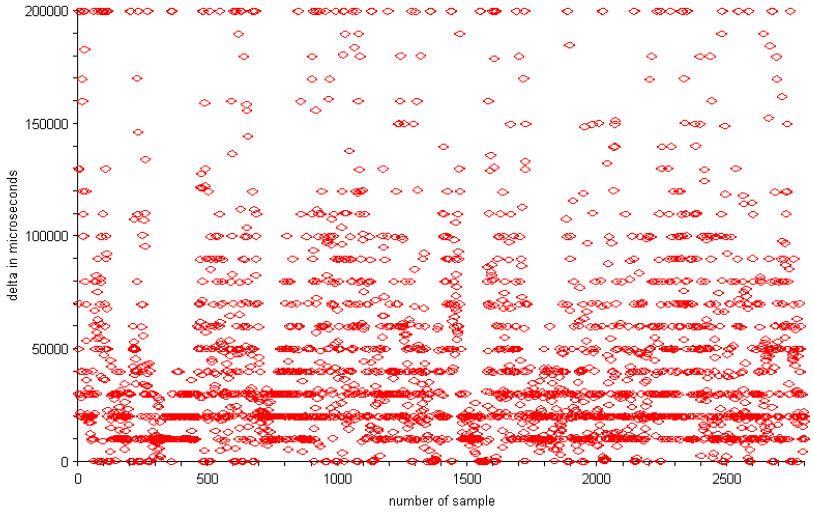

Figure 5.5: recording time stamp deltas for car.speed.current

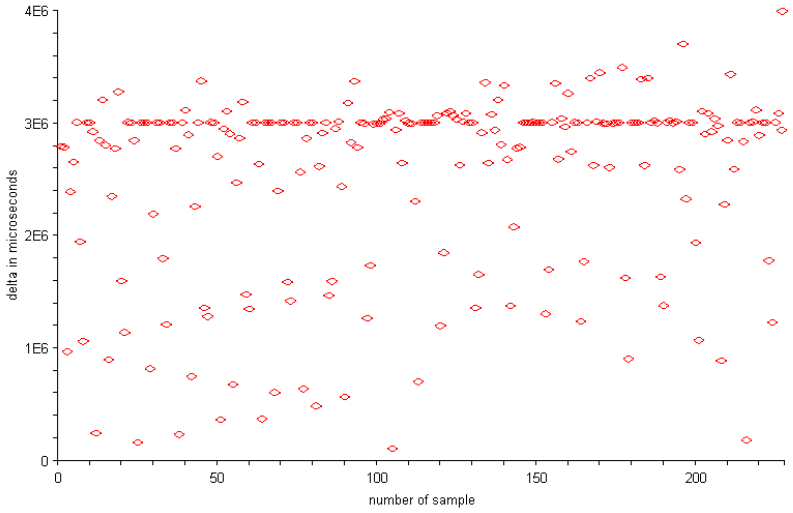

Figure 5.6: recording time stamp deltas for car.neighborhood.current

It must be noted, that in normal operations mode, two filtering methods are used to sort out data with similar consecutive values. The purpose of this mechanism is to reduce the number of POs on the event bus. During the test runs for this section these two filters were disabled, allowing consecutive POs with equal values to be sent to the event bus. With no filtering of data happening in the middleware and underlying drivers, further tests have to be conducted on the CAN data pipeline outlined in section 3.2.1 starting at the CAN bus interface. With means of a CAN simulator/tester producing data in controlled intervals these tests could be conducted, which was out of scope of this thesis.

To sum up the results of this section it can be said that the prototype could be operated successfully in a live environment with the restriction of the errors identified in this section. For a production environment these errors have to be identified using a CAN hardware simulator.

E200

Despite the test runs being made exclusively on the Smart test vehicles table 5.6 was added to show the availability of data elements on the future automotive development platform of Fraunhofer FOKUS, a Mercedes-Benz E200.

5.4 Synchronization of Logs

For now the focus of our tests was on the standalone operation of the DC-EDR. Since the main concept of the DC-EDR is its distributed cooperative operation mode, the following test will focus on the trigger event which enables the synchronization of multiple log files. The successful distributed recording was already shown during the basic operation test. To enable a cooperative mode of operation the log files which are now distributed across multiple vehicles have to be synchronized to allow the ordering of events in a linear timeline. This timeline allows the events on one vehicle (e.g. sharp breaking) to be put into perspective by looking at the data of the surrounding vehicles.

The method for log synchronization introduced in chapter 3.1 was tested in 2 stages. First the DC-EDR was installed on two workstations connected with a wired network interface. After successful completion of this test the implementation was moved onto the vehicles and tested during 2 test runs. To observe the behavior under an increased data load, the number of recorded POs was increased during the second test run by adding the recording of the lateral acceleration to the recorded data set.

In each test run a number of crashes were generated using the `DataGenAdaptor`. With no absolute time besides the system time stamps being available on the target systems, this time had to serve as a leveling rule for evaluating the accuracy of the DC-EDR. Since the system times of the internal clocks could be off by an undetermined amount of time, the following calculation is necessary to put the system times into perspective.

