

**Project no.:** IST-FP6-STREP- 26979  
**Project full title:** Highly dependable ip-based networks and services  
**Project Acronym:** HIDENETS  
**Deliverable no.:** D2.3  
**Title of the deliverable:** Service level resilience solutions for the ad-hoc domain

<b>Contractual Date of Delivery to the CEC:</b>	30 <sup>th</sup> June 2008
<b>Actual Date of Delivery to the CEC:</b>	27 <sup>th</sup> June 2008
<b>Organisation name of lead contractor for this deliverable</b>	FCUL
<b>Author(s)/Participant(s):</b>	António Casimiro (editor), Henrique Moniz, Mónica Dixit, Luis Marques, Marc-Olivier Killijian, Erling V. Matthiesen, Alessandro Daidone, Matthieu Roy
<b>Work package contributing to the deliverable:</b>	WP2
<b>Nature:</b>	R
<b>Version:</b>	3.0
<b>Total number of pages:</b>	61
<b>Start date of project:</b>	1 <sup>st</sup> Jan. 2006 <b>Duration:</b> 36 month

**Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)**

Dissemination Level		
<b>PU</b>	Public	<b>X</b>
<b>PP</b>	Restricted to other programme participants (including the Commission Services)	
<b>RE</b>	Restricted to a group specified by the consortium (including the Commission Services)	
<b>CO</b>	Confidential, only for members of the consortium (including the Commission Services)	

**Abstract:**

The main objectives of WP2 are to define a resilient architecture and to develop a range of middleware solutions (i.e. algorithms, protocols, services) for resilience to be applied in the design of highly available, reliable and trustworthy networking solutions. This deliverable presents the final descriptions of the middleware services that have been designed and developed in HIDENETS, whose purpose is to facilitate the construction of more resilient or dependable car2car applications operating in ad-hoc environments.

The deliverable extends the results presented in deliverable D2.1.2 (Resilient Architecture), focusing on the implementation aspects of the complex middleware services previously introduced. This includes, specifically, the Diagnostic and Reconfiguration Manager, the QoS Coverage Manager, the Replication Manager, the Proximity Map, the Cooperative Data Backup and the Intrusion-Tolerant Agreement.

The work concerning the development and implementation of the services was done in close relation to the prototype implementation performed in WP6. In fact, while the description of the HIDENETS proof-of-concept prototypes has been done in WP6, the design, development and implementation of the services was done in WP2 and WP3. Therefore, for practical reasons, some implementation decisions have been influenced by the fact that the services are intended to be used in particular applications, namely the Platooning and the distributed black box applications.

Finally, this deliverable also provides the most relevant results that we achieved with the implementation of these complex middleware services, as well as some analysis and discussion of these results. However, given that the full evaluation of the services is one of the goals of WP4, a more complete and thorough discussion of these results is addressed in WP4 and presented in the corresponding deliverables.

**Version information**

Version	Date	Comments
0.1	20.04.2008	First draft with initial contributions
1.0	13.05.2008	Added changes according to decisions during Munich meeting Re-structured initial part of the deliverable Added text to initial section, and conclusions Added updates provided by UNIFI, LAAS and AAU Cleaned up text and arranged all references
2.0	06.06.2008	Added new contributions and changes, according to input from the internal review.
3.0	23.06.2008	Addressed comments from the external review team.

# Table of Contents

<b>1</b>	<b>EXECUTIVE SUMMARY .....</b>	<b>5</b>
<b>2</b>	<b>MIDDLEWARE RESILIENCE SOLUTIONS.....</b>	<b>6</b>
2.1	SERVICES FOR THE AD-HOC DOMAIN .....	6
2.2	ARCHITECTURAL APPROACH.....	7
2.3	HIDENETS NODE SOFTWARE ARCHITECTURE .....	9
2.4	COMPLEX RESILIENCE SERVICES.....	10
2.5	FROM THE DEFINITION TO THE IMPLEMENTATION OF COMPLEX MIDDLEWARE SERVICES .....	11
<b>3</b>	<b>DIAGNOSTIC AND RECONFIGURATION MANAGER.....</b>	<b>13</b>
3.1	SERVICE DESCRIPTION .....	13
3.2	DESIGN, SPECIFICATION, INTERFACE.....	13
3.3	IMPLEMENTATION .....	17
3.4	RESULTS, ANALYSIS AND DISCUSSION .....	20
3.5	SUMMARY .....	20
<b>4</b>	<b>QOS COVERAGE MANAGER.....</b>	<b>22</b>
4.1	SERVICE DESCRIPTION .....	22
4.2	DESIGN, SPECIFICATION, INTERFACE.....	23
4.3	IMPLEMENTATION .....	26
4.4	RESULTS, ANALYSIS AND DISCUSSION .....	29
4.5	SUMMARY .....	30
<b>5</b>	<b>REPLICATION MANAGER.....</b>	<b>32</b>
5.1	SERVICE DESCRIPTION .....	32
5.2	DESIGN, SPECIFICATION, INTERFACE.....	33
5.3	IMPLEMENTATION .....	34
5.4	RESULTS, ANALYSIS AND DISCUSSION .....	35
5.5	SUMMARY .....	39
<b>6</b>	<b>PROXIMITY MAP .....</b>	<b>40</b>
6.1	SERVICE DESCRIPTION .....	40
6.2	DESIGN, SPECIFICATION, INTERFACE.....	40
6.3	IMPLEMENTATION .....	41
6.4	SUMMARY .....	44
<b>7</b>	<b>COOPERATIVE DATA BACKUP .....</b>	<b>45</b>
7.1	SERVICE DESCRIPTION .....	45
7.2	DESIGN, SPECIFICATION, INTERFACE.....	45
7.3	IMPLEMENTATION .....	47
7.4	SUMMARY .....	50
<b>8</b>	<b>INTRUSION-TOLERANT AGREEMENT.....</b>	<b>51</b>
8.1	SERVICE DESCRIPTION .....	51
8.2	DESIGN, SPECIFICATION, INTERFACE.....	51
8.3	IMPLEMENTATION .....	53
8.4	SUMMARY .....	57
<b>9</b>	<b>CONCLUDING REMARKS .....</b>	<b>58</b>
	<b>REFERENCES .....</b>	<b>59</b>

# **1 Executive Summary**

## **Objectives of the deliverable**

As stated in the HIDENETS Technical Annex, the main objectives of WP2 are “*to define a resilient architecture and to develop a range of middleware solutions (i.e. algorithms, protocols, services) for resilience to be applied in the design of highly available, reliable, and trustworthy networking solutions*”.

The main objective of this deliverable is to describe the HIDENETS middleware solutions that may be used to improve the resilience of applications operating in ad-hoc environments. The deliverable is specifically concerned with the description of complex middleware services, on the aspects of their development and implementation. Nevertheless, in order to ensure self-containment and completeness, the deliverable also provides general information on the services and their objectives, building from descriptions from previous deliverables.

## **Contents of the deliverable and relation to other work packages**

The work in WP2 was structured in three main tasks. The first one (Task 2.1) was mostly related to architectural issues, and the remaining two (Tasks 2.2 and 2.3) were concerned with service aspects. The first two deliverables in this work package (D2.1.1 and D2.1.2) presented results concerned with all the tasks, besides presenting the proposed architecture. The task concerned with architectural issues has been finished by the end of month 24. The present deliverable is focusing on the presentation of middleware solutions and their implementation, being the architectural aspects just shortly addressed for ease of reference. It also integrates some aspects initially addressed in D2.1.2, namely the aspects concerned with the design of the several services, which are included for completeness reasons.

The deliverable is structured in one introductory overview section, six sections for the description of middleware services, and a concluding section. The initial section provides a general introduction that goes through all the work done in the scope of WP2, recalling the general system architecture proposed for HIDENETS and the node software architecture, and making a bridge to the work that has been done in the other HIDENETS work packages.

After that, the main technical sections are presented, with focus on the middleware services that have been designed, developed and implemented in HIDENETS. For some of them, evaluation activities have also been carried out to evaluate and validate the properties and the effective quality and usefulness of the service. These evaluation results are summarised here and are presented in more detail in WP4, which is specifically concerned with evaluation. Other services will be evaluated through the use of proof-of-concept prototypes. This “demonstration-based” evaluation is within the scope of WP6 objectives, and will therefore be addressed in the corresponding deliverables.

The services that are specifically addressed in this deliverable are the Diagnostic and Reconfiguration Manager, the QoS Coverage Manager, the Replication Manager, the Proximity Map, the Cooperative Data Backup and the Intrusion-Tolerant Agreement.

Regarding the specific issues addressed in each of these six sections, we have decided to structure the descriptions in four subsections. The first one provides a general description of the services and should be sufficient to let the reader understand the purpose and objective of the service, the context in which it can be used, and the kind of benefit that may be achieved by using it. The next subsection provides a description of design issues, specification of service properties and functionalities and, also, service interfaces and relations to other services. After that, more specific implementation details are provided. In some cases, and where adequate, these descriptions refer to the proof-of-concept prototypes in which the services are being used. In fact, this is a natural consequence of the fact that the implementations were done having in mind the objective of demonstrating their benefit while integrated in one of the test-beds defined and described as part of WP6 work. Finally, each of the sections is closed with a presentation of results (not necessarily final ones, as the proof-of-concept prototypes have not been fully completed at the time of this writing) and/or observations from the practical setups, and their respective discussion. Detailed discussion of prototypes and respective evaluation results will be presented in forthcoming deliverables D6.3 and D6.4. The deliverable concludes with some remarks about what has been done and some of the still open issues.

## 2 Middleware resilience solutions

In this section we provide some context for the work described in the deliverable. We start by reviewing the reasons underlying the decision of considering different operation domains, namely the infrastructure and the ad-hoc domain, which justify the fact that we have two “twin” deliverables, the present one and deliverable D2.2. Then, based on text from the earlier deliverable D2.1.2, we provide:

- A brief introduction to the architectural approach adopted in HIDENETS, which has an impact on the way services are organised (and on the reason why they are presented in this deliverable),
- an overview of the work previously done in WP2, namely the node software architecture which includes the services described here
- an introduction to the complex middleware services on which this deliverable is focused and an explanation of why these were the selected services, the ones that have passed from the initial descriptions to concrete implementations, integrated in partial test-beds.

### 2.1 Services for the Ad-Hoc domain

In HIDENETS, the ad-hoc and the infrastructure domains have been distinguished from the beginning (see e.g. the technical annex in the project proposal). The differentiation is necessary because the two domains clearly have different characteristics. E.g., car-to-car communication scenarios require mostly ad-hoc communication support and may not have access to any infrastructure domain. From a dependability point of view, communication performed in an ad-hoc environment poses a number of hard requirements not present in the infrastructure domain (e.g. dynamic behaviour of communication end points).

Some of the most challenging characteristics of ad-hoc domains are the following:

- Unreliable communication due to the presence of wireless links;
- uncertain operational conditions, due to operation with variable numbers of users or traffic flows and highly dynamic network topologies;
- variable failure modes, ranging from simple accidental faults to malicious faults (attacks and intrusions), due to operation in shared and weakly controlled environments.

On the other hand, it is possible to consider scenarios in which additional infrastructure support is available, and this support might be useful to design improved solutions for integrated car-2-car communication settings. In fact, while in general the infrastructure domain consists of a back-bone IP network connecting both service providers and service clients, we can consider that parts of the ad-hoc domain may be connected to the infrastructure domain via various different means (e.g. WLAN hot-spots, IP over GSM, IP over 3G or 4G networks). In this way, this connection from clients in the ad-hoc domain to a server in the infrastructure domain can be exploited to improve performance and/or dependability, using a different range of solutions than the ones designed specifically for ad-hoc environments.

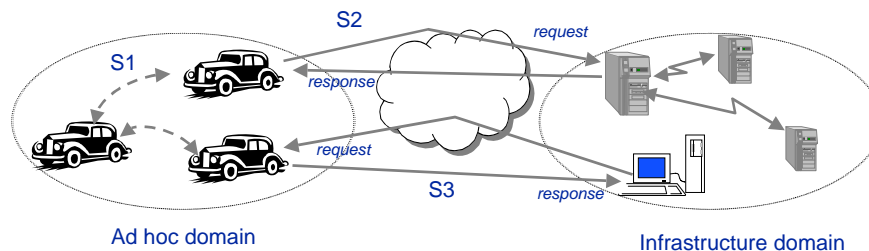
The different possibilities (or scenarios) that have been initially identified, and which the HIDENTS project has proposed to investigate, are illustrated in Figure 1, and can be described as follows:

- S1) Both the service providing and the service accessing entities are located in the ad-hoc domain. Note that this includes scenarios where the infrastructure domain is needed for connectivity, when the entities may not be within ad-hoc connectivity of each other. Examples of scenario S1 are the Platooning application and the first part of the Distributed Black Box application (the data distribution in the ad-hoc domain, see [5]), which have been explored in HIDENETS. In these applications we can observe the exchange of static and dynamic data obtained from the host vehicle, the road and other vehicles that participate in the traffic;
- S2) The service accessing entities are located in the ad-hoc domain and the service providing entities are in the infrastructure domain. Typically, this is the case for infotainment services which are accessed through the Internet, and also for emergency call scenarios. One particular case that we use in HIDENETS is once again the Platooning application, which may require all members to reach

agreement on a platooning speed, and this agreement can be achieved with the help of a server in the infrastructure domain, if present and available;

- S3) The service accessing entities are in the infrastructure domain and the service providing entities are in the ad-hoc domain. This scenario is covered by the Distributed Black Box application example, in which collaborative backup of critical data generated at the vehicular nodes has to be done and therefore requires vehicles to act as servers of collected data to a client that is in the infrastructure domain and collects this data.

Scenarios where both service providing and service accessing entities are located in the infrastructure domain have been explicitly left out of HIDENETS concerns. However, solutions for the other scenarios, e.g. S2 can in principle be applied to such a ‘standard’ scenario.



**Figure 1: Three service usage classes of scenarios.**

In terms of work structure, in WP2 we decided to distinguish between work on solutions possibly involving the infrastructure domain, and solution more focused for the ad-hoc domain. This was reflected in the creation of two tasks (task 2.2 and task 2.3) and two deliverables (the present one and D2.2). In fact, most of the work was done having in mind the challenging characteristics of the ad-hoc environments, and the middleware services that were developed are a direct consequence of this – they are included in this deliverable as part of the resilience solutions for the ad-hoc domain. In deliverable D2.2 [9] we provide some complementary work that is relying on the existence of an infrastructure domain.

## 2.2 Architectural approach

Differently from previous approaches, in HIDENETS we follow a hybrid distributed system model approach in which different parts of the system have different sets of properties and can rely on different sets of assumptions with respect to faults and synchronism. This has a number of advantages when compared to approaches based on homogeneous models, as explained in D2.1.2 [8] and in Reference [45].

One example of a hybrid distributed system model is the *Wormholes model*, named in this way after the astrophysics theory. This theory argues that one could take shortcuts, through, say, another dimension, and re-emerge safely at the desired point, apparently much faster than what is allowed by the speed of light. Those shortcuts received the inspiring name of *Wormholes*. In essence, Wormholes prefigure an ancillary theory which coexists with the classical theory, making possible subsystems which present exceptional properties allowing to overcome fundamental limitations of the systems under the classical theory.

The wormholes concept can in fact be instantiated in different ways. For example, when applied in the security domain, a wormhole takes the form of a security kernel, or a trusted component, as described in [12]. On the other hand, when timeliness is the relevant non-functional property to secure, the wormhole should essentially be timely. There are also examples of wormholes in the time domain. For instance, a very simple example of a timeliness wormhole can be a watchdog, which is able to shutdown a system or perform some other real-time action when a time bound is not met. Another example is a Timely Computing Base (TCB), as described in [46]. A system with a TCB has a control part, with synchronous properties, and a payload part, possibly asynchronous, in which the applications execute using a number of services provided by the former. A TCB constitutes an example of a timeliness wormhole.

The idea of an architecture based on the wormhole concept has been applied in HIDENETS. Therefore, a HIDENETS system can be subdivided in two parts, one “simple and trusted” and one “complex”. Each part

is characterised by its own system and fault model, where the first is typically stronger than the second (e.g., synchronous with a crash fault model, which is stronger than asynchronous with arbitrary faults of processes and communications). The stronger properties are secured by construction, i.e., the system is built in such a way so to make these better properties becoming true.

In summary, a simplified perspective of the HIDENETS node architecture is presented in **Fejl! Henvisningskilde ikke fundet.** The highlighted block corresponds to a separate realm of operation and represents the more timely and trusted part of the system. It is also possible to see that this realm of operation can be built over a separate hardware infra-structure. On the upper part of the architecture reside the complex resilience middleware services, that used the optimised network protocols and/or the trusted resilience services. The HIDENETS services can be accessed by applications through specific HIDENETS interfaces, while SAF interfaces could be used to access other availability services. A detailed description of this simplified node architecture figure can be found in D2.1.2 [8]. Here we want to point out that the solutions introduced in this deliverable are intended to be used in the general part of the system to improve some of the *Complex Resilience Middleware Services*. On the other hand, the description of the implementation aspects of the *Simple and Trusted Resilience Services*, which we also call *Timeliness and Trustworthiness Oracles* (the term ‘oracle’ being used here to convey the idea of a better service, which does not fails), is provided in deliverable D3.3 [16]. This is because these are services related to the provision of (possibly strict) timeliness and security properties, which is an issue covered in WP3.

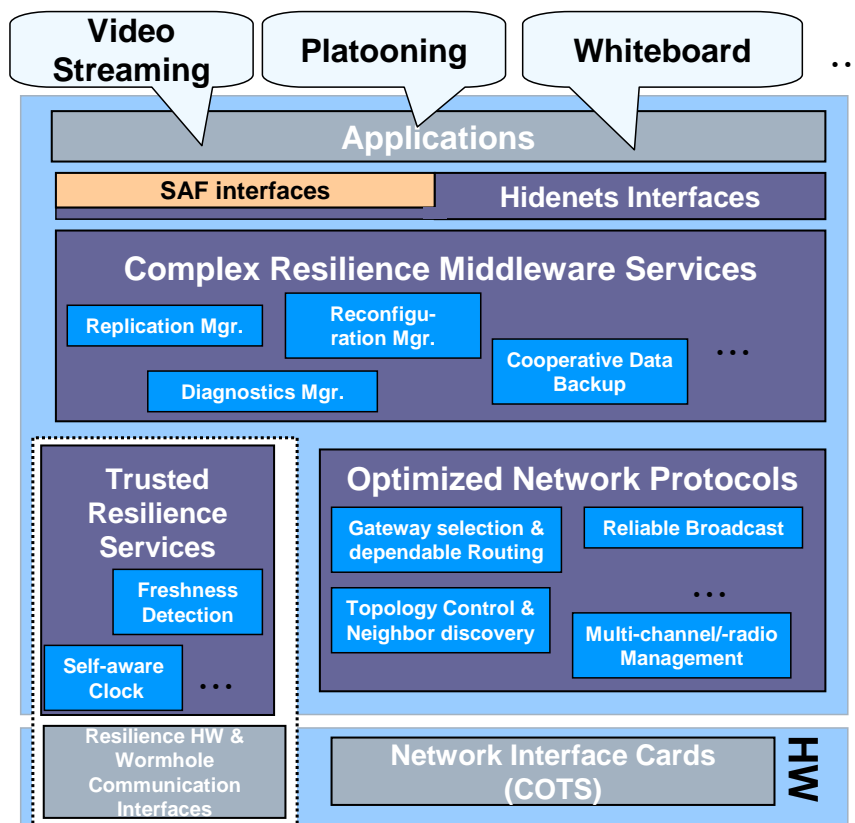


Figure 2: Simplified node architecture with Trusted Resilience Services.