Every transmitted crash trigger and acknowledge packet carries two system time stamps. One from the transmitting instance and one from the receiving instance of the DC-EDR. The difference of these time stamps δ_{crash} is made up by two addends:

- The transmission time $\delta_{I,II}$

Data element	middleware property name	resolution	sampling rate CAN
Acceleration x-axis	-	-	-
Maximum Δv	-	-	-
Vehicle indicated speed	car.speed.current	1 km/h	10 Hz
Engine RPM	car.engine.rpm.current	1 RPM	10 Hz
Engine throttle	car.engine.torque.pedal	0.4%	50 Hz
Service brake	car.manualbrake.current	on/off	10 Hz
Ignition cycle at crash time	-	-	-
Ignition cycle at download time	-	-	-
Driver safety belt status	-	-	-
Air bag warning lamp (for every airbags)	-	-	-
Air bag deployment level (used for multi-state air bags)	-	-	-
Air bag time to 1-deploy (if the airbag only has 1 level this denotes the time for deployment)	-	-	-
Event number for multi-events	N/A	timestamp	once per event
Time between events	N/A	1 μs	once per event
Complete file recorded	N/A	yes/no	once per event
Acceleration y-axis	car.acceleration.lateral	0.08 $\frac{m}{s^2}$	50 Hz
GPS longitude and latitude	car.navigation.position.current.*	0.000001'	N/A
Trigger event date	car.edr.event	1 day	N/A
Trigger event time	car.edr.event	1 ms	N/A

Table 5.6: Availability of properties in the E200 middleware

- The time difference of the two system clocks δ_{clock}

Therefore we can calculate the time difference of the internal clocks of the two machines with

- $\delta_{clock} = \delta_{crash} - \delta_{I,II}$

The system clock time difference should be nearly constant during the test runs, as they all took under 1 hour to complete. If the DC-EDR works flawlessly, the standard deviation of δ_{clock} should be equal or below the standard deviation of the system clocks on the target systems, because the standard deviation of $\delta_{I,II}$ would be zero with the data measured over several runs.

5.4.1 Dry Run

Test Setup

The first tests were conducted in a controlled lab environment where two instances of the DC-EDR ran on workstations connected with ordinary 100 Mbit wired ethernet network interfaces. The machines system clocks had a standard deviation of 7.8 ms on machine A and 0.5 ms on machine B.

Machine A simulates the car that is involved in the crash and sends the crash trigger out to machine B, which simulated the uninvolved car only recording data upon reception of the trigger.

This measurement was repeated 19 times consecutively. Every ten seconds a crash was simulated generating the according entries in the DC-EDR logs to obtain mean value and standard deviation.

Evaluation of Data

As expected with this kind of interface the average transmission time was 4.1 ms including the time for registration of the crash inside the `AdaptorEDR`.

The first value $n = 1$ of $\delta_{I,II}$ in table B.1 is unusually large due to the fact that after receiving the first trigger the networking part of the DC-EDR is instantiated for the first time.

The results in table B.2 show that the difference of the two internal clocks is 126 ms with a standard deviation of 2,5 ms which means that in this scenario we can synchronize the two logs with an average accuracy of 2,5 ms or 3 samples.

The deviation was in the same order of magnitude of the system clock accuracy which shows that log synchronization can be achieved in a sense that a linear timeline of events across multiple DC-EDRs can be established.

5.4.2 Test Run

With the successful dry test the system was now ready to be deployed on the test platform.

Test Setup

The setup for the test run involved 2 vehicles which were equipped with the `InCarMiddleware` and DC-EDRs. They were parked within communication range of one another. Then vehicle A was driven out of communication range

and returned to its origin, reestablishing the radio link with vehicle B. During the whole time vehicle A was generating crash triggers and broadcast them in one minute intervals. Vehicle B received these events and sent back an acknowledge.

The following high frequency (> 5 Hz)

- `car.speed.current`
- `car.engine.rpm.current`
- `car.engine.torque.pedal`

and low frequency (< 5 Hz)

- `car.neighborhood.complete`
- `car.navigation.position.current.*`
- `car.manualbrake.current`

POs were recorded during this test run, adding up to an average of one POs being sent to the middleware every 20 milliseconds. The configuration of the platforms was outlined in chapter 3.2.

The data pool consists of 22 crash / acknowledge cycles, which were collected over a time of 29 minutes. Since the connection between the two vehicles was severed during the test run 7 trigger cycles were performed without acknowledge of vehicle B.

Evaluation of Data

The results for the time differences are summarized in table B.3

When the crash trigger is generated two events take place:

- The ring buffer is written to disk
- The remote trigger is broadcasted.

With the data dump having started before the remote trigger is serialized and sent over the network the time from crash to transmission of the trigger $\delta_{I,1}$ increases due to the processing overhead of the dump thread. This dump continues until the post-recording time was reached, which means that the system performs file operations until time t after the acknowledge cycle is completed ($t > \delta_{I,1} + \delta_{I,2}$). As a result the accuracy of both, $\delta I, 1$ and $\delta I, 2$ decreases. This can be seen in the higher mean deviation, which is coherent to the results of previous tests, showing the negative effect of file operations.

Since the dumping of the data is equally important as the signaling of the crash to the network the only feasible solution here would be to transmit the crash trigger and after a timeout allowing neighboring vehicles to send an acknowledge packet start the saving of the data into non-volatile memory. This approach would trade the first duty of the DC-EDR: recording local data against the accuracy aspect.

If we now look at the data from the parked vehicle B, we can see that $\delta_{II,1}$ does not experience such an dramatic increase. Neither in its mean value nor in

the mean deviation since the DC-EDR on this vehicle is only recording minimal data. This is due to the filtering of data which blocks consecutive similar values like the speed being constantly zero.

Because of the inaccuracy of the previous data, the calculated values δ_t and $\delta_{I,II}$ have to be handled with care, since the measurement errors are accumulated.

Nevertheless the transmission time δ_t clearly increased when we the system was moved from the wired to the wireless network environment, which subsequently increases the deviation of the two DC-EDR logs in respect to each other.

The result of the evaluation using the system times from table B.4 revealed the indeterministic behavior of the DC-EDR. With both system running under Linux OS the mean deviation of the system time is 0.5 ms. The mean deviation of δ_{clock}, which should be in the same order of magnitude was much higher with a value of 118 ms.

The main result of this test is that even with optimized data handling and multithreading the accuracy of the DC-EDR is not sufficient to establish a timeline across multiple DC-EDRs on the test platform. This is due to the processing overhead of file handling opposing the accuracy necessary for an EDR.

5.4.3 Test Run with Increased Load

Test Setup

The second test run had a similar layout with the following differences:

The DC-EDRs on vehicle A was configured to record a higher data load. The following additional POs were recorded during this test run:

- `car.crashsensor.current` (low frequency)
- `car.acceleration.lateral` (high frequency)

In addition, the filtering mechanism was disabled, allowing consecutive POs with the same value to be sent to the event bus. The number of POs on the event bus doubled with an average of 1 PO every 10 ms.

The crash trigger was generated every 10 seconds which generated a pool of 51 samples during a recording time of 11 minutes. With the higher frequency of POs on the event bus the time for dumping the ring-buffer increased, sustaining file operations until after the crash / acknowledge cycle was completed. This is the same behaviour as during the previous test run.

With the values from section 5.2 taken into account the accuracy of the time stamps should decrease during this test run.

Evaluation of Data

The results for the time differences are summarized in table B.5.

As expected nearly all mean values as well as standard deviation increased during this test run. Since the parked vehicle B also dumped a minimal amount of data the increased trigger frequency had a negative impact on the mean processing time for the acknowledge packet $\delta_{II,1}$

Interestingly the mean and standard deviation of $\delta_{I,1}$ decreased, which can only be explained by the limited number of measuring data during the first test run.

Subsequently the evaluation of the data (B.6) using the system time shows a standard deviation of 342 ms for δ_{clock} which is a result of the decreased accuracy of $\delta_{I,II}$.

5.4.4 Possible Error Sources

The mean deviation measured for the time difference of two DC-EDR logs calculated in the previous section showed that this method could not be used in practive even though the dry runs had promising results.

Since the standard deviation of δ_{clock} breaks down into the deviations of

1. the measuring method itself

2. the time difference between registration of an event
   inside the `PropertyBroker` and the recording by the DC-EDR

the error for these parts has to be examined.

The magnitude of the first error could be evaluated by examining the logs recorded during the second test run.

The measuring method is based on the availability of two timestamps: the high resolution ticks and the system time. With both being generated subsequently during execution of the code, the deviation between them should be minimal. To be exact, it should be equal or below the mean deviation of the system time for a large enough pool of samples. If it is, we measured the system inherent error of the system time.

With `car.engine.torque.pedal` being the PO with the highest sampled rate the time difference between these two timestamps was calculated for a pool of over 10000 samples.

The result was that the standard deviation between the two timestamps is 303.37μs, which is below the system time deviation, with a maximum deviation from the average value of $max_m = 1761.5\mu$s.

The deviation inside the middleware is the second factor. Every event has to pass through the `PropertyBroker` (as described in section 3.3.2).

The second test had to be performed on data generated in a dry run, since the DC-EDR was not equipped to record such detailed data during the test runs.

For this test two `Adaptors` were equipped with high frequency timers. The first timer produced a timestamp after an event was sent off to the `PropertyBroker`. The second timer was placed after the event registered in the `EDRAdaptor`. During the test no file operations happened in the DC-EDR.

The results showed that one pass through the middleware took an average of $1066,09\mu$s and the standard deviation was $243,51\mu$s (with a pool of 127 samples). The maximum deviation from the average value in the test was $max_b = 1723,90\mu$s.

Since every event has to pass through the middleware before a time stamp is added, we define $max_{bm} = max_b + max_m$.

Putting all this together we get the following worst case deviation (all deviations adding up) for a complete crash trigger transmission and acknowledge cycle:

$$
\begin{aligned}
MAX = & \underbrace{\underbrace{\underbrace{max_{bm}}_{\text{crash registration}} + \underbrace{max_{bm}}_{\text{trigger registration}}}_{\text{vehicle A}} \\
& + \underbrace{\underbrace{\underbrace{max_{bm}}_{\text{crash registration}} + \underbrace{max_{bm}}_{\text{acknowledge registration}}}_{\text{vehicle B}} \\
& + \underbrace{\underbrace{max_{bm}}_{\text{acknowledge registration}}}_{\text{vehicle A}} \\
& = 5 * max_{bm}
\end{aligned}
$$

The maximum deviations of both tests are well below the standard deviation of δ_{clock} during the test runs, with $MAX = 17$ ms as the worst case being one magnitude below the standard deviation during the test runs.

With these results it can be said that the error cannot be found at a special point, like the measuring method or the delay introduced by one pass through the middleware, but in the general erratic system behavior during lengthy disk operations as shown is section 5.2. This problem has to be specially addressed when optimizing the system.

5.5 Image Capturing

5.5.1 Test Setup

To test the image capturing capabilities of the DC-EDR and illustrate a usage scenario the following test setup was used:

Both vehicles were parked within communication range of each other. One vehicle was faced towards a wall, whereas the other was parked farther away from the crash scene to provide a global view (see figure 5.7).

One vehicle was generating crash triggers in regular intervals, which started the local capturing of an image, showing the situation at crash time. The crash trigger being sent to the second vehicle triggered the secondary image capturing process thus providing a different angle of the scene.

Since the trigger intervals were set at 10 seconds a series of images (two shown in 5.8) were captured.