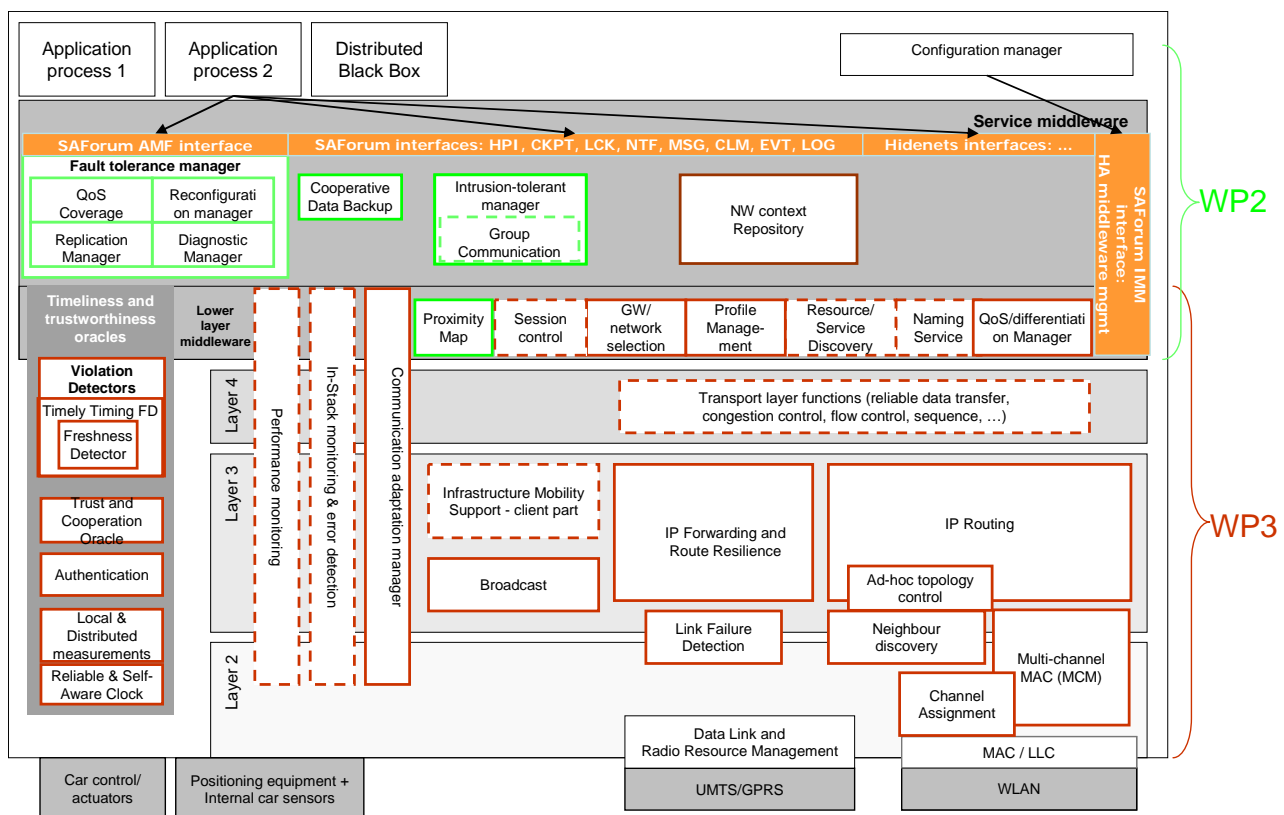
## 2.3 HIDENETS node software architecture

The complete node software architecture was developed by WP2 and WP3 together. The purpose of the node software architecture is to provide a coherent and structured framework where:

- i) The functional building blocks of a HIDENETS node are specified;
- ii) the relations between different building blocks are displayed;
- iii) the dependability related functions of interest to the supported applications are identified and made available through appropriate building blocks.

Figure 3 illustrates the building blocks, or services, that may be included in a HIDENETS node, while providing some additional information regarding the relationship of these building blocks within the node. This represents the final HIDENETS architecture, which has already been provided in deliverable D2.1.2 [8].

The layered structure as shown in the figure follows the standard ISO communication layer architecture. From the bottom, physical or hardware components are depicted including communication related components and other hardware that is relevant in the context of HIDENETS and that may be used in communication activities. This would correspond to layer 1 of the ISO standard. Then, several functional building blocks are depicted in layers 2 to 4, which offer typical network related functionalities. Some other services, like the In-Stack Monitoring and Error Detection, include functionalities that span across multiple layers. All these functional blocks are being addressed in the context of WP3 (as communication layer dependability improvements). Please note that some of the building blocks are represented with dashed lines. This means that although these blocks are considered necessary and are used by other services, they are not explicitly addressed in HIDENETS. That is, in HIDENETS we assume that existing implementations of these services are available and are adequate for the purposes of the project, and no particular improvement, modification or specification related to these services is done in HIDENETS. This is the case, for instance, for the Session Control service, the Naming service, and the Group Communication service.



**Figure 3: Overall picture of HIDENETS node software architecture**

On the left side of the figure, timeliness and trustworthiness oracles are grouped together. This separation is based on architectural principles established by the Wormholes model. We note that due to the nature of timeliness and trustworthiness oracles, which are supposed to provide strict timeliness and trustworthiness properties, the work concerned with their development and implementation was done essentially within task 3.3 of WP3. This is why they are not included in this deliverable, but are described in deliverable D3.3 [16]. We must note, however, that the initial work concerning the definition and design of these timeliness and trustworthiness oracles was done in the scope of WP2 and presented in the previous WP2 deliverables. We also note that time- or security-related functionalities may also exist outside the timeliness and trustworthiness oracles. E.g., applications may use standard system services like clock or authentication services. But since these services are not specifically related to dependability, we do not include them in this figure.

Above the communication services, we consider the existence of a lower middleware layer which is composed of building blocks that are mainly targeted at interactions between middleware services and communication services. All these services are addressed in the scope of WP3. The remaining building blocks, belonging to the group of middleware services, are supposed to handle higher level interactions with the applications. These middleware services are essentially addressed and developed in the context of WP2. The several blocks are coloured in red (darker) and green (lighter) to reflect the work package in which they are mainly addressed.

The HIDENETS effort for extended communication protocols focuses mainly on (congestion) failure-avoidance via efficient use of communication resources and on utilization of structural redundancy as given by multi-interface nodes and multi-network architectures (see D3.1.2 [42]). With respect to ensuring secure communication, it must be mentioned that for specific malicious faults, security solutions from other research projects [27] can be re-used. In particular cases (like for the implementation of Intrusion-Tolerant Agreement service, presented in 8.3), and also for practical reasons, IPsec [24] is utilized to ensure message integrity (and if needed, confidentiality).

For certain uses of applications and services, it may be necessary that an application fulfils specific requirements, e.g. performance related requirements, which translate into requirements for middleware and communication services. Such requirements are registered through high level middleware services.

In the figure the application interfaces are also represented, including the standard interfaces defined by the Service Availability Forum and also the interfaces that have been defined by HIDENETS. This illustrates the idea that existing services and standards should be used whenever possible, instead of redefining possibly redundant service interfaces. All of these interfaces are used by applications (and the Distributed Black-Box application is just an example for an application).

## 2.4 Complex resilience services

This deliverable is specifically concerned with the presentation of complex resilience services, targeting an operation in ad-hoc domain. These correspond, in fact, to the services presented in top part of Figure 3. Specific work concerned with the development of solutions targeted for environments involving communication to servers or nodes in the infrastructure domain, are addressed in deliverable D2.2 [9].

Complex resilience services are developed on top of a potentially asynchronous model. The idea is that they will use and will rely on the trusted resilience services if and when required, mainly for the execution of critical steps in their operation. The following complex resilience services have been considered in the scope of WP2:

- *Diagnostic Manager*. The Diagnostic Manager is in charge of monitoring critical components of the middleware and of lower levels in order to judge if the system is working properly or not. For this reason the Diagnostic Manager continuously assess the status of a selected set of sub-systems/components, aiming to discriminate acceptable from non-acceptable causes of their malfunctions/misbehaviours (e.g., transient from intermittent faults).
- *QoS Coverage Manager*. The QoS Coverage Manager interacts with the applications using

information possibly collected by the Diagnostic Manager. It has to assess if the application requirements are satisfied or not. E.g.: when the available QoS changes in run-time, the service calculates the new operational parameters that should be used or assumed by the application in order to keep a constant coverage of the requirement.

- *Reconfiguration Manager.* The Reconfiguration Manager handles reconfiguration procedures. The reconfiguration could be static (in presence of a particular class of faults the Reconfiguration Manager chooses a predefined strategy) or dynamic (many strategies can be defined, the one used is chosen basing on the best predicted impacts on the dependability of the application).
- *Replication Manager.* The Replication Manager handles the state sharing of stateful services provided by nodes potentially both in the ad-hoc domain and in the infrastructure domain. The manager provides an interface for service replication. The Replication Manager is selecting replicas based on estimates of network performance and based on fairness policies given at service deployment time.
- *Cooperative Data Backup.* The Cooperative Data Backup is in charge of managing backups of critical data despite failures of the data owner and of the nodes storing the critical data for the data owner. This service is responsible both for dissemination and for recovery of data.
- *Proximity Map.* The Proximity Map provides information on neighbouring nodes.
- *Intrusion-Tolerant Agreement.* The Intrusion-Tolerant Agreement is used to reach agreement among a well-known set of participating entities, even when some of these entities might behave in a malicious way.

## 2.5 From the definition to the implementation of complex middleware services

In the previous section we briefly introduced a set of services that have been selected in HIDENETS as important or relevant services to consider in the node software architecture, for the purpose of improving the resilience of applications in some way. We believe it makes sense to explain why we have selected this set of services, which we do in the following paragraphs.

First of all, it must be noted that the initial work in HIDENETS was concerned with the definition of a considerable set of applications and use cases (for those applications). This work, done in WP1 and reported in deliverable D1.1 [38], served as basis for the subsequent work in WP2, in particular to select a number of specific use cases and/or applications that could constitute good candidates in the sense of raising interesting challenges to be addressed by means of middleware services.

In the course of this process, the selected use cases were the Platooning use case, the Car Accident use case (including Distributed Black-Box application) and also the Assisted Transportation use case. Overall, these use cases involve a large set of dependability requirements, from availability to safety-criticality and including security. It was during this process that the set of required and relevant services became defined. Among other challenges that were identified, we mention here a few.

- In the Platooning application, if messages arrive at a platoon member with too much delay, this may result in too late reactions to a manoeuvre in the platoon, leading to safety problems. We investigated the problem in HIDENETS and we understood that by using a hybrid architecture in combination with a Timely Timing Failure Detection (TTFD) service we could ensure **appropriate means for timeliness** and a way to **secure safety despite potential lack of timeliness**.
- Also in the Platooning application, we observed that any information inside the platoon needs to be trustworthy. False messages may lead to traffic accidents. Therefore, in HIDENETS we **developed trust mechanisms**, in particular an Authentication service to be used in this concrete application, and a Trust and Cooperation Oracle that fulfils similar security goals and is in practice developed and demonstrated in the Distributed Black Box application. Related to security, and given the observed need or potential usefulness of an agreement service to be used by Platoon members, we devised an Intrusion-Tolerant Agreement service, which addresses at the same time the security concerns and the challenging task of performing agreement in an ad-hoc environment.

- Still in the Platooning application, we observed that this application presents high potential for adaptation, by adjusting the behaviour of the vehicles in order to cope with specific situations, including possible communication failures or simple degradation of the communication quality. Therefore, in HIDENETS we developed adequate solutions to **support the adaptation of the application** in a dependable way, which is done by using the QoS Coverage Manager, also demonstrated in the scope of this Platooning application.
- Within the Car Accident use case, and in particular in the case of the Distributed Black Box application, we identified a number of security related challenges. For instance, **confidentiality and privacy** are fundamental in this context, since the black box information should be accessible only from authorized parties. Only the original car should be allowed to write data, and only the data owner (or its delegates, e.g. its insurance company) should be allowed to read it. Also, **Integrity** of the black box information is important, and the original information produced by the car at the provider site should not be modifiable, either by the driver or by the other cars hosting copies of the original data, or by any third party. To address these requirements, and in addition to the Trust and Cooperation oracle already mentioned above, also the Proximity map service and the Cooperative Data Backup services were defined and implemented, and will be demonstrated in WP6.
- **Availability requirements**, which are implicit in all the considered use cases, have been considered in a more general way, not specifically focusing on one application. In HIDENETS we have devised three services which can clearly deal with the need for increased availability: A Replication service, a Diagnostic Management service and a Reconfiguration Management service. The Diagnostic and Reconfiguration Managers are also included in the Platooning proof-of-concept prototype and will be demonstrated in the scope of WP6.

After an initial definition of the HIDENETS services, the refinement work was started by taking into account the specific applications to be used as proof-of-concept prototypes, as described in WP6 deliverable D6.2 [5]. This also explains the close relation between the work developed in WP2 and the work in WP6. While in WP6 the work was concerned with the definition of the test beds and the specification of the proof-of-concept prototypes, in WP2 the work was concerned with the specification and development of the services. Additionally, in order to be able to finally integrate some of the services in particular test-beds, it was necessary to harmonize the interface specifications and have a more coherent view of all the services and their interconnections. This was done with the help of the methodologies developed in WP5, which took initial interface and service specifications and reported the necessary changes that would have to be made in order to achieve the desired uniformity.

Finally, we must mention that in the course of the project, and in particular during a refinement phase in the beginning of the second project year, there were some changes in terms of the considered services. For example, we mention a Freshness Detection service, that was initially considered as an independent service, but that was finally integrated as one particular instance of the Timely Timing Failure Detection service, and therefore not developed as a service on its own. We mention also the Diagnostic Manager and the Reconfiguration Manager services that, while described separately in former deliverables, from the point of view of implementation have been addressed as a single block and, in this sense, are presented in a single section of this deliverable.

### **3 Diagnostic and Reconfiguration Manager**

The functionalities of the Diagnostic Manager (DM) and Reconfiguration Manager (RecM) services are described in the general context of HIDENETS in [8]. This chapter describes the DM and RecM services which are tailored for usage within the ad-hoc domain by taking into account the specificities of this scenario.

#### **3.1 Service Description**

The role of the DM is to judge if the system is working properly or not. For this reason the DM continuously assesses the status of a selected set of sub-systems/components, aiming to discriminate acceptable from non-acceptable causes of their malfunctions/misbehaviours (e.g., transient from intermittent faults). The DM basically performs the following activities:

- Gathers data over time on the correct/deviated behaviour of the monitored components as perceived by appropriate error/deviation detection mechanisms the system is equipped with;
- Applies a diagnosing function to the gathered data in order to assess the status of the monitored components and identify the active faults.

DM judgments are used by the fault removal mechanisms (e.g. the RecM), which deal with how to prevent the diagnosed fault to be activated again. Typically, isolation of the faulty/deviated components is performed, possibly followed by a replacement with a well-working spare.

The RecM is in charge of managing system reconfigurations (selecting both the time of reconfiguration and the proper reconfiguration policy to be applied) on the basis of information coming from the system (e.g. from “DM” or from “QoS Coverage Manager”) and/or from application needs (e.g. applications organised in several different operational modes). The RecM aims to:

- Bring back the system to provide correct (although possibly degraded) services after the occurrence of some malfunctions.
- Properly manage system resources in order to provide the required QoS levels (if possible) after the occurrence of some deviations from expected QoS.
- Change system reconfiguration on the basis of the operational mode required by the running application.

RecM works on-line, basically performing the following activities:

- Gathering information about the status of the entities monitored by the DM.
- Gathering information about the evaluated QoS levels from the “QoS Coverage Manager”.
- Selecting the proper reconfiguration of components/system (if any) based on the gathered information. The choice of the reconfiguration action is guided by the expected benefit of applying it, obtained through a quantitative evaluation support (e.g., implementable as part of WP4).
- Selecting the current operational mode (if any is defined) and the corresponding configuration.
- Triggering the proper actuators to put in place the selected reconfiguration.

#### **3.2 Design, Specification, Interface**

When instantiating the DM and RecM services within the ad-hoc domain, the properties specific to this domain need to be taken into account. The relevant ad-hoc domain properties that could impact the diagnosis and reconfiguration activities include the following:

- End-to-end wireless communication. The HIDENETS ad-hoc domain deals with an autonomous collection of mobile nodes (cars) which communicate using wireless links. Wireless communication has variable, highly dynamic and unpredictable characteristics (the signal strength and propagation conditions fluctuate with respect to time and environment) and hence the quality of communication between two interacting nodes can vary significantly over time. Moreover, node mobility leads to a

- continuously rapid and unpredictable change in the communication topology, so that existing routing paths break and new ones form dynamically over time.
- Dynamic groups of nodes. The ad-hoc domain does not contain any fixed infrastructure and there is no central administration, so coordination within a group of collaborating nodes needs to be either distributed or based on the election of a group leader. Wireless links and mobility lead to some issues about coordination within a group: unpredictable number of group members (the number of nodes geographically close together is unpredictable), no prior trust relationships among nodes, possible partitions in the group.
  - Use of small low-cost COTS devices. The devices used to support the HIDENETS platform should be small low-cost COTS devices which, in principle, could be very heterogeneous. This heterogeneity leads to unpredictability when it comes to their computational, storage and communication capabilities. Services/applications should adapt to the specific capabilities of the device on which they are running. The use of COTS components requires a black-box design approach, so the granularity of the faulty units is at most at the COTS component level.

The design organisations envisioned in [8] to satisfy the different needs for diagnosis in HIDENETS were the following:

1. “Local Diagnosis”: Diagnosis on local resources/services, aimed to assess the status of some components/services inside the node (a sort of node auto-control).
2. “Private Diagnosis”: Diagnosis on remote node/services, aimed to quantify the local perception of some remote node/services, based on information available in the node.
3. “Distributed Diagnosis”: Diagnosis performed in a distributed way within a group of collaborating nodes when an agreement (Byzantine resilient) about some global status of the group is necessary.

The properties specific to the ad-hoc domain affect each of the above diagnosis (and reconfiguration) scenarios for different reasons.

Local diagnosis is continuously performed on the same node component/sub-system, and the major influence coming from the properties of the ad-hoc domain derives from the use of small low-cost COTS components: the presence of COTS components negatively impacts the granularity of the diagnosis itself.

Private diagnosis is mainly affected by the end-to-end wireless communication, because the unpredictability of communication deeply impacts the perception of remote events as requests for timely remote services could locally correspond to late or missed responses, regardless the effective result of the remote service. Another relevant parameter affecting private diagnosis is the duration of the interaction between the local node and the remote component/service. The dynamicity of the ad-hoc domain leads to frequent interactions between parties with no prior trust relationships, so this scenario is not adequate for accruing long time history about node/service behaviour. As the duration of the interaction between the parties could be very short, the diagnosis should be prompt, possibly impairing its accuracy (an unavoidable trade-off exists between diagnosis promptness and accuracy).

Distributed diagnosis is an agreement within a group of nodes regarding the private diagnosis of a certain node/service; in addition to the above discussion about the determination of a private diagnosis of a certain node/service, issues are related to the group membership and partitioning

In each of the above scenarios, the DM mainly needs to acquire error/deviation information from error/deviation detectors related to the monitored component/node. Focusing, for example, on the first scenario (local diagnosis), in principle any local error/deviation detection mechanism, from packet-level CRC to application-level exception handlers, is an eligible feeder of DM. In order to make such information available to DM, proper interfaces towards DM should be set up. Alternatively, such signals could be conveyed and stored in a repository, accessed by DM when necessary.

The following services/oracles of the HIDENETS middleware are potentially useful sources of error/deviation information for DM. Which ones are relevant strictly depends on the specific entity to be diagnosed, the faults it may be affected by, and what the consequences may be of such assumed faults.

- “Network Context Repository”, through which it may get specific error signals coming from various components in the node.
- “In-stack monitoring & error detection” (described in WP3), provides information related to monitoring and error detection carried out inside the communication protocol stack.
- “Performance Monitoring” (described in WP3), provides to get information about performance of specific components/services in the system (e.g. to infer the performance of a link or a path in the network).
- “Timely Timing Failure Detection”, useful to detect whether timed actions have been executed in a timely way or if they have incurred in a timing failure.
- “Authentication Service”, useful to detect whether communicating parties are really who they claim to be.
- “Trust and Cooperation Oracle”, useful to detect whether neighbouring entities involved in cooperative services are trusted or not.

The relationship between the DM and the above listed sources of error/deviation information related to the monitored entity can be proactive, reactive or something in the middle (e.g. event triggered). In a proactive schema, the DM gathers available detection results as soon as the error/deviation-detection mechanisms produces them, whilst in a reactive scheme, detection is explicitly requested by the DM. Note that, in absence of external error/deviation detection sources, the DM could be realized to also play this role, by running checks on the monitored entity (periodically, or in conjunction with specific events).