5.5.2 Results

With the USB webcams only delivering a limited quality picture the results cannot be used for reconstruction of the crash scene. The current implementation

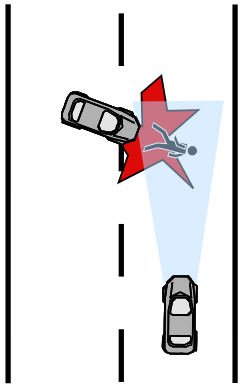

Figure 5.7: Image capturing test setup schema

Figure 5.8: Image capturing test results

delivers a 320x240 pixel resolution for all 3 installed webcams simultaneously with a possible resolution of 640x480 pixels for a single camera.

Nevertheless this scenario illustrates the basic idea of distributed capturing. Since both cameras have a high focal length, the camera in vehicle A does not capture the injured person in the top right picture, while the camera in vehicle B does.

5.5.3 Optimizations

When the system was operated under load the accuracy of the timers decreased, especially when data was written to disk. During the crash of a vehicle the system is under the most stress because it has to

- Store the data from volatile into non-volatile memory
- Send the crash trigger to other vehicles
- Send the data collected so far to other vehicles

In addition, this is the time where it has to operate with the highest accuracy to provide exact timing information for log synchronization and post-crash analysis of the data. The first second after the crash is the most interesting in terms of crash analysis.

To minimize the load on the system an optimization of the data format is of top-most priority for a DC-EDR being able to run in a production environment.

With file input / output operations using the XML codec being very time consuming the data format has to be optimized for size. This has the positive side effect that it would take less time to transmit the data captured on the accident vehicle to a remote location outside the hazard zone.

With an average of 500 Bytes for one data object in XML representation we would get a file of 562 kilobytes for one crash log containing the required data elements from table 5.3. The transmission with the current 802.11 network interface would take about one second to complete. Based on experience values during the operation of the FleetNet prototype the maximum net transmission rate with a 11 Mbit link using UDP is about 680 kilobytes per second.

With the IEEE Standard defining data formats for the required data elements the volume of data can be reduced. The recommended data formats are summarized in table 5.7.

With these optimized data formats we get a smaller file for storage and transmission which would dramatically reduce the load on the system. Since all elements except the crash acceleration will be recorded 8 seconds before to 5 seconds after the crash, the size of the file will be 12.7 kilobytes including 8 byte integer time stamps (or 3.7 kilobytes without timestamps), which would transmit in about 19 milliseconds.

By using this lightweight format for storage and transmission a complete DC-EDR log could be transmitted in approximately 10 packets[4]. The disadvantage would be the readability of this format. With the elaborate XML providing semantic information along with the raw data interoperability between different combinations of EDR and analysis equipment would be simplified. In addition,

[4] using standard UDP with an MTU of 1492 bytes

Data element	minimum sampling rate	data format
Acceleration x-axis (pre crash)	10 Hz	floating point[a]
Acceleration x-axis (crash -0.1s to +0.5s)	1000 Hz	floating point
Vehicle indicated speed	10 Hz (assumed)	floating point (assumed)
Engine RPM	5 Hz	integer (2 bytes)
Engine throttle	4 Hz	1 byte
Service brake	2 Hz	1 byte (assumed)
Ignition cycle at crash time	once per cycle	integer (4 bytes)
Ignition cycle at download time	once per cycle	integer (4 bytes)
Driver safety belt status	10 Hz	1 byte (assumed)
Air bag warning lamp (once for every airbag)	N/A	integer (for last 10 counts) (4 bytes assumed)
Air bag deployment level (used for multi-state air bags)	by manufacturer	1 byte
Air bag time to 1-deploy	once per event	integer (4 bytes assumed)
Event number for multi-events	once per event	1 byte (assumed)
Time between events	N/A	integer (4 bytes)
Complete file recorded	once per event	1 byte (assumed)

Table 5.7: Recommended data formats for the required IEEE Standard data elements

[a]assuming 32-bit IEEE standard format

with the standardized object serialization used for the prototype, format changes can be implemented globally for both encoder and decoder of the data in one central place (see [Mil]).

The data format recommended by the IEEE standard provides a dramatically reduced data rate for the log files thereby allowing an increased accuracy during operation.

Chapter 6

Conclusions and Outlook

The evaluation of the prototype tried to verify the improvements stated in chapter 2 to their road capability. The analysis provided the following results:

1. The basic operation of a distributed cooperative EDR in a vehicular network is possible with multiple vehicles recording data upon reception of a trigger event through the network.

2. A non-realtime software architecture cannot provide the necessary timing requirements to reach millisecond accuracy which is crucial in this application domain. Therefore a realtime OS or a hardware solution has to be used.

3. The DC-EDR can support the sustained data rate necessary for recording the data elements required by the IEEE Standard with the restriction of 2.

4. To establish a timeline of events across several DC-EDR logs the introduced method proved to be too susceptible to errors introduced by demanding disk operations, as they proved to have a drastic impact on the accuracy of the prototype.

5. The distributed capturing of images enabled by the DC-EDR successfully showed the future capabilities of this approach.

With the DC-EDR being integrated in the middleware, the use as a network probe will support multiple applications, since it can directly access all data on the event bus. As showed during the basic operation test, this usage can for example give insights into the behavior of the ad-hoc router.

Future work could improve the accuracy of the DC-EDR by porting it to a realtime system which would subsequently require the whole middleware to be running on such a system. The advantages of the DC-EDR being integrated into the middleware has to be weighted against the separation from the middleware, depending on its future use.