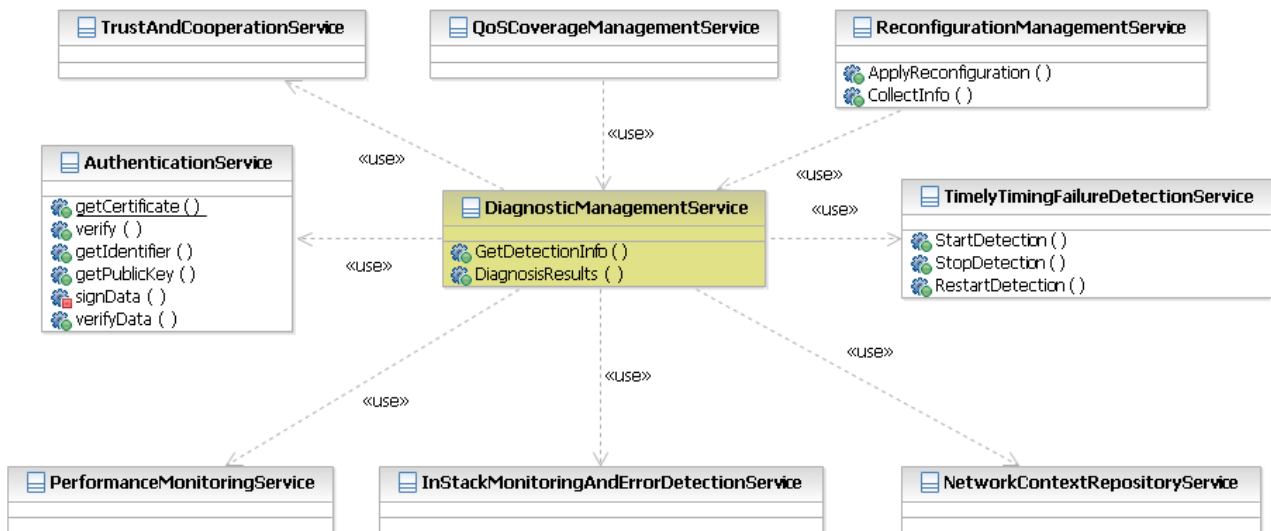
Of course, for all the information received, DM has to be aware of the “accuracy” of such information, mainly determined by the coverage factor of the component that has produced it (e.g., in case of the error detectors, the coverage factor is representative of their ability to detect malfunctions when malfunctions are present, and to avoid detecting a non-existing problem).

Diagnosis assessments made by the DM component are useful mainly for the following entities:

- “Reconfiguration Manager”, which has to handle the faulty entity, typically by applying a suitable reconfiguration strategy with respect to the diagnosed scenario. Example: if a transient fault is diagnosed in the computational part of a portable device, it is proper to retry some operations rather than to impose a full reset of the device.
- “QoS Coverage Manager”, which has to perform some probabilistic analysis, based on diagnostic data and performance historical data.

Concerning the relationship between the applications and the DM, we assume that the DM service is transparent to the application. The DM service offers a functionality that is not directly exploitable by an application but helps to offer to an application a more reliable HIDENETS platform to run on. The choice of which DM instances are in place, and which monitored entities these are associated with, is a choice completely internal to the HIDENETS instance. So, we consider that a HIDENETS instance can operate in different operational modes during its life (due to specific needs of the application it is supporting, or because reconfigurations have been performed due to experienced malfunctions), and each mode is characterised by the presence of specific instances of the DM service (possibly, no one).

An overview of the service interface and relations to other HIDENETS services is shown in Figure 4.



**Figure 4: Interoperation diagram for the DM Service.**

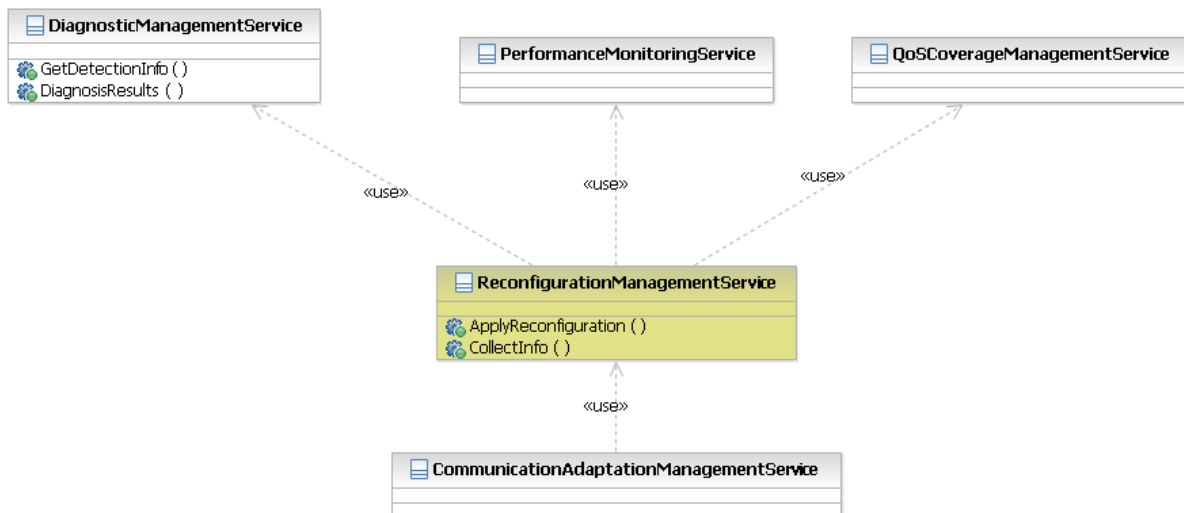
The RecM needs input from the following modules:

- “Diagnostic Manager”, in order to get diagnostic information about the status of modules/services which can be subjected to reconfigurations.
- “Performance Monitoring” (developed within WP3), in order to get information about performance of specific components/services in the system (e.g. to infer the performance of a link or a path in the network).
- “QoS Coverage Manager”, in order to know whether the QoS application requirements are satisfied or not.
- (Possibly) services from the Timeliness and Trustworthiness oracles, in order to be able to reconfigure within specific temporal bounds.
- Input from the application, which notify the RecM about changes to its operational mode (if any) requiring specific system configurations.

The output of the RecM is used by the following components/services:

- “Communication Adaptation Manager” (developed within WP3), to request specific communication adaptations.
- Application, to notify changes in system configuration which can correspond to changes in the operational mode (if any) of the application itself.

An overview of the service interface and relations to other HIDDENETS services is shown in Figure 5.



**Figure 5: Interoperation diagram for the RecM Service.**

### 3.3 Implementation

This section presents a specialisation of the DM and RecM services in the Platooning Test-bed; the specialisation focuses on a restricted set of functionalities related to the local diagnosis and reconfiguration aspects. This section is intended to provide only an overview of the specialisation of the DM and RecM services. More details will be given in the D6.4 HIDENETS deliverable.

Each vehicle used in the Platooning Test-bed is assumed to have two GPS receivers. The DM and RecM middleware services are used to manage the (active) redundancy of GPS receivers in order to increase the availability of GPS information to the Platooning application.

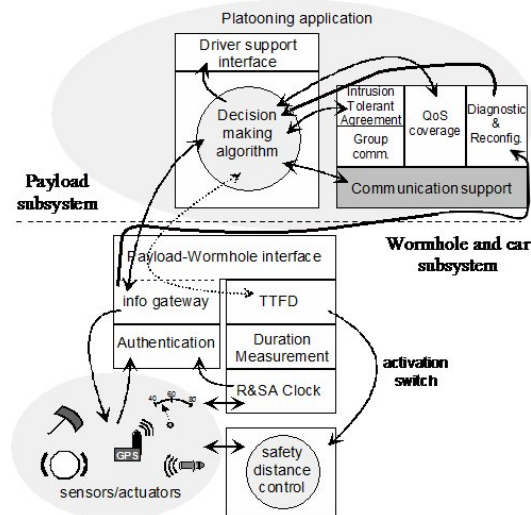
For the scope of the demonstration, the DM and RecM services are fused in one box named “DM+RecM”. The “DM+RecM” service is in charge of filtering the information flows produced by the GPS receivers in order to construct a unique consolidated information flow to be made available to the rest of the middleware and eventually to the “platooning application”.

#### 3.3.1 System description and assumptions

The vehicle running the Platooning application has two GPS receivers working in parallel (the receivers could be identical or different). Each receiver proactively produces a flow of information packets (*frames*) containing the evaluation of some physical measures related to the movement of the vehicle on which the receiver is installed (e.g. following the NMEA0183 protocol). Each frame contains information related to both time and space:

- Time reference  $t$ .
- Estimated position at time  $t$ .
- Estimated instantaneous speed at time  $t$ .
- Some information about the precision of the estimation of the above data.

All the GPS receivers are directly connected to the wormhole, as shown in the following Figure 6.



**Figure 6: Architectural view of the relationships between the DM+RecM service and the HIDENETS middleware in the Platooning application.**

Each GPS receiver proactively sends frames to the wormhole, and each frame received by the wormhole is first time-stamped by the “R&SA Clock” service and then forwarded to the payload by the way of the “info gateway” service.

Let  $F_1$  and  $F_2$  be the information flows generated by the first and the second GPS receiver respectively, and let  $f_i(t)$  be the frame generated at time  $t$  by the  $i^{\text{th}}$  GPS receiver.

Each not faulty receiver is assumed to generate a frame (about) every second. The receivers produce frames (if any) in the same instant of time.

The “DM+RecM” service, which resides in the payload, is in charge of gathering (up to) two information flows ( $F_1$  and  $F_2$ ) forwarded by the “info gateway” service and *consolidate* them in order to produce a unique flow ( $F$ ) to be made available to the rest of the middleware (in particular to the “decision making algorithm”). The “DM+RecM” service also signals the requirement of maintenance operations on receivers diagnosed as permanently faulty (these signals could be displayed on the vehicle’s dashboard).

### 3.3.2 Fault Model

The faults assumed to affect a GPS receiver are the following:

- **Crash:** The receiver stops sending any information to the wormhole (it seems the receiver is disconnected from the wormhole). This is a permanent fault requiring a maintenance operation.
- **Value:** The receiver produces wrong values (e.g. due to multi-path interferences caused by reflected GPS signals arriving at the receiver, typically as a result of nearby structures or other reflective surfaces).
- **Omission:** The receiver is not able to give the position of the vehicle (e.g. when the vehicle is within a tunnel). This is a transient fault; there is nothing that can be done to solve the problem.

### 3.3.3 Diagnosis and reconfiguration

This section explains how information coming from all the available GPS receivers is managed in order to produce a unique information flow for the rest of the middleware (in particular to the “decision making algorithm”).

The “DM+RecM” service receives as input (up to) two information flows from the “info gateway” service ( $F_1$  and  $F_2$ ) and outputs an information flow ( $F$ ) made up by frames containing “consolidated” time and position values.

The “DM+RecM” service has to continuously:

- This step consists in checking the *plausibility* of each frame received and *merging* the plausible results in some way, in order to produce the frames of the output flow ( $F$ ). The definitions of “*plausible frame*” and “*merging frames*” are given in the forthcoming text.
- Perform reconfiguration: This step consists in dropping information which seems not plausible and/or signalling the need for maintenance on a GPS receiver diagnosed as faulty (possibly disconnecting the faulty receiver).

Given that there are two GPS receivers in the vehicle, there are three different situations to cover:

1. There are no working receivers, so there is nothing to check against plausibility or to merge.
2. Only one receiver is sending information to the “DM+RecM” service, so there is the need for checking the plausibility of the received information.
3. Both the receivers are sending information to the “DM+RecM” service, so the received information needs to be checked against plausibility and “merged” in some way.

If no information flows are available, an “empty” frame (containing “no information”) is forwarded to the application, so that the application is informed about the unavailability of the GPS signals.

If only one information flow is available, the plausibility check on the last frame received must be performed; if the last frame emerges to be plausible, it is forwarded as it is, otherwise an “empty” frame (containing “no information”) is forwarded, so that the application is informed about the unavailability of the GPS signals.

If two information flows are available at time  $t$ , both the frames received at time  $t$  are checked against plausibility. If they aren’t both plausible, one of the actions described above is performed. Otherwise, if they are both plausible, they are merged in order to forward a unique frame labelled with time  $t$  to the application.

### 3.3.4 Plausibility Check

Position, speed and time values are changing over time. If the information contained within a certain frame  $f(t)$  is assumed correct (within certain bounds), it is possible to check whether the information contained in a subsequent frame  $f(t+\Delta t)$  is plausible or not, based on the fact that the movement of the vehicle must follow physical laws (position and speed values contained in frame  $f(t+\Delta t)$  must be within certain bounds, given the amount of time  $\Delta t$  passed between the two frames). Bounds depend on all the following information:

- Position, speed and time reference contained in the first frame  $f(t)$ ;
- The time  $\Delta t$  between the two frames considered;
- The physical laws describing the relationships between, time, space and speed.

When two GPS receivers are installed in the same vehicle, two information flows are available. Given that both the information flows describe the physical movement of the same vehicle (the relative position of the two receivers is assumed to be constant and known), both the flows can be used in the plausibility check and both contribute in consolidating the frames to be forwarded as output.

The coverage of the plausibility check is not 100%.

### 3.3.5 Merging frames

Two plausible frames  $f_1(t)$  and  $f_2(t)$  are merged in a frames  $f(t)$  using the following criteria:

- Time reference: the time reference  $t$  is used.
- Estimated position at time  $t$ : mean position between position given in frames  $f_1(t)$  and  $f_2(t)$ .
- Estimated instantaneous speed at time  $t$ : mean speed between speed given in frames  $f_1(t)$  and  $f_2(t)$ .

### 3.3.6 Implementation in C++

This section contains an overview of the implementation of the “DM+RecM” service, customised within the Platooning test bed. More implementation details will be given in the D6.4 HIDENETS deliverable.

The “DM+RecM” service, as customised in the Platooning test bed, is based on the following C++ class:

- “dmrecm”, which is the actual front-end of “DM+RecM” service managing the redundancy of GPS receivers;

The “dmrecm” class uses a “gpsData” data structure, which contains the GPS data presented above. The GPS data structure is composed of three float numbers related to time, position and speed:

```
typedef struct{float time; float position; float speed;} gpsData;
```

The main functions of the “dmrecm” class are the following:

- “gpsData merge(gpsData inputData1, gpsData inputData2)” merges the received GPS information (“inputData1” and “inputData2”) following the criteria explained in Section 3.3.5;
- “char DiagnosisResultGPS1( )”, which reveals the current diagnosed status of the first GPS receiver (as described in Section 3.3.3);
- “char DiagnosisResultGPS2( )”, which reveals the current diagnosed status of the second GPS receiver (as described in Section 3.3.3).

## 3.4 Results, Analysis and Discussion

The “dmrecm” class overviewed in the previous section is under integration in the Platooning Test-bed. The (expected) benefits given by the use of the “dmrecm” class will be experimentally evaluated once the integration will be successfully completed. Given that the Platooning Test-bed is not yet completed, experimental evaluation results will be given in the 6.3 HIDENETS deliverable.

## 3.5 Summary

This section briefly summarises the design choices and achieved results related to the Diagnostic Manager (DM) and Reconfiguration Manager (RecM) services.

The main activity about the DM and RecM services was the identification of a framework for system diagnosis and reconfiguration, trying to identify and address the relevant problems that need to be solved. The main role of DM and RecM in HIDENETS is the improvement of resilience and availability of services, despite inevitably changing user/application needs, resource availability and system/component faults. Aiming to address this objectives, three diagnostic scenarios were identified, trying to taking into account both the local and the global point of view: local diagnosis (performed by a node inside the same node on local resources/services), private diagnosis, (performed by a node on a remote node/service basing on the private perception of the remote node/service) and distributed diagnosis (performed in a distributed way among a set of collaborating nodes when an agreement about the healthy status of each node is necessary). The following problems were identified: the existence of wrong detections, due both to the imperfection of detection mechanisms themselves and to the deviated perception of remote activities (e.g. because of communication problems), the presence of malicious faults, aimed to distort the perception of the various system parts (e.g. a node tries to persuade other nodes that a certain server is congested in order to be the sole to gain its services), the cost of distributed diagnosis (despite of the use of an authentication mechanism) and the high dynamicity of group of nodes (especially in the ad-hoc scenario).

Further activities related to the Diagnostic Manager (DM) and Reconfiguration Manager (RecM) services are planned in the HIDENETS Platooning Test-bed. The “dmrecm” class (described in Section 3.3), which specializes a restricted set of functionalities of the DM and RecM services within the Platooning Test-bed, will be integrated with the rest of the code, and experimental evaluations of the service prototype will be

performed, aiming to quantify the (expected) benefit of using it. The details of the above mentioned activity will be documented in the D6.4 HIDENETS deliverable.

Future steps beyond HIDENETS are foreseen for the DM and RecM services, aiming to better integrate the diagnosis activities with the reconfiguration ones; for example, in a dynamic reconfiguration scenario (when many reconfiguration strategies are applicable for the same diagnosed scenario) the choice of the reconfiguration to be applied is performed on line, through a proper evaluation support. Work is needed to identify which specific system and environment conditions have to be fed to the evaluation support at the time the reconfiguration action is triggered by the DM subsystem.

## 4 QoS Coverage Manager

Computer systems and applications are becoming increasingly distributed and we witness the pervasiveness and ubiquity of computing devices. This openness and complexity means that the environment tends to be unpredictable, essentially asynchronous, making it impractical or even incorrect, to assume any time-related bounds. On the other hand, there are increased concerns about the dependability of these systems and applications, in the sense of their ability to meet some specified quality of service (QoS) levels. One possible way to cope with the uncertain timeliness of the environment while meeting dependability constraints consists in building adaptive applications and ensure that they adapt in a dependable way, that is, they remain correct as a result of adaptation.

The QoS Coverage Manager, sometimes also mentioned as QoS Coverage Management Service or simply QoS Coverage Service, will be in charge of evaluating if the QoS requirements of an application are satisfied, or else if it is necessary to provide an indication that QoS may no longer be guaranteed and may need to be renegotiated. In order to manage the requests, the QoS Coverage Manager performs two activities: characterisation of the current environment conditions, using probabilistic mechanisms; and verification if application requests are satisfied, providing the necessary indications of QoS variations when needed.

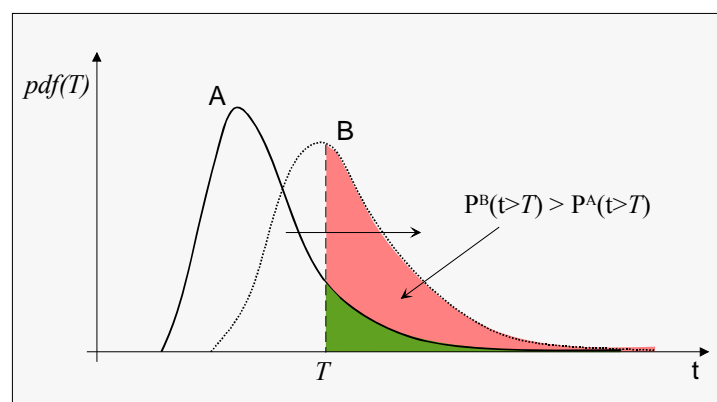
This section presents the QoS Coverage Manager, including service description, implementation, statistical mechanisms applied to perform probabilistic analysis, and main conclusions from experimental results.

### 4.1 Service Description

The QoS Coverage Manager objective is to provide enhanced support for applications to adapt to the available QoS, which is **dependable in spite of the uncertainty of the communication environment**. In essence, the idea behind dependable adaptation is to ensure that a **coverage stability** property is satisfied, that is, that the assumed bounds for fundamental variables (e.g. network delay) are secured with a known and constant probability. Note that the fundamental idea behind this service, the ability to dependably adapt to the available QoS, can be applied not only to the QoS of communication subsystems, but also to the QoS of, for instance, the processing subsystem. Here we are mostly concerned, however, with communication uncertainty.

Classical approaches for QoS provision assume that resource reservation is possible [41][18][47]. However, since we assume environments that are unpredictable and that can change dynamically, nothing can be guaranteed about timeliness or the amount of available resource. In such environments, the best that can be done is to adapt applications to the amount of resources available in a particular time instant.

The concept of dependable adaptation has been presented in previous work [7], as well as a general approach for applying it. The basic idea is to compute a probabilistic distribution function (*pdf*) based on a sufficient number of samples that were obtained through measurements of, for instance, message delivery delays. Since the state of the communication subsystem can vary over time, this will be reflected in a modification of the calculated *pdf*. This is illustrated in Figure 7, which depicts two distinct intervals of observation of a timing variable.



**Figure 7: Example of variation of distribution with the changing environment**

While in a first moment, characterised by *pdf* A, the cumulative probability of violating time bound T is given by  $P_A$ , when the state degrades and the distribution changes, being characterised by *pdf* B, the same time bound could be violated with an higher probability, given by  $P_B$ . The objective of adaptation is then to choose a new time bound, which ensures that the probability of failure can be kept stable throughout the execution. The output of the QoS Coverage Manager service would in this case be the new time bound.

In HIDENETS, the availability of a QoS Coverage Manager service is particularly important for applications that can adapt their behaviour depending on the available QoS, expressed in terms of the timeliness of the communication. For example, in the Platooning application, in which it is necessary to compute a target speed that is safe and, at the same time, as high as possible, the computation may take into account a certain expected communication delay. Without the QoS Coverage service the trade-off would be the following: by assuming a slow network with large communication delays (therefore using large bounds/timeouts), the platoon speed would be small but the probability of timing faults (expired timeouts) would also be small and the platoon would progress smoothly, without the need for the activation of any safety mechanisms (when timeouts expire). On the other hand, assuming lower communication delays may allow increasing the speed, while increasing the probability of the occurrence of timing faults and consequent activation of safety mechanisms (which would ultimately imply a less smooth behaviour of the platoon). The right compromise should be achieved by specifying a certain probability that the assumed bound/timeout will be satisfied (coverage), and the QoS Coverage service will provide this bound.

For adaptive applications, rather than adapting in some ad-hoc fashion, for instance by doubling timeout values as soon as a timeout is exceeded, it is much more interesting to perform the adaptation based on information obtained through the use of analytical formulations and detailed modelling of the environment, as provided by some underlying service like the QoS Coverage service. From a **dependability** point of view, the adaptation process should be done in a **controlled way**, that is, with exact knowledge of *how much* and *when* should it take place. This is only possible with an **accurate characterisation** of the actual conditions of operation and the available resources at a certain moment.

## 4.2 Design, Specification, Interface

The QoS Coverage Manager service is designed to be composed by two activities: Identification of the current environment conditions and, consequently, QoS adaptation.

Environment recognition - The environment conditions can be inferred by analyzing a real-time data flow representing, for example, the end-to-end message delays in a network. When the analytical description of the data is not known, we need to determine the model that best describes the data. Using statistics the data may be described by a probability density function (*pdf*) or by a cumulative distribution function (CDF). We note that the data models can be so complex that they can not be described in terms of simple well-know probabilistic distributions. In the case in which the model that describes the data is known, the problem is reduced to estimating unknown parameters of a known model from the available data.

QoS adaptation - Once the most fitting distribution (together with its parameters) has been identified, its statistical properties can be exploited to find a pair <bound, coverage> that will satisfy the objective of keeping a constant coverage of the assumed bound throughout the execution.

We are assuming an interleaved probabilistic behaviour of the environment. Therefore, we can consider that the system alternates between periods during which the conditions of the environment remain fixed (stable phases) and periods during which the environment conditions change (transient phases). In the first case, the statistical process that generates the data flow is under control and then we can compute the corresponding distribution using an appropriate number of samples; On the contrary, if the environment conditions are changing, then the associated statistical process is actually varying, so no fixed distribution can describe its real behaviour.

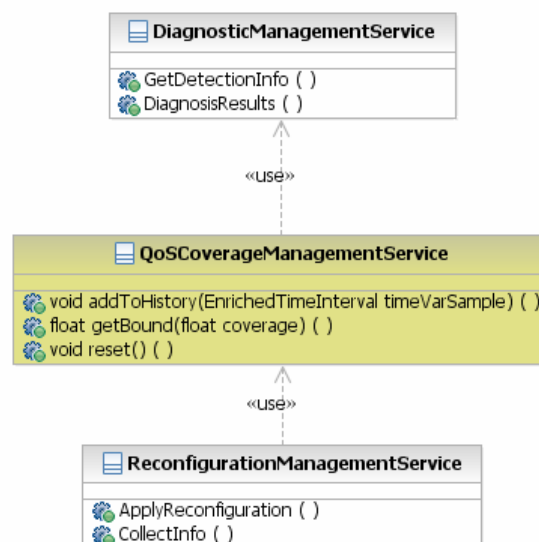
During the transient phases we adopt a *pessimistic approach* and we set the pair <bound, coverage> using a probabilistic formulation based on the one-sided inequality, which provides a pessimistic bound, but which holds for all probabilistic distributions. As soon as the presence of a stable phase is detected, a proper *pdf* is identified and then an improved (lower) bound can be computed according to the new distribution (*optimistic approach*), still ensuring the coverage stability property. The bound adaptation is then triggered by the detection of a new stable/transient phase.

The QoS Coverage Manager will provide two main functionalities to applications:

- Request monitoring of the coverage of a QoS (temporal) requirement: It allows setting up a monitoring activity, which will be used to provide information and indications to help an application adapt in a dependable way, that is, to let it adapt in order to meet a specified coverage criteria.
- Request for indication of a QoS change: Block the application (or thread) waiting for an indication from the manager, which will contain updated information relative to bounds to be used.

Applications have to inform to the service the bound that must be observed and its respective desired coverage. Moreover, for the QoS Coverage service to work properly it is necessary that it knows when to inform the application that a bound needs to be adapted. It would be possible to let the service provide information whenever a new *pdf* was built, independently of the variation degree. But in the case of applications that define QoS levels and are able to adapt among these levels, this may cause unnecessary performance degradation due to the potentially large number of QoS change indications for small variations that would not result in adaptation by the application. To prevent this situation, the application can specify an allowed interval of variation of the monitored bound, which will be used to determine when the deviation relative to the current bound is sufficient to trigger the delivery of a QoS change indication.

An overview of the service interface and relations to other HIDENETS services is shown in Figure 8.



**Figure 8: Interoperation diagram for the QoS Coverage Manager service.**

It is possible to see that the QoS Coverage Manager service might be used by the Reconfiguration Manager in case the latter wants to detect some specific QoS change in order to reconfigure the system. Note, however, that this relation between the two services is not strictly required and that the QoS Coverage Manager service would be used just as it is used by any user application.

On the other hand, the QoS Coverage Manager connection with the Diagnostic Manager is justified by the fact that the latter could possibly provide input information, possibly already pre-processed, to serve as the basis for the operation of the QoS Coverage Manager. In this case, the Diagnostic Manager would be in charge of collecting information about the relevant observed temporal variables, building histories with the collected information, and making all this information available to the QoS Coverage Manager. In fact, other services related to monitoring functions could also be considered as possible services to be used by the QoS Coverage Manager. In practice, the implementation can be made efficient by letting the QoS Coverage Manager itself collect the necessary information.

The QoS Coverage Manager internal design is hidden from the service's clients. To use this service, an application should instantiate an object of the *QoSCoverage* class, providing the history size. The history size defines the number of measurements recently collected that will be considered to characterise the environment and compute a new bound.

To feed the service's history, applications should use the *addToHistory* method. This method's parameter is the collection of measured bounds that will be used as input samples by the service. In fact, this method allows decoupling the monitoring activities from the processing part, since the monitoring could be done by any external entity that would then call this method to provide the necessary input. In practice, the QoS Coverage Manager itself may do the monitoring (e.g., with some independent thread) and call this method directly. Given that the history has sufficient sample points, a new bound can be produced. A new time bound is obtained by calling the *getBound* method, which receives as parameter the required coverage. In fact, an application would have to call this method repeatedly (or periodically) to become aware of relevant changes. A practical implementation would free the application from this task, allowing this periodic evaluation of the bound/coverage pairs to be done by the QoS Coverage Monitoring service itself, namely every time a new input sample is provided and added to the history, triggering the re-evaluation of the environment state. Then some callback would have to be setup, so that the application could be informed of relevant changes.

If the client application needs to reset the history in order to start a new computation, it can use the *reset* method or instantiate a new *QoSAdaptation* object.

### 4.3 Implementation

The scheme depicted in Figure 9 shows the current QoS Coverage Manager implementation: (i) It accepts the history size (i.e., the number of collected samples of the random variable under observation) and the required coverage as dependability related parameters; (ii) it reads samples/measured delays as input, using them to fill up the history buffer that is used by the phase detection mechanisms and for the estimation of distribution parameters; and (iii) it provides, as output, a bound that should be used in order to achieve the specified coverage.

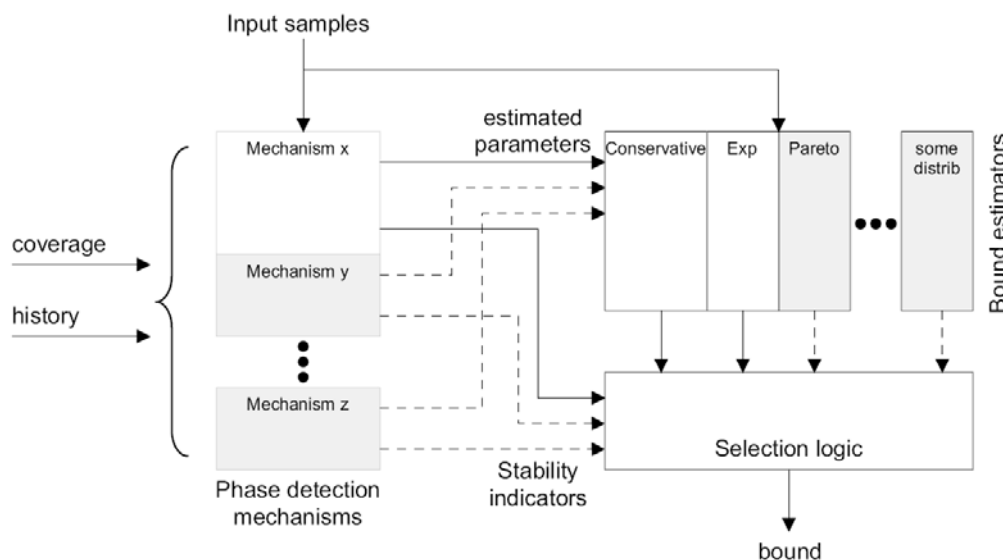


Figure 9: QoS Coverage service.

#### 4.3.1 Phase detection mechanisms

A phase detection mechanism must detect the beginning of a new transient phase, as soon as the environment conditions start changing (“changing environment” detection time), as well as the beginning of a new stable phase, as soon as the environment conditions stabilise (“stable environment” detection time). The phase detection mechanism analyses the last  $N$  measured delays (history and input samples in Figure 9) to verify if the environment is stable or in a transient period. If in a stable period, it determines the distribution that best characterises the environment.

We implemented two goodness-of-fit (GoF) tests as phase detection mechanisms: the Kolmogorov-Smirnov (KS) test and the Anderson-Darling (AD) test. GoF tests are formal statistical procedures used to assess the underlying distribution of a data set. Both AD and KS are distance tests based on the comparison of the assumed distribution CDF and the empirical distribution function (ECDF), which is a CDF built from the input sample: If the assumed distribution is correct, the theoretical CDF closely follows the empirical CDF. It should be noted that in our methods we first estimate the parameters of the tested distributions and only then apply the tests, as detailed next.

The first phase detection mechanism that was implemented uses the KS test to identify and characterise stable/transient phases. This test performs the following steps:

1. Order the sample points
2. Build the empirical distribution function:

$$F'_N(x) = \frac{\text{number of values in the history that are } \leq x}{N}$$

3. Assume some distribution and estimate its parameters
4. Compute the KS statistic:

$$D_N = \max_x |F'_N(x) - F(x)|$$

5. If  $D_N \leq d_{N;\alpha}$ , the KS test accepts that the sample points follow the assumed distribution  $F$ , with significance level  $\alpha$ , and a stable phase is detected. Otherwise, a transient phase is detected.

The value of  $d_{N;\alpha}$  is obtained in a published table of KS critical values. The significance level defines the probability that a stable phase is wrongly recognised as a transient one. The algorithm described above is executed for each one of the distributions that we are considering in the current implementation: Exponential, shifted exponential, Pareto and Weibull distributions. If the mechanism does not distinguish a stable phase with any of these distributions, it assumes that the environment is changing, i.e., a transient phase is characterized.

The main advantage of the KS test is that the KS statistic critical values are independent of any specific distribution. That is the reason for having a single table of critical values, which is valid for all distributions. However, the test has some limitations. It tends to be more sensitive near the centre of the distribution than at the tails, and if the distribution parameters must be estimated from the data to be tested (which is what we do in our current implementation), the results can be compromised.

The second phase detection mechanism implements the AD test. Since both tests are distance tests based on CDFs, their algorithms are very similar. The steps of AD test execution are:

1. Order the sample points
2. Assume some distribution and estimate its parameters
3. Compute the  $A^2$  statistic (where  $N$  is the history size,  $Y_i$  are the ordered data, and  $F$  is the CDF of the assumed distribution):

$$A^2 = -N - S,$$

$$S = \sum_{i=1}^N \frac{(2i-1)}{N} [\ln F(Y_i) + \ln(1 - F(Y_{N+1-i}))]$$

4. If  $A^2 \leq a_{N;\alpha}$ , the sample points follow the assumed distribution  $F$ , with significance level  $\alpha$ , and a stable phase is detected. Otherwise, a transient phase is detected.

The value of  $a_{N;\alpha}$  is obtained in AD critical values tables. Analogously to the KS test, the AD test is applied for the exponential, shifted exponential, Pareto and Weibull distributions, trying to identify the distribution which represents the current conditions of the environment.

The most serious limitation of the KS test is solved when using the AD test: Distribution parameters can be estimated from the sample points without any influence on the results. However, the AD test is only available for a few specific distributions, since the critical values depend on the assumed distributions and there are published tables for a limited number of distributions.

In our implementation we are applying the critical values for the Weibull distribution also for the exponential and shifted exponential distributions, since they are special cases of Weibull (with shape parameter equals to 1.0). In the case of Pareto distribution, [37] presents a table of Pareto critical values obtained through simulations for some few shape parameters (0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, and 4.0). Given that the estimated parameters are used to compute the  $A^2$  statistic (specifically to calculate  $F(Y)$ ), the estimation must be as close as possible to the real value. If the sample points follow a Pareto distribution with shape parameter larger than 4.0, we cannot perform the AD test, since we do not have the necessary critical values. Thus, in these cases we assume a transient phase. Otherwise, we round the estimated parameter for the closest defined value (to select a more adequate critical value) and continue the test execution.

Both mechanisms need to estimate the distributions parameters in order to execute the statistical tests. There are various methods, both numerical and graphical, for estimating the parameters of a probability distribution. From a statistical point of view, the method of maximum likelihood estimation (MLE) is considered to be one of the most robust techniques for parameter estimation. The principle of this method is to select as an estimate of  $\theta$  the value for which the observed sample is most "likely" to occur [43], and we are applying it to estimate exponential and Pareto parameters. For Weibull and shifted exponential distributions, the MLE method produces equations that are impossible to solve in closed form. This means that they have to be simultaneously solved using iterative algorithms, which are time consuming. Because

execution time is a very important factor, we decided to estimate Weibull and shifted exponential parameters through linear regression analysis.

For a given input sample, it is possible that a phase detection mechanism identifies more than one distribution. This is due to similarities between distributions, uncertainty of the statistical method, and uncertainty of parameters estimation. However, each mechanism must return only one distribution when a stable phase is detected. In our implementation there are two options available to the users of the service: The mechanism should return the distribution that presents the lowest time bound, which is a more aggressive approach; or the mechanism should be more conservative, choosing the distribution with highest time bound (see Section 4.3.2).

### 4.3.2 Bound estimators

Depending on the output of the phase detection mechanisms, which consists of both stability indicators and estimated parameters that characterise a certain distribution, one of the bounds calculated by the implemented bound estimators will be selected as the output of the framework.

The current implementation has five bound estimators: The conservative estimator, which is always required, and estimators for the exponential, shifted exponential, Pareto and Weibull distributions.

The conservative estimator is based on the statistical one-sided inequality, which provides a pessimistic bound, but which holds for all probabilistic distributions. The other estimators are derived from the CDF (Cumulative Distribution Function).

The CDF is a function which describes the probability distribution of a real random variable  $X$ . For every real number  $x$ , the CDF of  $X$  represents the probability that the random variable  $X$  takes on a value less than or equal to  $x$ :

$$F_X(x) = P(X \leq x)$$

Since coverage is defined as the probability that a given bound is secure, i.e., that the real delay will be less than or equal to the assumed bound, in the CDF function:

- $X$  is the observed delay;
- $x$  is the assumed bound  $t$ , provided to the QoS Coverage Manager;
- $F_X(t) = C$ , the expected coverage.

For example, the exponential CDF is:  $F_X(t) = 1 - e^{-\lambda t} = C$

For a given coverage  $C$ , the QoS Coverage Manager can derive a secure bound by isolating  $t$ :

$$t = \frac{1}{\lambda} \ln \frac{1}{1-C}$$

The same methodology was applied to define bound estimators for the other three distributions. The five estimators are presented in Table 1. These estimators correspond to the several distributions that were considered, and that are used for evaluating the QoS Coverage Manager service. The details regarding these distributions will be provided in deliverable D4.2.2, as part of the evaluation work.

Estimator	Minimum time bound $t$
Conservative	$t = E(D) + \sqrt{\frac{V(D)}{1-C}} - V(D)$
Exponential	$t = \frac{1}{\lambda} \ln \frac{1}{1-C}$
Shifted Exponential	$t = \gamma + \frac{1}{\lambda} \ln \frac{1}{1-C}$
Pareto	$t = \frac{k}{\alpha \sqrt{1-C}}$
Weibull	$t = \alpha \sqrt{\ln \frac{1}{1-C}}$

**Table 1: Bound Estimators for a required coverage  $C$ .**

### 4.3.3 Selection Logic

The phase detection mechanisms are individually executed in parallel, and each one indicates the environment condition (transient or stable period) and returns one, and only one distribution which characterises the history trace whenever a stable phase is detected. Depending on the output of the phase detection mechanisms, one of the bounds calculated by the implemented bound estimators will be selected as the output of the QoS Coverage Manager. The selection is performed with the help of some logic, which might be made more or less aggressive.

For now, we are considering three options: (i) A stable phase is characterised only if all mechanisms identify the same distribution (with the same parameters), for instance, both identify a Pareto distribution; (ii) accept a stable phase when different distributions are detected and choose the distribution which produces the lowest bound; or (iii) accept a stable phase when different distributions are detected and choose the distribution which returns the highest bound.

## 4.4 Results, Analysis and Discussion

We evaluated the QoS Coverage Manager focusing on two main points: Comparing the ability of both phase detections mechanisms to correctly characterise the environment; and validating the entire service using real RTT (round-trip time) measurements collected in real environments. The full description of the experimental setting and of the obtained numerical results will be reported in WP4 (D4.2.2), while in this section we just summarise the main outcomes having an impact on the design and implementation of the service.

In the first part of our evaluation, we tested the QoS Coverage Manager using synthetic data traces in order to analyze the correctness of AD and KS phase detection mechanisms. For each one of the four considered distributions we synthetically generated 10 different traces of 3000 samples. We executed the QoS Coverage Manager for each trace using both mechanisms individually. For these executions we defined history size  $h=30$  and desired coverage  $C=98\%$ . The significance level for both mechanisms was  $\alpha=0.05$ . We defined that when more than one distribution is detected by a phase detection mechanism, it chooses among them the distribution which returns the highest time bound.

We specified four metrics to compare AD and KS goodness-of-fit tests: The first two (*stable phases* and *correct distribution*) demonstrate the accuracy of the mechanisms on detection and characterisation of stable phases; the third metric (*improvement of bounds*) compares the adaptive and pessimistic time bounds in terms of the improvement obtained with the adaptive approach; and the fourth metric (*coverage*) quantifies the achieved coverage. From the analysis of the preliminary results it was apparent that **there is not a best**

**goodness-of-fit test for all distributions.** Actually, **the performance of each mechanism depends on the chosen evaluation criteria, on its intrinsic features and on the distribution characteristics.**

In the second part of our evaluation we collected data from different projects on the Internet (Immotion [19][20], Umass [44], CNU [28][29], Dartmouth [26][23]) in order to (1) show that our assumption about the interleaved probabilistic behaviour is realistic for network delays, and (2) demonstrate that the QoS Coverage Manager is able to characterise these behaviours and provide applications with improved, but still dependable, time bounds. We downloaded files from these projects and extracted RTT traces using *tcprace* tool. These RTT traces were used as input samples for our service.