Additionally the aspect of data distribution and recovery with geographically agile nodes has to be researched. The following questions outline some of the problems of distributed storage.

- How can vehicles be identified and requested to transmit their captured data to a central storage?
- Should data be transmitted upon request or at the next possible opportunity?
- How can data be associated with a specific accident?
- How many accidents or near-accident should be stored?
- Which amount of data should be stored for which time period? (video vs. low-profile data)

With the analysis of data rates, some starting points are already given in this thesis, but they have to be researched in depth by subsequent work.

Finally, the use of a VANET traffic simulator could provide insights to the behavior of the DC-EDR in an environment with more than two cars which could answer the following questions.

- Did the crash triggers get routed to all relevant vehicles in the network?
- What was the network's response time after the crash with multiple vehicles broadcasting messages?
- Did parts of the network collapse due to overload of messages?

Since these problems do not only concern the operation of the DC-EDR but many of the applications in a VANET domain, they are currently hot topics in the research community.

Appendix A

DC-EDR Graphs

The following graphs were generated using the data collected during the test runs. They show the coherence between several recorded data sets on one vehicle. The triggers used to record the data sets on which these graphs are based were generated manually by the `DataGenAdaptor`. Since a real crash test using two vehicles was out of the question during this work, no real correlation between the logs of two vehicles can be shown here.

Graph A.1 shows the coherence between the engine torque requested by the driver in percentage of throttle depression and the actual vehicle speed.

Graph A.2 shows the coherence between requested torque and actual RPM of the engine.

Graph A.3 shows the recording of a test drive conducted in underground parking. A positive lateral acceleration depicts a right turn whereas a negative depicts a turn in the left direction.

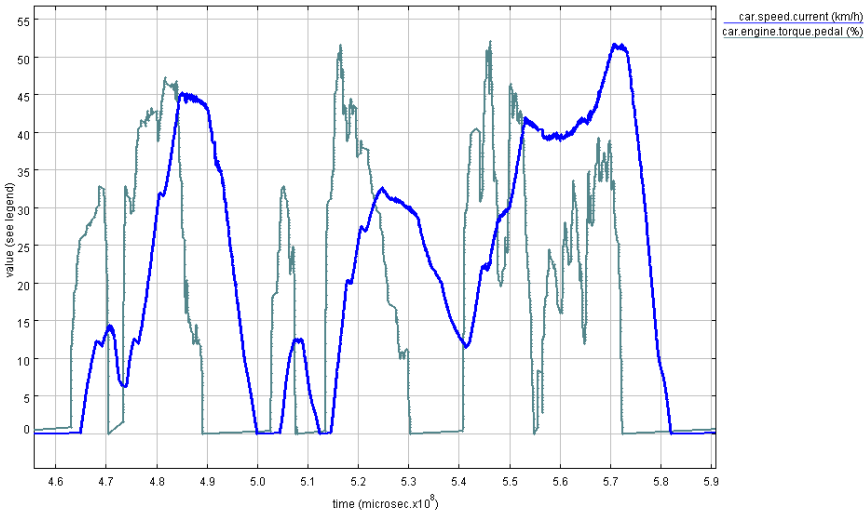

Figure A.1: Torque / speed graph

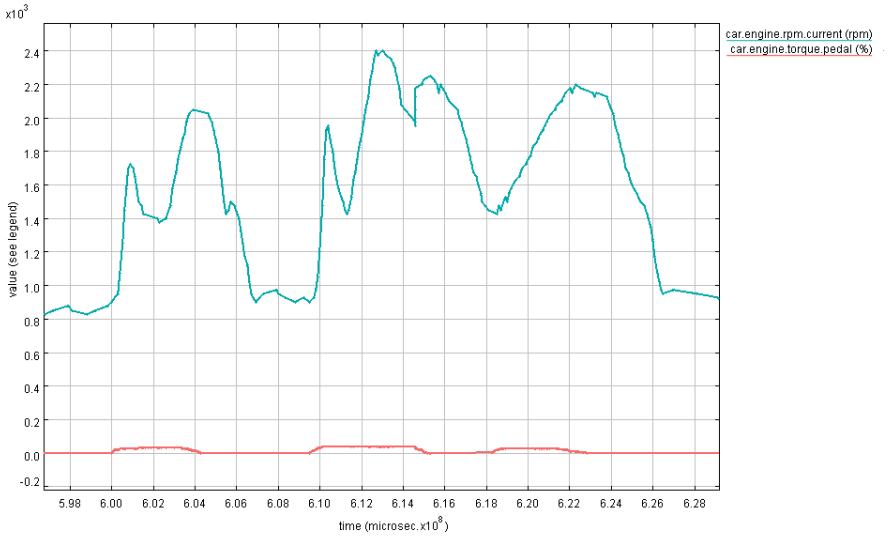

Figure A.2: Torque / RPM graph

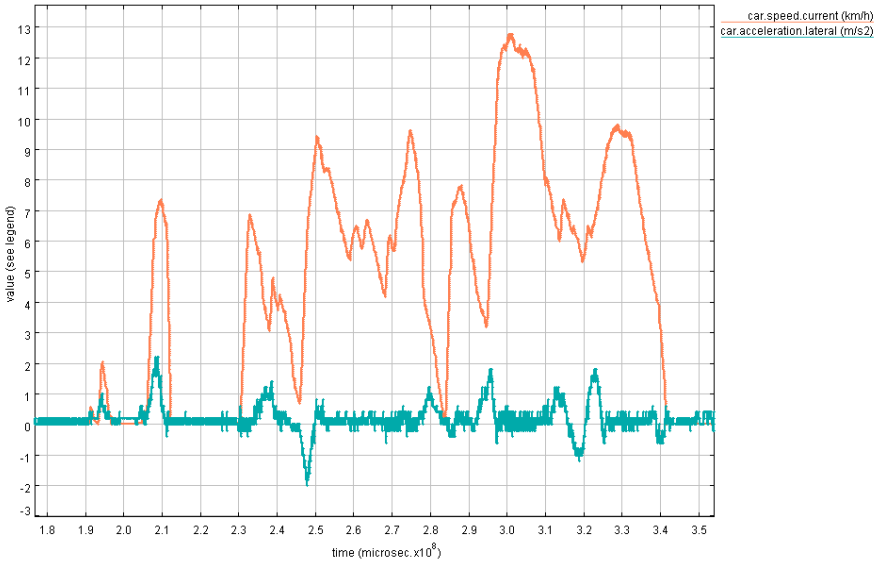

Figure A.3: Speed / lateral acceleration graph

Appendix B

Time Difference Data Sets

All δs in the following tables are measured in μs.

Data Sets Recorded during Dry Runs

n	$\delta_{I,1}$	$\delta_{I,2}$	$\delta_{II,1}$	δ_t	$\delta_{I,II}$
1	38098	91642	61081	15280.5	53378.5
2	7050	12281	4178	4051.5	11101.5
3	5998	10647	5746	2450.5	8448.5
4	8393	10053	4269	2892	11285
5	6479	10747	4613	3067	9546
6	5584	30917	21118	4899.5	10483.5
7	6095	7093	53	3520	9615
8	6423	18297	12645	2826	9249
9	6080	7944	1851	3046.5	9126
10	6662	5543	53	2745	9407
11	6143	10684	126	5279	11422
12	6572	8959	2074	3442.5	10014.5
13	5574	9817	65	4876	10450
14	6305	7270	55	3607.5	9912.5
15	6131	5376	838	2269	8400
16	6379	44029	37673	3178	9557
17	5610	7319	2540	2389.5	7999.5
18	6230	9408	50	4679	10909
19	5760	7439	64	3687.5	9447.5
mean	7977.16	16603.42	8373.26	4115.08	12092.24
std. dev.	7126.37	19979.59	15438.82	2772.94	9776.94

Table B.1: Calculation of DC-EDR trigger time difference during dry run in section 5.4

With mean being calculated as

- $x_{mean} = \frac{1}{n} \sum_{i=1}^{n} x_i$

and the standard deviation as

- $x_{std.dev.} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - x_{mean})^2}$

n	δ_{crash}	$\delta_{I,II}$	δ_{clock}
1	171000	53378.5	117621.5
2	136000	11101.5	124898.5
3	133000	8448.5	124551.5
4	136000	11285	124715
5	134000	9546	124454
6	133000	10483.5	122516.5
7	135000	9615	125385
8	134000	9249	124751
9	136000	9126.5	126873.5
10	136000	9407	126593
11	136000	11422	124578
12	137000	10014.5	126985.5
13	137000	10450	126550
14	137000	9912.5	127087.5
15	136000	8400	127600
16	137000	9557	127443
17	136000	7999.5	128000.5
18	139000	10909	128091
19	138000	9447.5	128552.5
mean	137736.84	12092.24	125644.61
std. dev.	7985.79	9776.94	2453.25

Table B.2: Calculation of system time difference during dry run in section 5.4

Data sets recorded during test runs

n	$\delta_{I,1}$	$\delta_{I,2}$	$\delta_{II,1}$	δ_t	$\delta_{I,II}$
1	146422	95727	301	47713	194135
2	36315	304338	284	152027	188342
3	19987	390896	275	195310.5	215297.5
4	25088	59615	280	29667.5	54755.5
5	46368	416291	27534	194378.5	240746.5
6	27873	19649	265	9692	37565
7	55497	596504	282	298111	353608
8	44643	105410	286	52562	97205
9	31659	10050	259	4895.5	36554.5
10	262502	29100	302	14399	276901
11	9919	86757	293	43232	53151
12	43641	98121	60512	18804.5	62445.5
13	7104	104380	62689	20845.5	27949.5
14	57066	24746	271	12237.5	69303.5
15	29860	694007	275	346866	376726
16	38730	78330	269	39030.5	77760.5
17	49285	43706	245	21730.5	71015.5
18	44671	44624	274	22175	66846
19	29983	73543	271	36636	66619
20	37352	28920	258	14331	51683
21	39977	40767	4448	18159.5	58136.5
22	11565	217939	223	108858	120423
mean	49795.77	161973.64	7277.09	77348.27	127144.05
std. dev.	53638.93	190486.03	18098.01	95791.75	103026.23

Table B.3: Calculation of EDR trigger time difference during first test run in section 5.4

n	δ_{crash}	$\delta_{I,II}$	δ_{clock}
1	1376000	194135	1181865
2	1536000	188342	1347658
3	1604000	215297.5	1388702.5
4	1294000	54755.5	1239244.5
5	1647000	240746.5	1406253.5
6	1255000	37565	1217435
7	1870000	353608	1516392
8	1374000	97205	1276795
9	1262000	36554.5	1225445.5
10	1270000	276901	993099
11	1329000	53151	1275849
12	1270000	62445.5	1207554.5
13	1273000	27949.5	1245050.5
14	1275000	69303.5	1205696.5
15	1967000	376726	1590274
16	1336000	77760.5	1258239.5
17	1309000	71015.5	1237984.5
18	1286000	66846	1219154
19	1308000	66619	1241381
20	1301000	51683	1249317
21	1299000	58136.5	1240863.5
22	1409000	120423	1288577
mean	1402272.73	127144.05	1275128.68
std. dev.	195456.15	103026.23	118024.38