We executed the service using the three possible configurations for the selection logic: A stable phase is characterised only if all mechanisms identify the same distribution, the service chooses the distribution which produces the lowest bound, or the service chooses the distribution which produces the highest bound.

As a result, **all configurations achieved the desired coverage.** Nevertheless, **the best relationship among aggressiveness and dependability was obtained when choosing the lowest bound.** As should be expected, **the other two configurations produced most conservative results, but all resulted in some improvement in comparison with the pessimistic approach.**

These results prove that there are real environments in which our assumptions hold. In these environments, the QoS Coverage Manager can be successfully applied in order to make estimations through analysis of historical data and provide applications with better information related to the environment behaviour (e.g. network delays), while ensuring the same level of dependability.

## 4.5 Summary

This section presented the QoS Coverage Manager, which is intended to provide a support service for applications that may adapt their behaviour depending on the available QoS (typically of the communication channels). The service uses monitoring information to characterise the probabilistic state of the environment and, based on this characterisation, determine the exact operational parameters (e.g. timeouts) that should be used in order to ensure that the selected values will be secured with some desired probability (given the observed conditions).

The design and implementation of this service implied a comprehensive study of probabilistic approaches to handle the assumed probabilistic behaviour of the environment. The possibility of adapting in a dependable way is a very difficult problem when considering uncertain environments. First, the definition of “dependable” in this context is in itself a problem, which we have solved by defining a theoretical framework that allow us to consider that the objective of adaptation is not to select a guaranteed bound, but to select a guaranteed <bound, coverage> pair. Then, even to provide this kind of guarantees, it was necessary to establish bounds on the dynamics of the environment. We did that by establishing a model for the assumed behaviour, in which the environment alternates between stable and unstable periods, and we assumed that this alternation cannot be arbitrarily fast.

With the above mentioned assumptions, we were then able to study several probabilistic methods that we implemented within a QoS Coverage framework. The methods were refined and improved, namely in terms of the means to estimate the parameters for the assumed distributions, and several probabilistic distributions were added to the framework (corresponding to different methods for estimating their probabilistic parameters) during the development and implementation work.

With this framework we were then able to perform a number of evaluation and quantification experiments, which have only been briefly presented in this deliverable. The results were interesting and promising, showing that it is indeed possible to use the QoS Coverage Manager service to help applications determine the right bounds that must be used given the available QoS.

The QoS Coverage Manager service was implemented both as a separate service (that was used for the evaluation) and as a service within the Platooning Test Bed. In this case, the objective of using the QoS Coverage Manager is to monitor transmission delays for the communication between the several platoon members, allowing to select the bounds that should be used within the control algorithms, and corresponding periods of transmission, in a way that control decisions use, as best as possible, recent information regarding

platoon members' positions. The QoS Coverage Monitoring is not in the safety-critical path of the Platooning application. However, it can be used to improve the overall perception of quality of the application (from the perspective of a driver or an external observer), by optimizing the trade-off between global speed of the platoon and the smoothness of its behaviour. The final presentation of the results achieved with the proof-of-concept test beds will be presented in deliverable D6.4.

## 5 Replication Manager

The Replication Manager is used to help provide dependable services in the ad-hoc domain. It replicates the state of the application on other nodes in the ad-hoc network. The replication of the application is done in small groups. The Replication Manager (RM) locates peer nodes in the network using standard service discovery libraries. If a peer is found and it is suitable for inclusion in the replica group it will be swapped into the group.

### 5.1 Service Description

Many of the future networking scenarios consist both of wireless multi-hop parts and infrastructure based network components. For new application types and future service platforms, server-based application access is not only offered by the infrastructure network part, but also by the potentially mobile nodes in the ad-hoc domain. An example of such service provisioning scenarios is the vehicular communication setting [40], in which cars will be able to communicate with each other. This communication will be used for safety critical applications that require high availability. For applications used in automotive traffic this is especially true since application failure could affect driver behavior or directly affect the state of the car, e.g., spurious application of brakes. Traditional solutions for high-availability rely on redundancy offered by cluster implementations, in which Middleware services [40] support the timely replication and fail-over in case of crash failures of individual cluster nodes [10]. For statefull applications, such fail-over capability typically involves timely replication of application state, which could be implemented by a redundant distributed shared memory [4].

In mobile ad-hoc networks (MANETs) the lifetime of a communication path may be short [1]. Communication delays are in principle unbounded due to the unpredictability of the Medium Access procedures on the link-layer and potential re-routing delays in dynamic multi-hop scenarios [11]. Replication strategies for dynamic data such as application state need to take these communication properties into account [31]. Larger communication delays can increase the probability of inconsistent replica state, so dynamic cluster member selection can lead to substantial improvements in mobile scenarios [34]. The Replication Manager defined in [22] is an example of such a middleware component that provides applications with a resilient shared memory area and performs management of such a dynamic cluster. Heuristic algorithms as investigated in [31], based on measured communication delays and geographic positioning and speed information, can be utilized to trigger membership reconfiguration in a mobile ad-hoc network setting.

The Replication Manager replicates the state of the application onto other members of the same replication group. It is assumed that the same applications are running in the HIDENETS ad-hoc nodes or at least that the nodes participating in the replication groups have the same applications running. The application users can find and access the application using a combination of the zeroconf protocol and a local instance of the Replication Manager used to gain access to the application server.

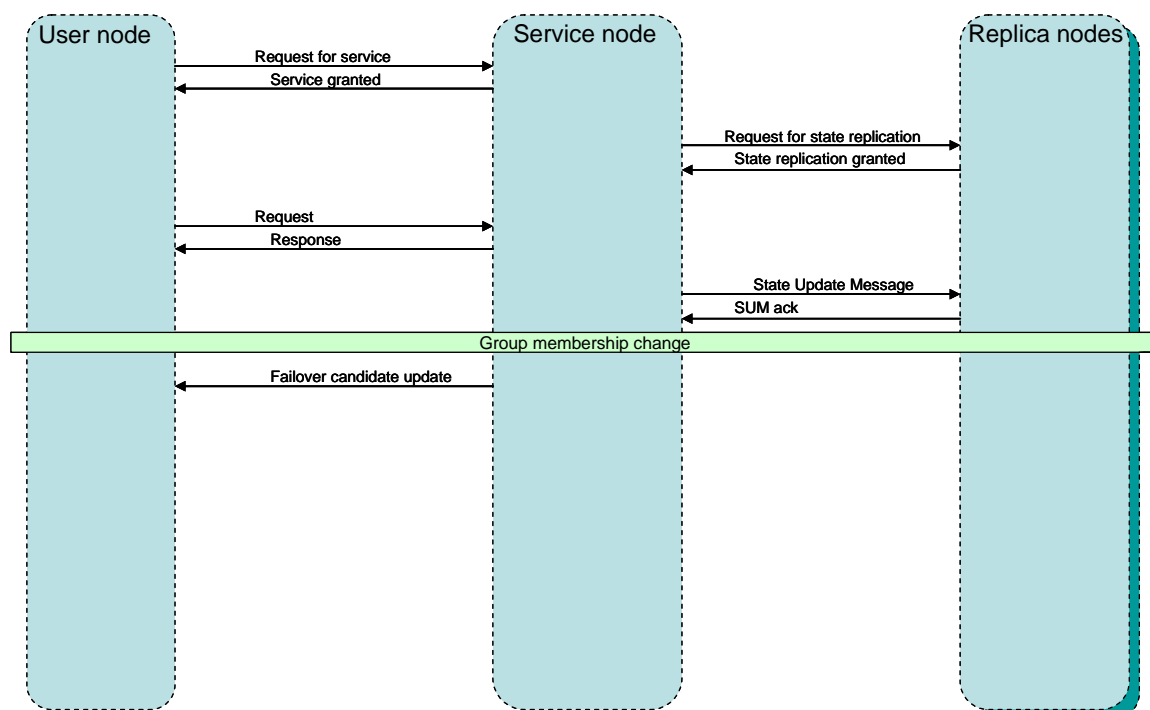
Using numerical results from analytic Stochastic Petri Net models of this dynamic replication scenario, we experiment with different geographic mobility types, and different degrees of dynamicity of the application state[6]. These results are used to make design decisions and prove the viability of the Replication Manager.

## 5.2 Design, Specification, Interface

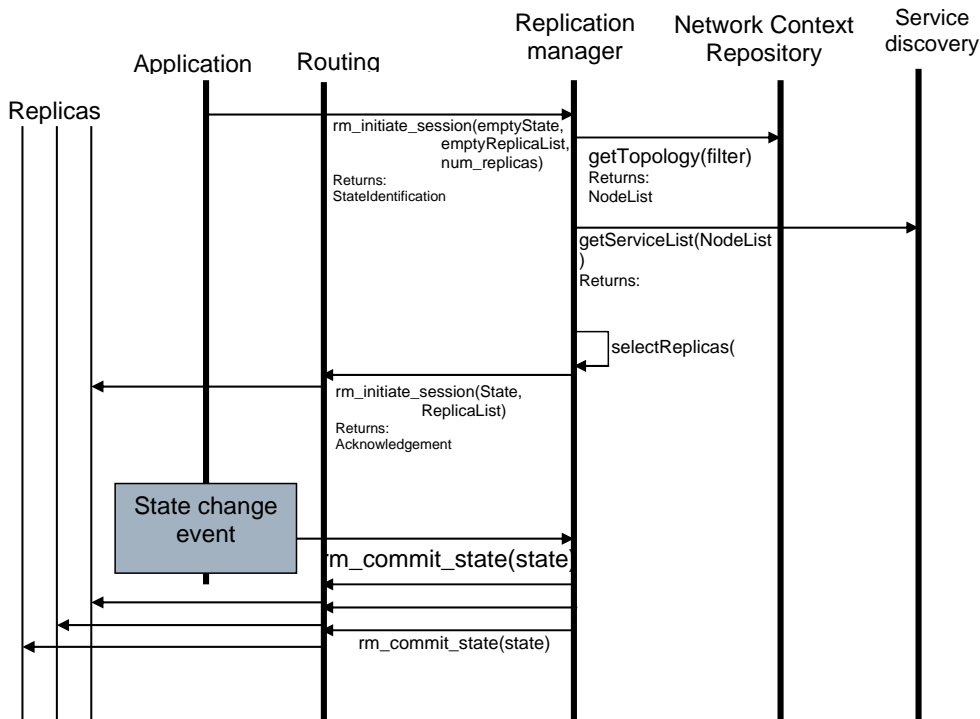
The application programming interface to the Replication Manager is defined in Table 2. The functions provided by the Replication Manager to the application that needs to be replicated are used as follows: `initiate_session` is used by the application to setup an application context and request that the application context is replicated to the desired number of servers. The counterpart is `close_session`, which closes the session with the Replication Manager. `Commit_state` is used by the application to request that update messages are sent to the replica group. `Read_state` is used by the user node as a part of the library. It is both used by user nodes and service nodes. The interplay between nodes in a scenario where the Replication Manager is used is described in Figure 10. In a node the different parts of the HIDENETS middleware take part in the complex task of providing dependable communication resources on top of an unreliable ad-hoc network. The interaction between the Replication Manager and the other middleware services is described in Figure 11.

Return type	Function name
int	<code>rm_initiate_session (int n)</code>
void	<code>rm_close_session (int session_id)</code>
void	<code>rm_session_keepalive (int session_id)</code>
void	<code>rm_install_callback (int session_id, callback_t notifier)</code>
void	<code>rm_install_session_list_callback (session_list_callback_t notifier)</code>
void	<code>rm_commit_state (int session_id, char *state)</code>
char *	<code>rm_read_state (int session_id)</code>

**Table 2: Replication Manager API.**



**Figure 10: Interaction between participating nodes in RM use-case.**



**Figure 11: Interaction between parts of the node architecture, the bold lines represent processes within one node.**

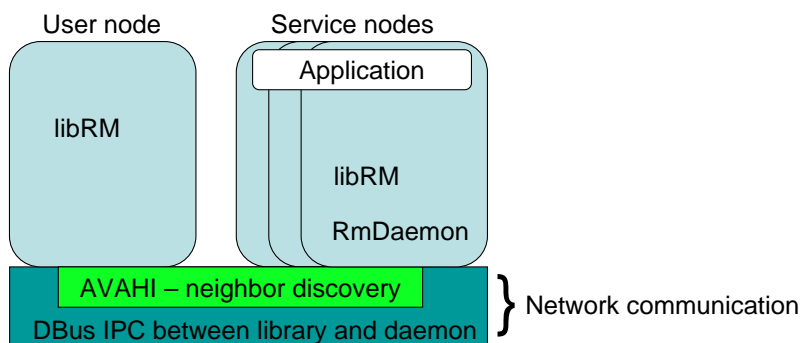
### 5.3 Implementation

The Replication Manager implementation consists of two different software entities:

- **librm** – A library which provides the Replication Manager API for applications, and passes on the API calls to the `rmdaemon`. The library is also able to locate failover candidates for the user nodes.
- **rmdaemon** – Performs the actual work of the Replication Manager, including managing sessions, facilitating state updates, and group discovery.

The Dbus system bus is used for the IPC between `librm` and `rmdaemon`, and the AVAHI service discovery system is used by the `rmdaemon` to implement the interplay between nodes.

The Replication Manager is separated into two entities to allow for a single daemon per node, which manages all sessions for all application on that node, using a single hook to the AVAHI service discovery system, yet allowing multiple applications utilising multiple sessions to use a simple API, implemented by the library.



**Figure 12: Interaction between parts of the replication manager.**

### 5.3.1 The library (librm)

The library is dynamically linked to applications which uses the Replication Manager and provides the API described in Section 5.2 for the applications. The responsibility of the library includes keeping track of initiated sessions and registered callbacks, mediating API calls to the daemon using the Dbus interface, receiving messages from the daemon, and invoking registered callbacks based on these messages.

Thus a single instance of the library is running for each instance of an application using the Replication Manager, which typically runs as a normal user, without special privileges.

Upon initialisation of the library a connection is established to the service provided by the daemon on the Dbus system bus. This connection is maintained throughout the lifetime of the application, and used for all communication to and from the daemon.

### 5.3.2 The daemon (rmdaemon)

Exactly one daemon is running on every node running the Replication Manager. Upon initialisation the daemon connects to the Dbus session bus and registers a service under the name *org.aau.rm*. The daemon maintains several connections to the Dbus established by different instances of the library, and it manages zero or more initiated sessions per connection.

The daemon is typically launched as a system daemon, either with escalated privileges or with normal user privileges. If executed with escalated privileges it creates a system wide lock file and drops privileges to a normal user (typically the *nobody* system daemon user,) as common security practices advocates.

## 5.4 Results, Analysis and Discussion

In a car-to-car ad-hoc network the topology may change so frequently that the amount of signalling needed to keep a group of replicas consistent is potentially large [25]. The smaller the number of active replicas, the higher the unavailability of the application service due to crash failures of the server (which is undistinguishable from the server leaving the connected network). On the other hand service replication by broadcasting state information to all nodes in the network will increase wireless bandwidth consumption and may lead to congestion. The number of replicas to select is hence a trade-off between overhead and availability. The design goal of the RM is to provide services to the user with a high perceived quality of service, while keeping replication and reconfiguration overhead as low as possible. Therefore we have chosen to favour the tradeoffs that give the best perceived service to the user node.

The metrics with the highest impact on user perceived quality of service are service response time, service availability, and the correctness of the service [2]. Correctness is influenced by the consistency of the replicated data. If a user is getting a service which is provided based on out-of-date data the service is not correct. Incorrect service influences the user's perception of the service dependability. The faults that affect the user perceived quality of service are closely related to the metrics already described. For example, packet loss or excess delay affects the timeliness of the service and also the correctness. A lost update packet is extending the time where the replica server is in an inconsistent state until a successful retransmission or a new update message is received. The same is true for large delay. If a single server in the group is experiencing these types of faults, it can be excluded from the group and a new service node – if any eligible ones are reachable in the network - can be included. The replicas are selected in order to achieve stable clusters as this is preferred to minimize reconfiguration overhead; see [31] for strategies to increase stability.

In summary, the most important problems addressed by the RM are:

- Selection of replicas with stability criteria and best communication metrics.
- Reduction of reconfiguration overhead.
- Reduction of service response time.

Any node in the replica group can propose new group members. When a member is proposed all existing members must acknowledge the adoption of the new members. If the majority of the existing members accept the new group member by sending a positive acknowledgement to the proposing node, the latter will send an updated membership list to all nodes in the group and to all user nodes using the service provided by

the group. The user nodes need to get member updates in order to have an up to date list of failover candidates in case they loose their connection with the server they are currently using. The state manager part of the Replication Manager is responsible for sending update messages in case of a state change caused by a write operation to the shared memory region, to store received state variables, and to send an acknowledgement when a state update message has been received. In this way the write operation will be replicated to the other group members. The retrieval part is responsible for selecting a server to be used by the user nodes in case they loose the connection with the server they are currently using. The retrieval part of the RM is storing the updated list of group members and selects the failover server based on communication metrics and geographic properties of the servers. Furthermore the retrieval part of the RM is responsible for retrieving the state of the shared memory area when a new service node joins the replica group.

Naturally there are constraints on the rates used in the model if the resulting numbers should show results for realistic scenarios. For instance the update rate is used to represent the end-to-end delay of the network between the nodes in the group. There is a theoretical minimum to this delay which is the time it takes to transfer one update message to a link-layer neighbor. Assuming that one update message can be sent within 50ms under good conditions (one hop, no MAC delay) the corresponding update rate will be around  $\delta=1200$  updates per minute. To analyze the impact of this parameter on the results, two different values of  $\delta$  are considered: 1000 and 100 updates per minute. The order of magnitude of the data change rate  $\gamma$  depends on the considered application. Two different values of this parameter have been considered for the results presented in this section: 10 and 100 updates per minute, corresponding to an average time between two consecutive updates of 6 sec. and 600 msec. respectively.

As regards the rate  $\alpha$  of meeting new cars that are able to join the group, we have also considered different values corresponding to different traffic situations. The minimum rate is zero meaning not meeting any cars at all on the road. An example value for  $\alpha$  is 30 cars/sec, which is approximately equivalent to cars, with the length of 5 meters, driving in opposite directions with a speed of 200 km/h with little or no space between the cars. This value could be much higher when considering multi-hop communication. Various values for the leaving rate  $\beta$  can also be considered to reflect different behaviors of the participating nodes. This value is also difficult to estimate because it depends on several influencing factors.

Rather than focusing on the absolute values of  $\alpha$  and  $\beta$ , one can analyze the results by considering the relative ratio  $\alpha/\beta$ . Higher values of this ratio correspond to environments where the probability of meeting a new car is higher than the probability that a participating node will leave the network. In a freeway scenario where cars join and leave the road via on and off-ramps and cars travel with a mean speed of 130 km/h excluding trucks the rate of cars joining the group is quite low because the cars travel with approximately the same speed. Assuming three cars joining the group per minute gives a join rate of 3 cars/minute. With an assumption that each car stays in the group on average for 3 minutes the actual leave rate is  $\beta=0.33$  cars/minute. Accordingly the ratio  $\alpha/\beta$  will be approximately 10. Much higher values can be obtained for other scenarios.