Table B.4: Calculation of system time difference during first test run in section 5.4

n	$\delta_{I,1}$	$\delta_{I,2}$	$\delta_{II,1}$	δ_t	$\delta_{I,II}$
1	205142	961915	260064	35925.5	556067.5
2	17385	133132	9964	61584	78969
3	32209	277849	12636	132606.5	164815.5
4	52868	361834	36581	162626.5	215494.5
5	262841	25768	10483	7642.5	270483.5
6	44446	243175	16420	113377.5	157823.5
7	19989	334680	335	167172.5	187161.5
8	9541	425610	146	212732	222273
9	41428	266355	319	133018	174446
10	9971	582703	8864	286919.5	296890.5
11	60384	10142	347	4897.5	65281.5
12	32081	395439	16211	189614	221695
13	42803	449390	343	224523.5	267326.5
14	5663	75532	19117	28207.5	33870.5
15	10006	158925	16471	71227	81233
16	40021	127184	145	63519.5	103540.5
17	40027	183139	21500	80819.5	120846.5
18	35270	362365	151	181107	216377
19	4293	142026	17431	62297.5	66590.5
20	4188	57222	11403	22909.5	27097.5
21	3446	405304	13068	196118	199564
22	4204	54827	10998	21914.5	26118.5
23	4352	85168	318	42425	46777
24	4064	649312	19901	314705.5	318769.5
25	3310	74282	145	37068.5	40378.5
26	4070	52520	12852	19834	23904
27	4237	64916	11159	26878.5	31115.5
28	4013	112776	45449	33663.5	37676.5
29	4052	766475	144	383165.5	387217.5
30	4159	437217	314	218451.5	222610.5
31	3615	454866	145	227360.5	230975.5
32	3920	201169	16715	92227	96147
33	20110	471159	145	235507	255617
34	3208	527040	312	263364	266572
35	3462	445605	136	222734.5	226196.5
36	3929	30258	299	14979.5	18908.5
37	4070	55846	33727	11059.5	15129.5
38	3475	389423	307	194558	198033
39	3311	440280	113	220083.5	223394.5
40	3938	49690	308	24691	28629
41	3430	215437	178	107629.5	111059.5
42	3355	215691	23689	96001	99356
43	3525	259070	170	129.5	132975
44	3883	29649	318	14665.5	18548.5

continued on next page

n	$\delta_{I,1}$	$\delta_{I,2}$	$\delta_{II,1}$	δ_t	$\delta_{I,II}$
45	3357	188806	141	94332.5	97689.5
46	3836	29666	312	14677	18513
47	4290	31988	10775	10606.5	14896.5
48	3966	32574	9232	11671	15637
49	4288	38352	143	19104.5	23392.5
50	3241	180035	64066	57984.5	61225.5
51	3273	189340	120	94610	97883
mean	21645.98	25061.29	14404.51	117828.39	139474.37
std. dev.	45943.01	211730.85	37108.91	99873.29	113887.99

Table B.5: Calculation of EDR trigger time difference during test run with increased load in section 5.4

n	δ_{crash}	$\delta_{I,II}$	δ_{clock}
1	972000	556067.5	415932.5
2	1768000	78969	1689031
3	1524000	164815.5	1359184.5
4	1410000	215494.5	1194505.5
5	1666000	270483.5	1395516.5
6	1563000	157823.5	1405176.5
7	1481000	187161.5	1293838.5
8	1391000	222273	1168727
9	1555000	174446	1380554
10	1215000	296890.5	918109.5
11	759000	65281.5	693718.5
12	1399000	221695	1177305
13	1308000	267326.5	1040673.5
14	717000	33870.5	683129.5
15	1680000	81233	1598767
16	1647000	103540.5	1543459.5
17	1580000	120846.5	1459153.5
18	1450000	216377	1233623
19	1760000	66590.5	1693409.5
20	756000	27097.5	728902.5
21	1409000	199564	1209436
22	752000	26118.5	725881.5
23	758000	46777	711223
24	1314000	318769.5	995230.5
25	722000	40378.5	681621.5
26	750000	23904	726096
27	749000	31115.5	717884.5
28	755000	37676.5	717323.5
29	1450000	387217.5	1062782.5
30	1362000	222610.5	1139389.5
31	1340000	230975.5	1109024.5
32	1617000	96147	1520853
33	1291000	255617	1035383
34	1269000	266572	1002428
35	1351000	226196.5	1124803.5
36	747000	18908.5	728091.5
37	725000	15129.5	709870.5
38	1387000	198033	1188967
39	1336000	223394.5	1112605.5
40	745000	28629	716371
41	1562000	111059.5	1450940.5
42	1576000	99356	1476644
43	1516000	132975	1383025
44	744000	18548.5	725451.5

continued on next page

n	δ_{crash}	$\delta_{I,II}$	δ_{clock}
45	1599000	97689.5	1501310.5
46	743000	18513	724487
47	740000	14896.5	725103.5
48	739000	15637	723363
49	741000	23392.5	717607.5
50	645000	61225.5	583774.5
51	581000	97883	483117
mean	1188549.02	139474.37	1049074.65
std. dev.	378420.31	113887.99	341521.04

Table B.6: Calculation of system time difference during test run with increased load in section 5.4