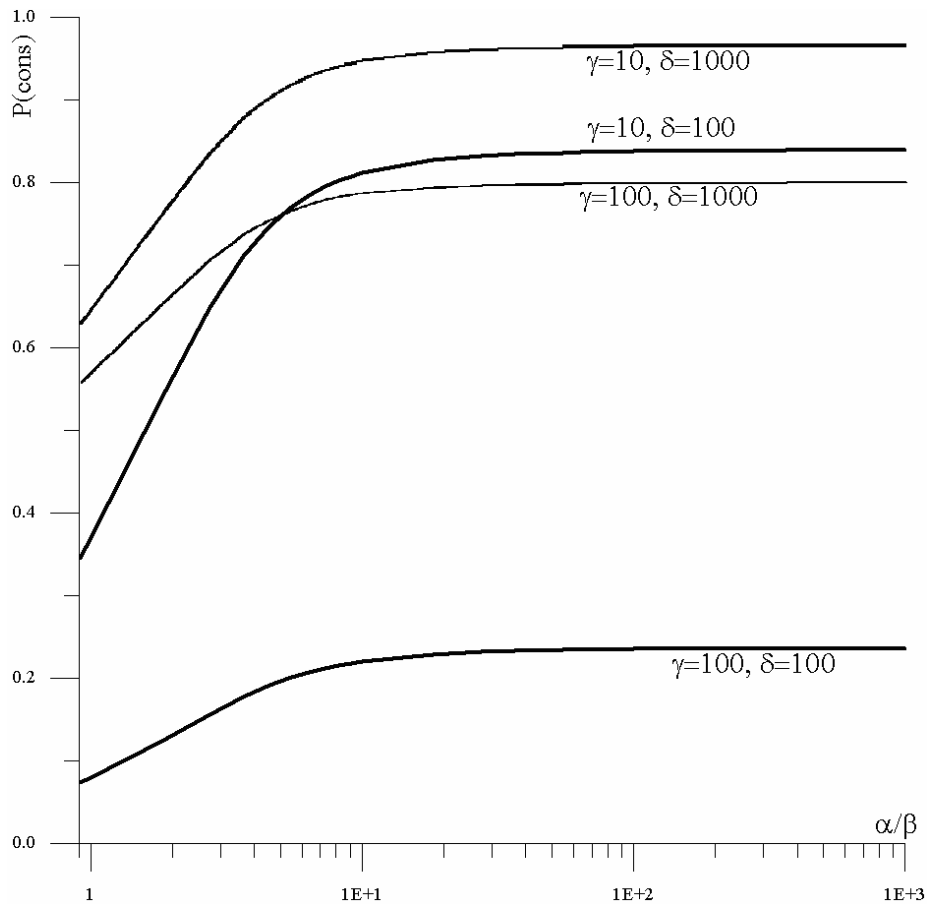
Table 3 summarises our settings of the parameters  $\alpha$ ,  $\beta$ ,  $\delta$ , and  $\gamma$  that represent the range of values expected to be found using measurements from vehicular networks. It is worth noting that the ranges presented in Table 3 represent approximations of the parameter ranges. .

Parameter	$\alpha$	$\beta$	$\gamma$	$\delta$
<i>Min</i>	0	0	10	100
<i>Max</i>	10000	10000	100	1000

**Table 3: The maximum and minimum values for each parameter.**

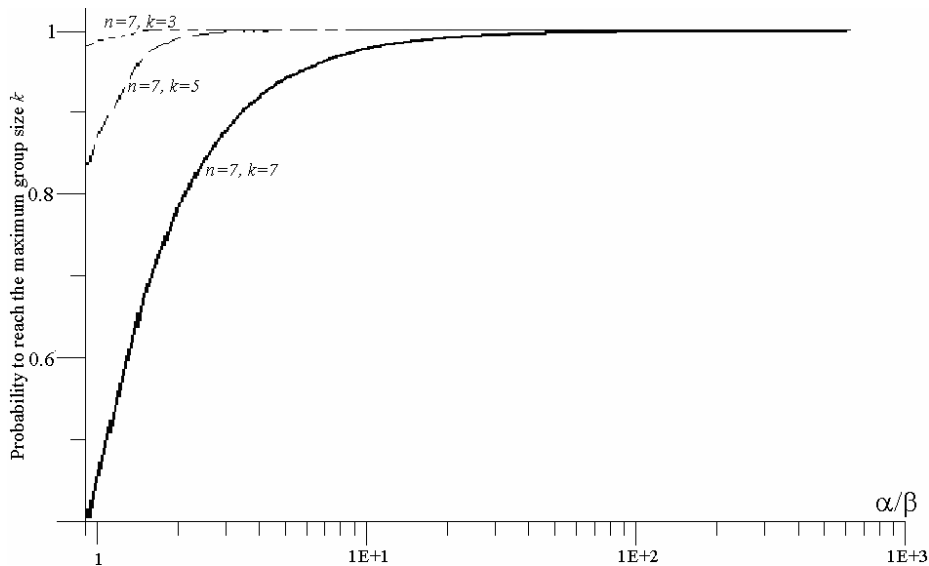
As regards the replica consistency, Figure 13 shows that the probability of a consistent replica set converges rather fast to a limit value for increasing  $\alpha/\beta$ . This limit value depends strongly on the relation of the application state change,  $\gamma$ , and the network delay rate,  $\delta$ . This is intuitive too, since timely updates are needed to achieve consistent replica groups.

Figure 13 shows the combined impact of parameters  $\gamma$  and  $\delta$  on the consistency probabilities. It can be seen that a decrease of  $\delta$  of one order of magnitude (from 1000 to 100), could lead to a degradation of the consistency probability in the order of 4 to 9 times, depending on the considered values for the ratio  $\alpha/\beta$  and the value of  $\gamma$ . Also, for  $\alpha/\beta$  values higher than 10, the higher the ratio  $\delta/\gamma$ , the higher the consistency probability.



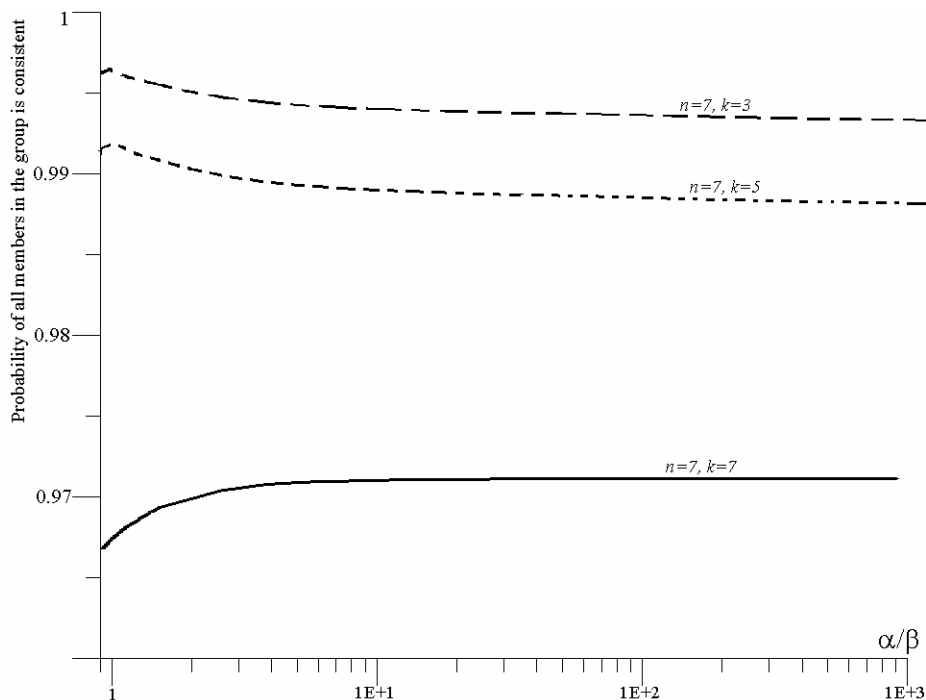
**Figure 13: Probability of having a fully consistent group for  $n=k=5$ .**

Considering the behaviour of the consistency probability as a function of the replica group size, Figure 14 shows that for traffic situations corresponding to small values of the ratio  $\alpha/\beta$  (e.g., in highway systems) the probability of achieving a full replica group is reduced significantly when the desired group size grows. For the desired group size of  $k=7$  in a network with  $n=7$  nodes the ratio between  $\alpha$  and  $\beta$  must be larger than 3 to get a probability of more than 50% of achieving a full replica group. The above estimates of input values show that it is unlikely to fill up groups larger than  $n=5$  nodes.



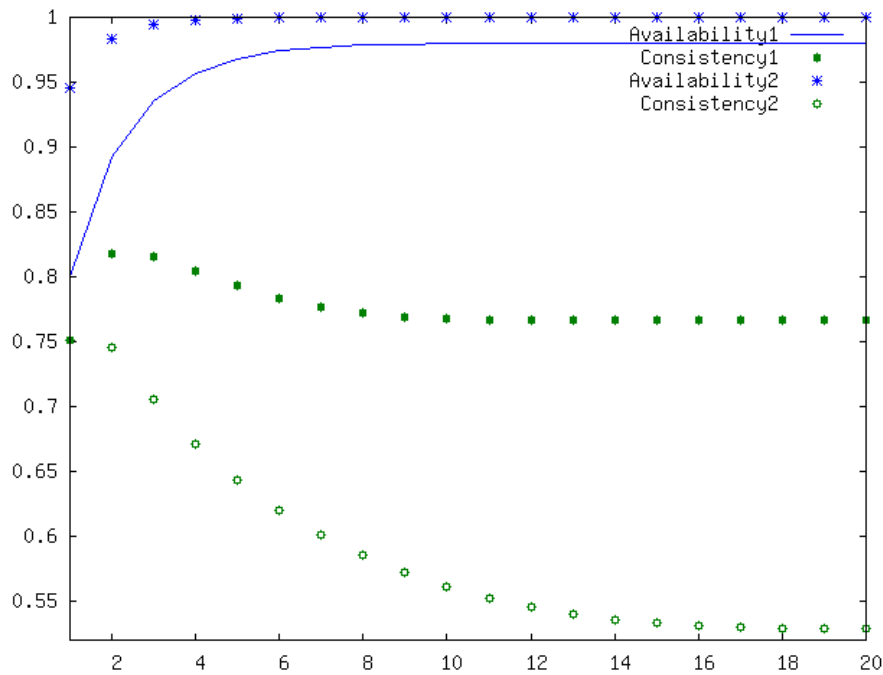
**Figure 14: Probability of reaching different maximum group size  $k$  with  $n=7$ .**

With respect to the probability of all replicas being consistent the probability is high even with the ratio between  $\gamma$  and  $\delta$  being as low as 10, the probability of having consistent replicas is about 80%. Citing the limits given above the ratio between  $\gamma$  and  $\delta$  is 10 both when considering the extreme high rates and the extreme low rates. Overall the probability of achieving consistent replica group members is high when the network is fast and not congested. Furthermore the data change rate  $\gamma$  must be considerably lower than the update transmission rate. In Figure 15,  $\delta$  is 100 times larger than  $\gamma$ .



**Figure 15: Probability that all members in the group are consistent.**

Figure 10 shows the service availability for two different settings of  $\alpha$ ,  $\beta$ ,  $\gamma$ ,  $\delta$ . In the first setting, the parameters are set as follows:  $\alpha=5$ ,  $\beta=1$ ,  $\delta=100$ ,  $\gamma=10$ . In the second setting, the parameters are set as follows:  $\alpha=10$ ,  $\beta=1$ ,  $\delta=100$ ,  $\gamma=20$ . From the figure it shows that there is a trade-off between service availability and service correctness (here measured in data consistency). In Figure 16  $n=k$  and it goes from 1 to 20 along the x-axis. It seems that the best group size for maximum availability and correctness is 3 to 4 servers.



**Figure 16: Service availability vs. group consistency for different group sizes.**

## 5.5 Summary

Experimenting with the implementation of the Replication Manager and modelling the Replication Manager has taught us about some important aspects of the configuration of the Replication Manager. We have shown that service provisioning in dynamic ad-hoc networks is possible to a certain degree if the application dynamics are well suited for ad-hoc networks. As the dependability increases with higher number of service nodes in the group there is a trade-off to be found because the data consistency decreases with higher number of nodes in the group. Partially due to the additional overhead related to managing the group of service nodes. The selection of the nodes is done in a best effort way where nodes are selected first come first serve for inclusion in the group. At a later stage more complex selection criteria will be investigated in the implementation of the Replication Manager as in [38].

## 6 Proximity Map

### 6.1 Service Description

In distributed systems such that the one targeted by HIDENETS, the notion of the position of nodes is as important as the logical properties of the network. Moreover, the geographic locations and movements of the entities are tightly coupled to the quality of service attainable by the communication system in the underlying network. As an example, two vehicles driving in the same direction and sufficiently close to each other may open a connection and exchange data.

The goal of the Proximity Map service is to provide an abstraction of the map of neighbouring nodes (along with estimated position, speed and direction), by computing a hybrid representation of the local environment. In this context, hybrid means that the map will present application and services an abstraction that mix physical (i.e. geographical) and computational (i.e. communication related) information. Thus, the Proximity Map represents the local knowledge a node can compute about its vicinity. All applications and services that require to link the physical world with the network can benefit from the Proximity Map service.

This map is parameterized by its wideness (the number of communication hops represented on the map, which is related to the maximal physical distance of a node on the map) and its accuracy, i.e. how often the map is updated.

### 6.2 Design, Specification, Interface

The Proximity Map service provides a position and an evaluated accuracy for each neighbour node, according to given parameters specified by the calling service, which include:

- Maximum physical distance to the node,
- Maximum logical distance to the node, i.e. maximal number of communication hops,
- Maximum age of information.

#### 6.2.1 Proactive vs. Reactive design

Since this service is closely related to neighbour discovery protocols, at least for the networking part, we investigated the various approaches proposed for solving this problem. In ad-hoc networks, proposed neighbour discovery protocols that are used to provide routing schemes can be divided into *proactive schemes* and *reactive schemes*. In a proactive scheme, the entity periodically sends messages on the network to look for new neighbours, and to check the availability and reachability of already discovered routes. In a reactive scheme, the routing map evolves only when requested by the application/caller.

Since it is assumed that

- calls to the Proximity Map will be issued at any point in time, and
- the physical dissemination of nodes is evolving rapidly

the Proximity Map services acts *proactively* in order to serve requests without further communication.

#### 6.2.2 Short Specification

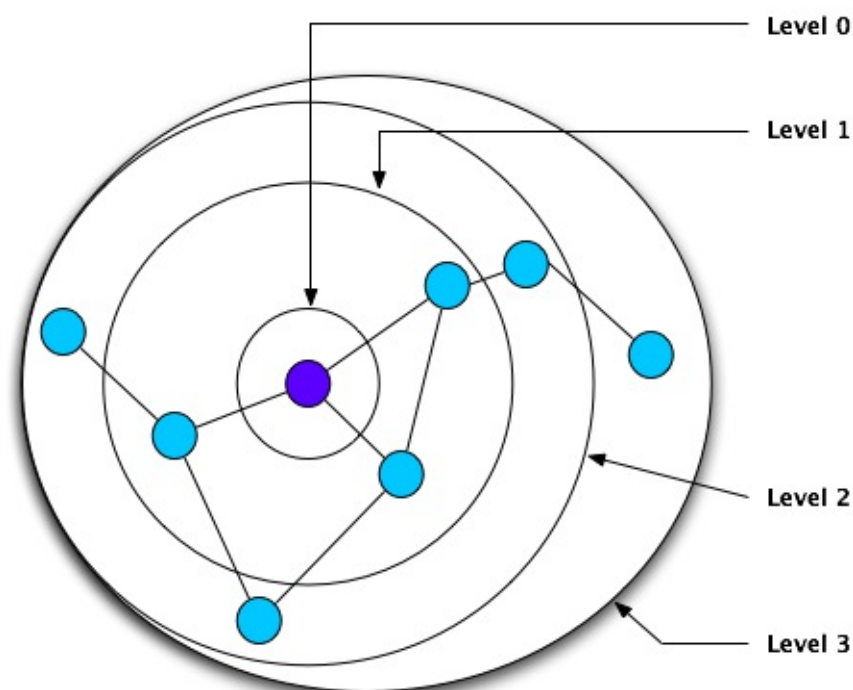
The goal of the Proximity Map is to gather physical information about nodes in the vicinity. When using its Proximity Map, a given node has a view of the nodes in its vicinity (defined as being the nodes which are reachable within  $H$  hops), their location information, and indications on the freshness of information.

## 6.3 Implementation

### 6.3.1 Implementation strategy

Roughly speaking, each node periodically beacons<sup>1</sup> its Proximity Map to its 1-hop neighbours, and collects similar information from its direct neighbours. When merging these pieces of information, it is able to update its Proximity Map with up-to-date information on already discovered nodes, new nodes that appeared as neighbours of neighbours, nodes whose connectivity changed, etc.

To implement the Proximity Map, we use location-stamped beacons. Each node keeps a map of its knowledge of the location and connectivity of other nodes, which is represented as a graph as shown in Figure 17. This graph is regularly updated when the node receives a beacon and is also regularly sent to the node's neighbours in its beacons.



**Figure 17: Link between physical and logical representation.**

### 6.3.2 Link between nodes knowledge and representation

The problem of representing locally-computable knowledge is indeed tightly related to the problem of information dissemination. Thus, the Proximity Map implementation is based on the notion of *Knowledge Map* that represents knowledge dissemination in the system.

The basic piece of information is abstracted by the notion of *Located Node*, that contains a Node Id, a speed, a direction, a location in a given coordinate system ( $(x,y,z)$  or  $(latitude, longitude, altitude)$ ) and a timestamp. It represents typical information available in positioning devices such as GPS receivers.

Periodical messages containing the knowledge map are sent by nodes that receive other nodes' knowledge map, merging messages information with local information in a tree that enjoys the following properties:

<sup>1</sup> Beacons, in our context, are short and periodic messages that can be piggybacked on top of other messages when necessary/possible.

- each node in the tree (be it a leaf or a node with children) contains positioning information for a particular vehicle, *i.e.* nodes in the knowledge map are Located Nodes as defined below, and contain position, speed, direction, timestamp of this piece of information.
- the root of the tree is the knowledge a vehicle has on itself. It is thus the most accurate and precise piece of information available to the vehicle.
- If a node representing vehicle  $A$  has children  $C1, C2, \dots, CN$  it means that the information on  $C1, \dots, CN$  has been forwarded by  $A$ , *i.e.* that  $C1, \dots, CN$  were one-hop neighbours of  $A$  at the date indicated by the timestamp.

As explained above, this tree has a maximum height of  $H$ , and is periodically sent to direct neighbours. When a vehicle updates its tree with a tree  $T$  sent by another vehicle  $A$ , it performs the following actions:

1. Merging information, by updating the subtree rooted in  $A$  by  $T$
2. Pruning the resulting tree, *i.e.* the resulting tree is chopped in such a way that its height is bounded by  $H$
3. Cleaning the resulting tree of useless redundant knowledge about the vehicle itself.

Notice the following properties of the tree:

- The branch from the root to a node  $A$  is a possible routing scheme from the current node to  $A$ .
- A node  $A$  may appear more than once in the tree. It means that  $A$  is attainable by more than one node at the last hop and, consequently, can be contacted via multiple paths.
- Links are supposed to be bi-directional.
- The freshest and the more trusted information are preferred.

### 6.3.3 Map accuracy:

As beacons are sent every  $D$  time units, level  $L$  information is  $(L*D)$  old. Because high-level knowledge is older and because it is neither desirable nor possible to have the knowledge of the whole network, the maximum level of knowledge is bounded. This bound,  $H$ , is determined statically according to the application, the density of the network and other environment parameters.

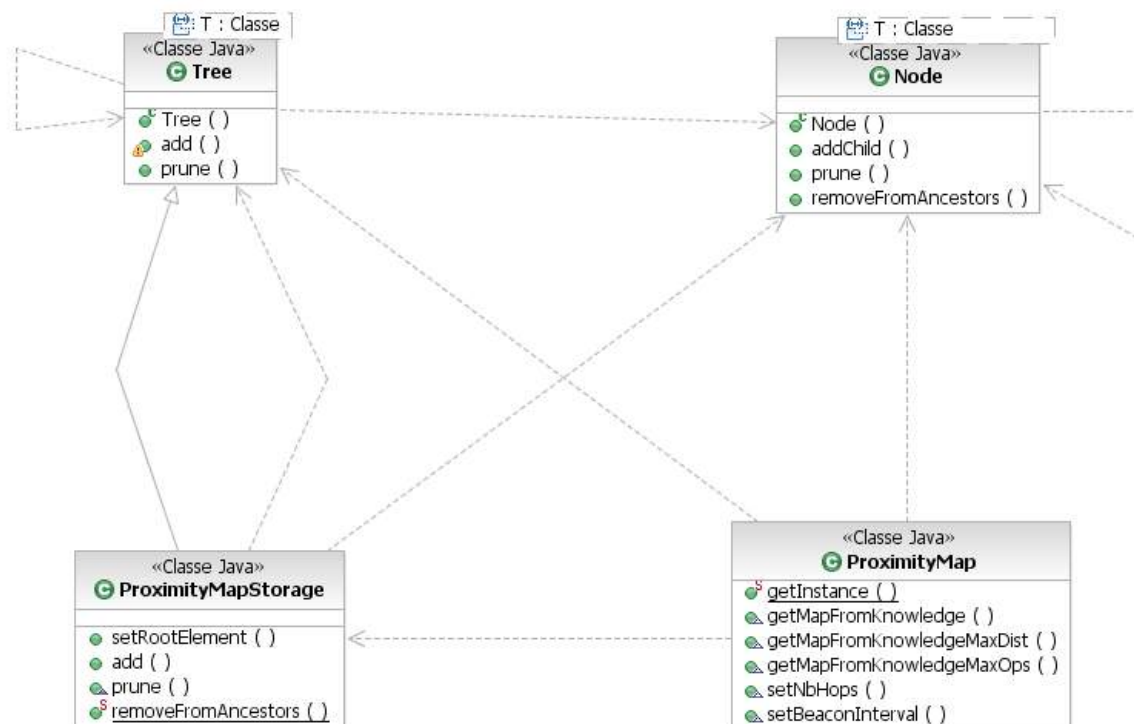
This algorithm is pro-active and enables a node to know the other nodes physically present within the area if  $H$  is sufficiently large. When  $H$  is not large enough, or the coverage obtained is not sufficient, a reactive protocol may be used to complete location information further than  $H$  hops.