Acronyms

AS Active Safety

BMBF Bundesministerium für Bildung und Forschung

CAN Controller Area Network

DC-EDR Distributed Cooperative Event Data Recorder

DO DatedObject

EDR Event Data Recorder

EEPROM Electrically-Erasable Programmable Read-Only Memory

GMC General Motors Corporation

GPS Global Positioning System

GSM Global System for Mobile Communications

JNI Java Native Interface

MTU Maximum Transmission Unit

MVEDRs Motor Vehicle Event Data Recorders

Mbit Megabit

NHTSA National Highway Transport Safety Agency

NTP Network Time Protocol

OEM Original Equipment Manufacturer

OS Operating System

PIO PropertyInfoObject

PO PropertyObject

PROM Programmable Read-Only Memory

PS Passive Safety

RCM Restraint Control Module

RPM Revolutions Per Minute

SDM Sensing and Diagnostic Module

TSC Timestampcounter

UDS Unfalldatenspeicher

VANET Vehicular Ad-hoc Network

References

General references

[CEF02] Cseh, Christian ; Eberhardt, Reinhold ; Franz, Walter: Mobile Ad-Hoc Funknetze für die Fahrzeug-Fahrzeug-Kommunikation. In: *Deutscher Workshop über Mobile Ad Hoc Netze, WMAN 2002, Ulm*

[CHMS99] Chidester, Augustus ; Hinch, John ; Mercer, Thomas C. ; Schultz, Keith S.: Recording Automotive Crash Event Data. In: *Proceedings of the International Symposium on Transportation Recorders*

[CHR98] Chidester, Augustus B. ; Hinch, John ; Roston, Thomas A.: Real World Experience with Event Data Recorders / National Highway Traffic Safety Administration. Version: 1998. `http://www-nrd.nhtsa.dot.gov/edr-site/uploads/Real_world_experience_with_event_data_recorders.pdf` (247). – Paper. – Online–Ressource

[Eck02] Eckel, Bruce: *Thinking in Java*. Prentice Hall Professional Technical Reference, 2002. – ISBN 0–1310–0287–2

[EGE02] Elson, Jeremy ; Girod, Lewis ; Estrin, Deborah: Fine-grained network time synchronization using reference broadcasts. In: *SIGOPS Oper. Syst. Rev.* 36 (2002), Nr. SI, S. 147–163. `http://dx.doi.org/http://doi.acm.org/10.1145/844128.844143`. – DOI http://doi.acm.org/10.1145/844128.844143. – ISSN 0163–5980

[ER03] Elson, Jeremy ; Römer, Kay: Wireless sensor networks: a new regime for time synchronization. In: *SIGCOMM Comput. Commun. Rev.* 33 (2003), Nr. 1, S. 149–154. `http://dx.doi.org/http://doi.acm.org/10.1145/774763.774787`. – DOI http://doi.acm.org/10.1145/774763.774787. – ISSN 0146–4833

[Gro99] Grossi, Dennis R.: Aviation Recorder Overview. In: *Proceedings of the International Symposium on Transportation Recorders*

[IEE04] IEEE Task P1616: *IEEE Standard for Motor Vehicle Event Data Recorders (MVEDRs)*. : IEEE, New York, 2004

[Sti05] Stiller, Andreas: Zeitfenster. In: *c't* (2005)

Patent references

[Bag99] Bague, Adolfo V.: *Traffic Accident Data Recorder and Traffic Accident Reproduction System and Method*. 1999

[CGTW99] Chainer, Timothy J. ; Greengard, Claude A. ; Tresser, Charles P. ; Wu, Chai W.: *Event-Recorder for transmitting and storing electronic signature data*. 1999

[JHB98] Jambhekar, Nilkanth S. ; Hara, Jacques ; Barr, John R.: *Verfahren und Gerät zur Datenaufzeichnung und -sicherung von Fahrzeugsteuerereignissen*. 1998

[MPB98] Mackey, John J. ; Pandolfi, Richard ; Brogan, Christopher J.: *Mobile Vehicle Accident Data System*. 1998

[Ray99] Rayner, Gary A.: *Vahicle Data Recorder*. 1999

[Sch01] Schatebeck, Harald: *Verfahren zur Meldung eines Notrufs*. 2001

[Sza88] Szabo, Viktor: *Accident Data Recorder*. 1988

Internet references

[Alt] *Altius Solutions LLC.* http://www.altiusllc.com

[CAN05] *Controller Area Network an Overview.* http://www.can-cia.org/can/. Version: 2001-2005

[CDR] *Crash Data Retrieval System from Vetronix.* http://www.vetronix.com/diagnostics/cdr/index.html

[Dri] *DriveCam Video Systems.* http://www.drivecam.de

[Fle] *FleetNet Homepage.* http://www.et2.tu-harburg.de/fleetnet/index.html

[Hai01] Haight, W. R.: *Automobile Event Data Recorder (EDR) Technology - Evolution, Data, and Reliability.* www.collisionsafety.net/documents/Event%20Data%20Recorders.pdf. Version: 2001

[Mil] Milne, Philip: *Using XMLEncoder.* http://java.sun.com/products/jfc/tsc/articles/persistence4/

[OnS] *OnStar from General Motors.* http://www.onstar.com/us_english/jsp/explore/onstar_basics/technology.jsp#aacn

[PB05] Pfisterer, Matthias ; Bomers, Florian: *Java Sound Resources: FAQ: Performance Issues.* http://www.jsresources.org/faq_performance.html. Version: 2005

[Ptp] *Ptplot.* http://ptolemy.eecs.berkeley.edu/java/ptplot/

[Rou] Roubtsov, Vladimir: *My kingdom for a good timer! - Reach submillisecond timing precision in Java.* http://www.javaworld.com/javaworld/javaqa/2003-01/01-qa-0110-timing.html