At any time after  $H*D$  time units, a node knows the location and coverage of its  $H$  hops neighbours. It should however be noted that at time  $t$ , every node in the network has a different view of the connectivity since its level 1 information is  $D$  time units old, its level  $n$  is  $n*D$  time units old, etc.

### 6.3.4 Implementation in Java

The implementation is based on four classes, as shown in Figure 18:

- ProximityMap is the actual frontend of the module
- ProximityMapStorage implements the knowledge map mechanisms (tree sending, merging and backend for pruning)
- Tree is a classical Java implementation of  $n$ -ary trees, fitted for the knowledge map
- Node is the Java implementation of Tree nodes, adapted for our scenario. The actual pruning code is located in this class.



**Figure 18: ProximityMap class diagram.**

#### *ProximityMap class:*

This class is a singleton, i.e. only one instance of a Proximity Map can exist on a Node. The main methods exported by the class are:

- `setBeaconInterval(long)` sets the periodicity of broadcasts of the map. Notice that this does not affect the node's map precision, but affects its neighbours' maps' precision. We assume that for a given application, all nodes will use the same beaconing interval.
- `setNbHops(int)` impacts the maximum height of the knowledge map stored on the node. If  $H$  is the local maximum height of the map, and  $N$  is the bound on neighbours attainable in one communication hop, then the actual size of the knowledge map is bounded by  $N^H$ .
- `getMapFromKnowledge()` exports a collection of `LocatedNode` elements from the knowledge map, retaining for every node the freshest pieces of information.
- `getMapFromKnowledgeMaxHops(int maxHops)` exports a collection of `LocatedNode` elements from the knowledge map, retaining for every node the freshest pieces of information, and restraining it to nodes attainable in at most `maxHops` communication hops.
- `getMapFromKnowledgeMaxDist(long maxDist)` exports a collection of `LocatedNode` elements from the knowledge map, retaining for every node the freshest pieces of information, and restraining it to nodes at physical distance `maxDist` at most.
- `getMapFromKnowledgeMaxTime(long maxTime)` exports a collection of `LocatedNode` elements from the knowledge map, retaining only nodes for which information is at most `maxTime` milliseconds old.

#### *ProximityMapStorage class, Tree class and Node class:*

The `ProximityMapStorage` class is a serializable class that supports merging (by the `add` method), pruning to a specified height (using the method `prune(int)`), and cleaning of the `Tree/Nodes`, i.e. removing duplicate nodes created by the merging. This class uses extensively the `Node` and `Tree` classes.

## 6.4 Summary

The Proximity Map service provides an abstraction of the map of neighbouring HIDENETS nodes (along with estimated position, speed and direction). The Proximity Map represents the local knowledge a node can compute about its vicinity. It provides an abstraction that correlate both geographical and communication related information.

It is worth noticing that the implementation of the Proximity Map service is truly a “Best-Effort” one, since the information it provides is the most accurate possible, but no real quality of service can be guaranteed.

When several nodes run in the same direction at comparable speeds, the level of details of the Proximity Map service is dictated by the accuracy parameters (BeaconInterval and NbHops), provided all nodes use the same parameters, which is assumed in our test scenario.

When a node is seen only a small amount of time, the accuracy of the Proximity Map depends on the following hardware parameters: the relative speed of cars, the time necessary to open a connection, and the throughput of the connection (those last two being related to the network interface).

In our laboratory setup, relatively small speeds are attained, and a traditional WiFi (with reduced range) is used. In a real life setup, the performance of the network interface will have to match the physical constraints of a car-to-car scenario, which shall be the case with the new 802.11p interface.

Further results will be provided in the D6.4 HIDENETS deliverable.

## 7 Cooperative data backup

### 7.1 Service Description

The problem of cooperative backup of critical data consists essentially in: discovering storage resources in the vicinity (using the Proximity Map service), establishing trust with the neighbouring cars for the use of their resources (using the trust and cooperation oracle), handling a stream of data chunks to backup and assigning these chunks to the negotiated resources according to some data encoding scheme and with respect to desired properties like dependability, privacy, confidentiality, etc. On the other hand, it must also take care of the recovery phase, i.e., the data restoration algorithm.

The cooperative backup service aims to improve the dependability of the data stored on participating nodes by providing them with mechanisms to cope with hardware or software faults, including permanent faults such as loss, theft, or physical damage. To handle permanent faults, the service must provide mechanisms to store the user's data on alternate storage nodes using the available communication means.

A cooperative backup service can leverage the resources available in a device's neighbourhood. Such a service is operated in a decentralised fashion by the nodes themselves, i.e. by the vehicles in the ad-hoc domain, without the need of a centralised service in the infrastructure domain. Nodes are free to participate or not in the service. They can benefit from the service by storing backups of their data on other nodes. The nodes are expected to contribute to the service in return by donating storage and energy resources for the service. For fairness, participating nodes are expected to act as both data owners and contributors; this is ensured by using the Trust and Cooperation Oracle (TCO).

In the sequel, we use the term *contributor* when referring to a device acting in its role of storage provider; we use the term *data owner* when referring to a device in its role of "client", i.e., one that uses storage provided by the contributors to replicate its data.

### 7.2 Design, Specification, Interface

The cooperative backup uses the service of the Proximity Map to discover potential contributing nodes in its vicinity. The maximum number of hops between contributors can be specified. It also uses the trust and cooperation oracle to assess confidence in the potential contributors: it does not request storage on untrusted nodes, neither does it store data on their behalf. It also uses the networking services extensively in order to exchange data with cooperating nodes. The cooperative data backup can be seen by other building blocks as a safe repository for critical data:

- An application can request backup of some data: the cooperative backup will produce data chunks and distribute it to contributors.
- Later on the application can then request for restoration of this data: the cooperative backup service will try to find back all the data chunks, from both neighbouring nodes and the infrastructure (when available).

In the sequel, we first present some of the techniques we used to implement the distributed storage layer of the backup service. We then describe the implementation of opportunistic replication and storage provision.

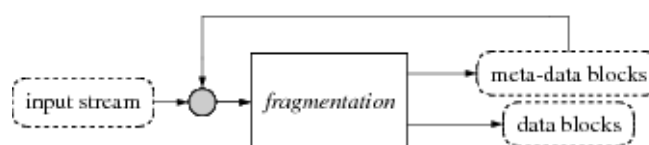
#### Storage mechanisms

Stream meta-data is information that describes which blocks comprise the stream and how they should be reassembled to produce the original stream. Such meta-data can either be embedded in each data block or stored separately. The main evaluation criteria of a meta-data structure are read efficiency (e.g., algorithmic complexity of stream retrieval, number of accesses needed) and size (e.g., how the amount of meta-data grows compared to data).

We suggest a more flexible approach whereby stream meta-data (i.e., which blocks comprise a stream) is separated both from file meta-data (i.e., file name, permissions, etc.) and the file content. This has several advantages. First, it allows a data block to be referenced multiple times and hence allows for single-instance storage at the block level. Second, it promotes separation of concerns. For instance, file-level meta-data (e.g.,

file path, modification time, permissions) may change without having to modify the underlying data blocks, which is important in scenarios where propagating such updates would be next to impossible because of mobility. Separating meta-data and data also leaves the possibility of applying the same "filters" (e.g., compression, encryption), or to use similar redundancy techniques for both data and meta-data blocks.

The separation of data and meta-data means that there must be a way for meta-data blocks to refer to data blocks. This requires that the data blocks are indexed or named, and the block-naming scheme must fulfil several requirements. First, it must not be based on non-backed-up user state that would be lost during a crash. Most importantly, the block-naming scheme must guarantee that name clashes among the blocks of a data owner cannot occur. In particular, block IDs must remain valid in time so that a given block ID is not wrongfully re-used when a device restarts the backup software after a crash. Given that data blocks will be disseminated among several peers and will ultimately migrate to their owner's repository, blocks IDs should remain valid in space, that is, they should be independent of contributor names. This property also allows for pre-computation of block IDs and meta-data blocks: stream chopping and indexing do not need to be done upon a contributor encounter, but can be performed a priori, once for all. This saves CPU time and energy, and allows data owners to immediately take advantage of a backup opportunity.



**Figure 19: Fragmentation process.**

Byte streams (file contents) can be thought of as sequences of blocks. Meta-data describing the list of blocks comprising a byte stream need to be produced and stored. In their simplest form, such meta-data are a vector of block IDs, or in other words, a byte stream. This means that this byte stream can in turn be indexed, recursively, until a meta-data byte stream is produced that fits the block size constraints, as illustrated in Figure 19.

With such a design, contributors do not need to know about the actual implementation of block and stream indexing used by their clients, nor do they need to be aware of the data/meta-data distinction. All they need to do is to provide primitives of a keyed block storage:

- *put (key, data)* stores the data block data and associates it with a key, a block ID chosen by the data owner according to some naming scheme;
- *get (key)* returns the data associated with key.

This simple interface suffices to implement, on the data owner side, byte stream indexing and retrieval. Also, it is suitable for an environment in which contributors and data owners do not trust each other because it places as little burden as possible on the contributor side. Furthermore, data encryption may be performed either at the level of individual blocks, or at the level of input streams, e.g., using public key cryptography. Of course, meta-data blocks may also be encrypted similarly.

### Opportunistic data replication

We consider two aspects of opportunistic data replication: replication strategies and related algorithms, as well as contributor discovery.

#### Replication strategies

First of all, upon arrival of new data, both data and meta-data blocks are created and inserted into a queue, where they are associated with information describing how many times they have been replicated and which contributors hold a replica. Opportunistic replication is then essentially a matter of traversing the list of block meta-data structures and selecting blocks for replication upon contributor encounter. Several replication strategies that use the available per-block information can be devised. First, per-block replication decisions are easily expressed as a function of a block's meta-data and the name of the device whose contribution is being considered. One simple and obvious per-block replication strategy would be to not send a block to a

contributor if the contributor is known to already hold a copy of the block. Such a strategy can be augmented, for example, by storing at least  $n$  different replicas of the block at  $n$  different contributors.

### Data retrieval

Upon recovery, an application requests restoration of a data stream by submitting its root key that relates to the root index and stored under a fixed name. Once it has been retrieved, the revision pointed to by this block is retrieved. This allows users to know the date of the snapshot they are about to recover from. Data retrieval generates get requests to cooperative backup services at neighbouring devices or statically registered storage providers. If the request succeeds, it populates a local block store for faster access upon future reference. This corresponds to a recovery scenario in purely ad-hoc mode. Alternatively, backup daemons can also retrieve data from a fixed store (e.g., on the Internet), provided their connectivity allows this.

### Storage provision

Cooperative Backup services also listen for connections from neighbouring nodes and serve put and get requests. Cooperative Backup services can thus act as contributors, handling storage and retrieval requests from their peers. Contributors implement per-owner name spaces, so that the block names of different data owners do not collide. This is achieved by locally maintaining per-owner stores designated by the owner's identifier. Upon a successful handshake, the backup daemon opens the peer's block store, and subsequent put and get requests operate on this store.

A contributor can choose to reject storage requests from a data owner, for instance because it does not have sufficient resources, or because it does not trust the data owner. The latter is achieved using the Trust and Cooperation Oracle.

## 7.3 Implementation

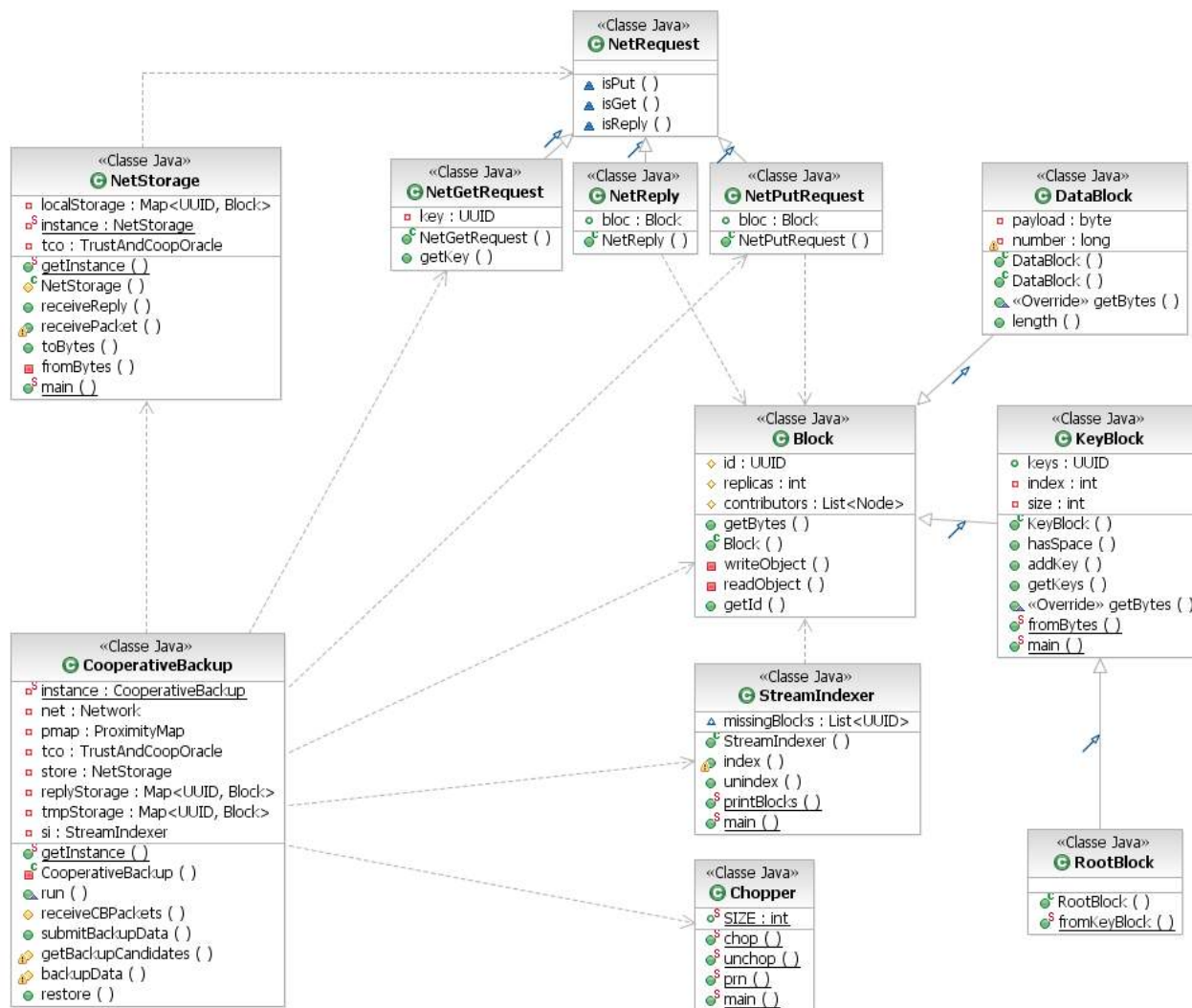
The Cooperative Backup Service is implemented in Java by a *cooperativeBackup* package comprising the necessary Java classes. The main classes are pictured in Figure 20 and described in the following.

### 7.3.1 Fragmentation and construction of a data block tree

The *Chopper* and *StreamIndexer* classes together implement the fragmentation processes illustrated in Figure 19. The role of the *Chopper* class is to cut backup data into an ordered list of blocks (*chop*) and to perform the opposite operation (*unchop*), from data blocks to a data stream. The actual implementation uses fixed sized blocks in order to better suit the small MTU size of typical networks - which is typically 1500 bytes for Ethernet-like networks. The idea is to have one block together with its meta-data to fit into a single network packet.

The *StreamIndexer* class complements the *Chopper* class. Its role is to produce a tree of uniquely identified data blocks that can be distributed over the network (*index*) and reversely from a tree of blocks to an ordered list of blocks (*unindex*). It uses the *Block*, *DataBlock*, *KeyBlock* and *RootBlock* classes that form together a tree of blocks.

The main idea is that the *StreamIndexer* *index* method produces a *DataBlock* and its associated key (a unique identifier) for each fixed-sized data chunk given by the *Chopper*. The *DataBlock* keys are stored into *KeyBlocks*, which are recursively handled by the *StreamIndexer*, i.e. a key is also produced and stored into a *KeyBlock*. Ultimately, the last meta-data block produced is the *RootBlock*, it is not assigned a unique identifier as the user submits its own identifier. This last identifier uniquely identifies the data flow as a whole: it references the *RootBlock*, which has keys of *KeyBlocks*, which have keys of *KeyBlocks* or *DataBlocks*, etc.



**Figure 20: The classes of the cooperativeBackup Java package.**

The reverse process is carried out in the *unindex* method: from a unique identifier (the key of the *RootBlock*), it produces an ordered List of data chunks. First, it needs to retrieve the *RootBlock* and to extract the keys it stores. The Blocks referenced by those keys are either *DataBlocks* or *KeyBlocks*. *KeyBlocks* are extracted the same way the *RootBlock* is. The *DataBlocks* are just retrieved and inserted in the ordered list. During this process, the identifiers of the *Blocks* that cannot be retrieved are inserted into a list of missing blocks called *missingBlocks*. When the *unindex* method fails, the restore process can thus try to recover this *missingBlocks* list, i.e. request it from its neighbours or wait for a connection to the infrastructure to retrieve it.

### 7.3.2 Networked Block Storage

The *NetStorage* class implements a keyed block storage, as described in 7.2, over the network, serving remote storage (*put/get*) requests. The *NetStorage* class is a singleton, i.e. only one instance of the class is instantiated. Users get access to this single instance using the *getInstance* method.

The requests are encoded within the *NetRequest*, *NetGetRequest*, *NetPutRequest* and *NetReply* classes. These classes embed the data necessary for each request, i.e. a Block of data (actually a *DataBlock*, *KeyBlock* or *RootBlock*) for *NetPutRequest* and *NetReply*, and a unique identifier (using the Java provided UUID class<sup>2</sup>) for *NetGetRequest*. Those four request classes implements the *Serializable* interface so that the *NetStorage*

<sup>2</sup> A UUID is used to identify a data flow. They are to be generated by the application. Any kind of UUID can be used: i.e. random UUIDs, public keys, etc.

can use serialisation in order to produce network packets from *NetRequests*. The behavior of this service is rather simple: it stores the blocks for those it receives *NetPutRequest* and serves the blocks into *NetReply* for those it received *NetGetRequest*. Before serving a *NetGetRequest* or a *NetPutRequest*, the service uses the TCO to choose whether or not to serve the client.

### 7.3.3 Cooperative Backup Service API

The *CooperativeBackup* class implements the actual Cooperative Backup Service using the services of the other classes of the package. It serves backup and restoration requests to the application level (*submitBackupData/restore*). It also uses the *ProximityMap Service* and the *Trust and Cooperation Oracle* to select the nodes that it will ask for storage contribution. The *CooperativeBackup* class is also a singleton. Users get access to this single instance using the *getInstance* method.

The following code illustrates the use of the Cooperative Backup API. It generates some synthetic data, submits this data to the Cooperative Backup singleton, restores the data, and compares the restored data to the original one.

```
// Creation of synthetic data to backup
int sizeOfData=16*1024*1024;
byte []inputData=new byte[sizeOfData];
java.util.Random rnd=new java.util.Random();
rnd.nextBytes(inputData);

// Unique Identifier of the backup client
UUID myID=UUID.randomUUID();

// The CB service is a singleton
CooperativeBackup cb=CooperativeBackup.getInstance();

// Data is then submitted to the CB
cb.submitBackupData(inputData, myID);

// Data can be restored this way
byte[] result=cb.restore(myID);

// And restored data can be checked to validate the CB
if(java.util.Arrays.equals(inputData, result)) {
    System.out.println("Data restored correctly");
} else {
    System.out.println("Failure");
}
```

## 7.4 Summary

The role of the cooperative backup service is essentially:

- Discovering storage resources in the vicinity using the Proximity Map service.
- Establishing trust with neighbouring cars for the use of their resources using the trust and cooperation oracle.
- Handling a stream of data chunks to backup, and assigning these chunks to the negotiated resources according to some data encoding scheme (and with respect to desired properties like dependability, privacy, confidentiality, etc). This was described in section **Fejl! Henvisningskilde ikke fundet.**
- Storing data chunks on behalf of other nodes by providing a networked block storage interface as described in section 7.3.2.
- Taking care of the recovery phase, recollecting necessary data chunks and reassembling them.

The *cooperativeBackup* package described in this section conforms to the specification of the Cooperative Backup Service from the HIDENETS architecture. The real performance still has to be experimentally evaluated once integrated in a full test bed. Such a test bed is not yet completely realized at the time of writing. However, it is worth noting that partial validation, using *JUnit* testing, of the Cooperative Backup has been performed successfully. The implementation is able to backup up to 16Mbytes of data at once.

Performance and dependability evaluation is rather difficult as it depends heavily on the system's model: i.e. network bandwidth, the rate at which a node encounters other participants, the rate at which it can connect to the infrastructure, the rate at which nodes fail, etc. The interested reader can refer to [14] and to WP4 deliverables.

## 8 Intrusion-Tolerant Agreement

The intrusion-Tolerant Agreement service provides HIDENETS entities with various flavours of agreement protocols (e.g., binary consensus, vector consensus) allowing these entities to coordinate their actions. The protocols operate correctly provided that less than one third of the involved entities try to disrupt their operation.

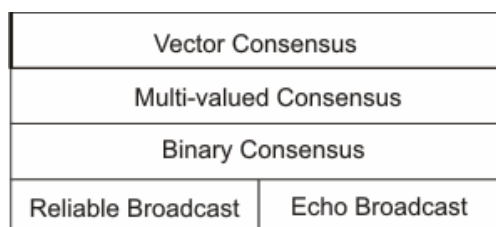
### 8.1 Service Description

In an inherently distributed environment such as HIDENETS, it is essential that the participating entities are able to coordinate their actions. The building block of every coordinated activity is an agreement. In order to preserve their safety and efficient operation, entities have to reach agreement on their actions. This agreement can be on a common action (e.g., “all vehicles brake”, or “all vehicles accelerate”), or non-conflicting actions (e.g., “I stop at road intersection, you pass”). Reaching agreement in distributed systems subject to arbitrary faults is a non-trivial and fundamental problem, and is relevant when some kind of coordinated activity must take place.

The Intrusion-Tolerant Agreement Service is relevant for any application that requires any kind of coordinated activity, i.e., applications involving entities that need to perform agreement on some common arbitrary value or set of values. For instance, the Platooning application can use the agreement service to allow vehicles to coordinate their road manoeuvring, or to decide on a common platoon leader or on a common platooning speed.

### 8.2 Design, Specification, Interface

This service is designed as a protocol stack called RITAS (Randomized Intrusion-Tolerant Asynchronous Services) where each layer of the stack provides a different form of agreement. The protocols allow a group of  $n \geq 4$  entities to reach agreement (of several kinds) even if a subset of the entities are *corrupt*, i.e., they take arbitrary actions in order to disrupt its operation. More specifically, up to  $f$  entities can be corrupt where  $n \geq 3f+1$ . Additionally, it is assumed that all processes are within communication range of each other, or some kind of underlying routing infrastructure that can reliably deliver messages must be in place. In order to cope with denial-of-service attacks, the protocols do not rely on time for their correct operation, i.e., they are completely asynchronous. They cope with the FLP impossibility result [17], which states that consensus (a crucial form of agreement) is impossible in asynchronous systems with process crashes, by employing randomization [32]. The stack is organised as shown in Figure 21 and is inspired by [13]. Related implementations and performance evaluations are described in [32] and [33]. An extension to this stack is described in [39], where one can take advantage of a centralised server within the infrastructure domain to increase performance. In fact, this extension has been developed in the scope of HIDENETS and is described Deliverable D2.2 [9].



**Figure 21: Intrusion-Tolerant Agreement service protocol stack.**

At the bottom of the stack depicted in Figure 21 there are the *reliable broadcast* and *echo broadcast* protocols. Reliable broadcast ensures that, upon a broadcast, all correct entities (which behave as specified and do not crash) either deliver the same message or no message at all. Echo broadcast is a weaker, but more efficient, version of reliable broadcast. In the case where the sender is corrupt, it does not guarantee that all correct entities will deliver the message. It only ensures that the subset of correct entities that deliver, will do it for the same message.

Above these broadcast primitives is binary consensus. This protocol, being the simplest form of consensus, is where randomization is applied to circumvent the FLP result. With it, entities can agree on a binary value.

Above binary consensus is *multi-valued consensus*, which allows entities to propose and decide on a value with an arbitrary domain. Depending on the proposals, the decision is either one of the proposed values or a default value.

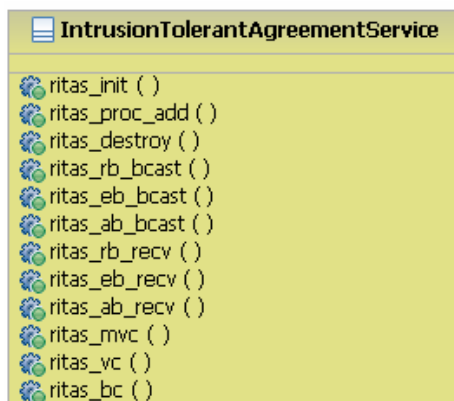
At the top of the stack is *vector consensus*. This protocol allows entities to agree on a vector with a subset of the proposed values. It ensures that every correct entity decides on the same vector  $V$  of size  $n$ ; if an entity  $p_i$  is correct, then the vector element  $V[i]$  is either the value proposed by  $p_i$  or a default value, and at least  $f+1$  elements of  $V$  were proposed by correct processes.

The API of RITAS revolves around a data structure `ritas_t`. This structure holds all the necessary context information (variables and data structures) for a communication session, and is completely opaque to the application programmer. The functions provided by the API can be divided into two categories: Context management and service requests. A typical RITAS session is composed by four basic steps executed by each process:

- i) Initialize the RITAS context by calling `ritas_init()`;
- ii) add the participating processes to the context by calling `ritas_proc_add()`;
- iii) call the service request functions as many times as desired, and
- iv) destroy the RITAS context by calling `ritas_destroy()`.

There are service request functions for the broadcast, and consensus protocols. Each broadcast protocol has associated two functions: `ritas_XX_bcast()` and `ritas_XX_rcv()`. The former is utilized to transmit a message, and the latter blocks the program until a message arrives (where  $XX$  can be  $rb$ ,  $eb$ , or  $ab$ , for reliable broadcast, echo broadcast, or atomic broadcast, respectively). Each consensus protocol has an associated `ritas_YY()` function that proposes a value, blocks until a decision is made, and returns the decision value (where  $YY$  can be  $bc$ ,  $mvc$ , or  $vc$ , for binary consensus, multi-valued consensus, or vector consensus, respectively).

An overview of the service interface is shown in Figure 22.



**Figure 22: Interface for the Intrusion-Tolerant Agreement Service.**

## 8.3 Implementation

The protocol stack was implemented in the C language as a shared library, which provides a simple interface to applications wishing to use the protocols. The protocol stack runs in a single thread, independent of the application thread. The following subsections describe the internal structure of the protocol stack, and provide an insight into the design considerations and practical issues that arose during the development of RITAS.

### 8.3.1 Single-threaded vs. Multi-threaded Operation

When developing a software component such as a protocol stack, it may either be implemented with multi-threaded or single-threaded operation. The RITAS protocol stack runs in a single thread, independent of the application threads.

In a typical multi-threaded protocol stack, every instance of a specific protocol is handled by a separate thread. Usually, there is a pivotal thread that reads messages from the network and instantiates protocol threads to handle messages that are specific to them. Another option is to avoid the pivotal thread, and have the protocol threads responsible for reading messages from the network.

The multi-threaded approach may be simpler to implement since there is practically no need to synchronise (each context is self-contained in a given thread), and there is virtually no need for protocol demultiplexing since messages can be addressed directly to the threads handling them. This leads to a cleaner implementation (i.e., more verbatim translations from pseudo code) because the protocol code has only to deal with one protocol instance (the context is implicit). Nevertheless, one can reach a situation where the constant context switching and synchronisation between the various threads - and a loaded system can easily have several hundreds of threads - poses a serious performance impact on the stack, and this may result in unfair internal scheduling.

A single-threaded approach, while more complex to develop, allows a much more efficient stack operation when properly implemented. A single-threaded protocol stack ensures a fair first-come, first-served scheduling as messages are processed by the relevant protocol instances one-by-one as they are received. But such an approach poses additional challenges. The contexts for the different protocol instances are not self-contained and require explicit management, which adds complexity to such tasks as message passing, protocol demultiplexing, and packet construction. The specific protocol code also becomes harder to implement since it has to juggle between multiple contexts.

Since one of the main goals of RITAS was the implementation of an efficient protocol stack, the extra complexity of a single-threaded approach was outweighed by its potential performance advantages.

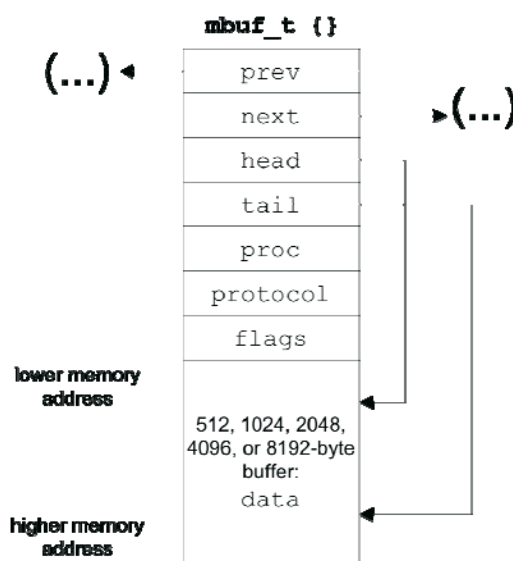
### 8.3.2 Message Buffers

In a multi-layered network protocol stack, messages need to be passed back and forth. A certain degree of flexibility is needed to manipulate the buffers that hold the messages since data may need to be prepended or appended to these buffers, existing data may need to be transformed or deleted, and the amount of operations that actually copy data needs to be kept to a minimum to reduce performance penalties. Additionally, messages may need to be tagged with several pieces of meta-information such as their size or where they came from.

This implies that there is a need for some kind of data structure, preferably hidden behind a simple interface that abstracts most of the complex operations needed for its manipulation. Examples of such operations are initialisation and destruction of messages, and the prepending and appending of data to the message. Besides being able to hold a message, this data structure should have the fields to store all the necessary meta-information for a correct and efficient message management.

In RITAS, information is passed along the protocol stack using *message buffers* (*mbuf* for short). An *mbuf* is used to store a message and several metadata related to its management. All communication between the different layers is done by passing pointers to *mbufs*. In this way, it is possible to both eliminate the need to copy large chunks of data when passing messages from one layer to another, and have a data structure that

facilitates the manipulation of messages. This data structure was inspired by the TCP/IP implementation in the Net/3 Operating System kernel.



**Figure 23: The mbuf structure.**

Figure 23 shows the structure of a *mbuf*. Each field is explained below:

- prev: Pointer to the previous *mbuf* in a *mbuf* list.
- next: Pointer to the next *mbuf* in a *mbuf* list.
- head: Pointer to the beginning of the message held by the *mbuf*. This is some memory address inside the *data* buffer, lower or equal to the *tail* pointer.
- tail: Pointer to the end of the message held by the *mbuf*. This is some memory address inside the *data* buffer, higher or equal to the *head* pointer.
- proc: Identifier of the process that sent the message held by the *mbuf*. Only relevant if it is an inbound message.
- protocol: Indicates which was the latest protocol layer to process the message.
- flags: Special flags. These indicate to the *mbuf* manipulation functions if some sort of special behaviour is necessary when processing the *mbuf*.
- data: Buffer where the actual message held by the *mbuf* is stored.

An *mbuf* is usually created when a new message arrives from the network. The RITAS network scheduler creates a *mbuf*, then it reads the message from the socket directly into the *mbuf*, and passes the *mbuf* to the appropriate protocol layer. A *mbuf* can also be created by a specific protocol layer, for instance if it needs to send a message to other processes. Every *mbuf* is reutilized as much as possible.

There are also specific rules as to when a *mbuf* should be destroyed. An outbound *mbuf* should be destroyed immediately after its message is sent to all relevant processes. The exception is when a `RITAS_MBUF_PROTECTED` flag is set. In this case, the *mbuf* was explicitly marked for no destruction by a particular protocol layer, which then becomes solely responsible for the *mbuf* destruction. For an inbound *mbuf*, the last protocol to which the *mbuf* is going to be passed is responsible for its management. A protocol layer has three options, which are mutually exclusive, after it has processed the message contained in the *mbuf*: It passes the *mbuf* to an upper layer protocol, it destroys the *mbuf*, or it reuses the *mbuf* to transmit a new message. The chosen action depends on the semantic of the protocol and the current state of the particular protocol instance context to which the *mbuf* is relevant.

### 8.3.3 Control Blocks and Protocol Handlers

Each protocol implemented in RITAS is formed by two protocol-specific components: The *control block*, and the *protocol handler*. The control block is a data structure that holds the state of a specific instance of the protocol. The protocol handler is the set of functions that implement the operation of the protocol.

The control block data structure is responsible for holding all the necessary context for the execution of a specific instance of the protocol. It keeps tracks of issues like the instance identification, the current protocol step, the values received so far, etc. The code block below presents the skeleton of a typical control block using the binary consensus protocol as an example.

```
/**
 * General Control Block Information. In one form or another, all control blocks in
 * RITAS maintain* this information.
 */
/* Identifier of the specific protocol instance. */
u_short id;
/* Indicates if this control block is chained to an upper-layer control block. */
u_char upper;
/* If so, this pointer points to it. */
void *parent;
/* A linked list of the control blocks used by this protocol instance as primitives.
 * In this case, the binary consensus protocol needs several reliable broadcast
 * control blocks in order to communicate.
 */
ritas_rccb_t **rccb;

/**
 * Now we're entering more protocol-specific state information. This can be anything
 * the protocol needs in order to maintain the logical state of the protocol
 * instance. Below are a few examples.
 */
/* The current round the protocol is in. */
u_short round;
/* The current step the protocol is in. */
u_short step;
/* The binary majority value of the specific steps of the protocol. */
u_char majority_value[3];
/* (...) */
```

Protocol handlers are formed by initialisation and destruction functions, input and output functions, and one or more functions that export the protocol functionality. The purposes of the initialisation and destruction functions are, respectively, to allocate a new control block and initialize all its variables and data structures, and to destroy the internal data structures and the control block itself. The input and output functions are used for inter-protocol communication, and both receive as parameters the respective control block and the *mbuf* to be processed. The code block below depicts the skeleton of a typical protocol handler using the binary consensus protocol as an example.

```
/*
 * Initialization function. Creates a binary consensus control block.
 */
ritas_bccb_t *ritas_bc_init(ritas_t *ctx, u_short id, void *parent);
/*
 * Destruction function. Destroys an existing binary consensus control block.
 */
void ritas_bc_destroy(ritas_t *ctx, ritas_bccb_t *bccb);
/*
 * Input function. Used by the lower-level layers to pass an mbuf to a binary
 * consensus specific protocol instance referenced by the respective control block.
 */
int ritas_bc_input(ritas_t *ctx, ritas_bccb_t *bccb, ritas_mbuf_t *m);
/*
 * Output function. Used by the upper-level layers to pass an mbuf to a binary
 * consensus specific protocol instance referenced by the respective control block.
 */
```

```

*/
int ritas_bc_output(ritas_t *ctx, ritas_bccb_t *bccb, ritas_mbuf_t *m);
/*
 * Function that exports the binary consensus protocol functionality to applications.
 */
int ritas_bc(ritas_t *ctx, u_short bcid, u_char proposal);

```

### 8.3.4 The RITAS Channel and Control Block Chaining

Since applications might execute several broadcast and/or agreement operations simultaneously, the ability to execute multiple instances of the same protocol is a prerequisite. Therefore, one needs to support many contexts for the different protocol instances. When a message is passed to a given protocol layer, that layer must be able to identify the relevant context for the message, and process the message according to it. This indicates that it is necessary to have each protocol instance uniquely identified, and to have messages addressed to specific protocol instances to avoid overlapping of multiple instances. Two techniques in RITAS make possible the efficient implementation of this functionality: The RITAS Channel, and the Control Block Chaining.

The *RITAS channel* is a special protocol handler that sits between the broadcast layers and the Reliable Channel layer (the Reliable Channel layer corresponds to the implementation of TCP and IPSec [24] that is accessed through the socket interface). It is placed in the stack such that it is the first layer to process messages after they are read from the network, and the last one before they are written to the network.

The purpose of the RITAS channel is to build a header containing an unique identifier for each message. Messages are always addressed to a given RITAS Channel. The message is then passed along to the appropriate protocol instances by a mechanism called control block chaining, described in the next section.

Control block chaining manages the linking of different protocol instances, solving several problems: It gives a means to unambiguously identify all messages, provides for seamless protocol demultiplexing, and facilitates control block management.

Control block chaining works in the following way. Suppose an application executes an atomic broadcast, and the creation of the atomic broadcast protocol instance is done by calling the corresponding initialisation function that returns a pointer to a control block responsible for that instance. Since atomic broadcast uses multi-valued consensus and reliable broadcast as primitives, the atomic broadcast initialisation function also calls the initialisation functions for such protocols in order to create as many instances of these protocols as needed. The returned control blocks are kept and managed in the atomic broadcast control block. This mechanism is recursive since second-order protocol instances may need to use other protocols as primitives, and so on. The result is a tree of control blocks that has its root at the protocol called by the application and goes down all the way, having control blocks for RITAS Channels as the leaf nodes.

A unique identifier is given to each outbound message when the associated *mbuf* reaches the RITAS Channel layer. The tree is traversed bottom-up starting at the RITAS Channel control block and ending at the root control block. The message identifier is generated by appending the protocol instance ID of each traversed node to a local message identifier that was set by the node that created the *mbuf*.

Protocol demultiplexing is done easily. When a message arrives, its identification defines an association with a particular RITAS Channel control block. The RITAS Channel passes the *mbuf* to the upper layer by calling the appropriate input function of its parent control block. The message is processed by that layer and the *mbuf* if forwarded once again in the same fashion.

### 8.3.5 Out-of-context Messages

The asynchronous nature of the protocol stack leads to situations in which a process is receiving correct messages but they are destined to a protocol instance for which a context has not yet been created. These messages - called *out-of-context* (OOC) messages - have no context to handle them, though they will, eventually.

Since the correctness of the protocols depends on the eventual delivery of these messages, they cannot simply be discarded. All OOC messages are stored in a hash table. When a RITAS Channel is created, it checks this hash table for relevant messages. If any relevant messages exist, they are promptly delivered to the upper protocol instance.

It is also possible that a protocol instance is destroyed before consuming all of its OOC messages. To avoid a situation where OOC messages are kept indefinitely in the hash table, upon the destruction of a protocol, the hash table is checked and all the relevant messages are deleted.

## 8.4 Summary

The Intrusion-Tolerant Agreement (ITA) service is being used in the scope of the Platooning test bed. It is used for a decentralised decision process where the whole platoon agrees on a common speed at which all cars will converge.

Of the available agreement protocols, we chose the vector consensus as the top layer protocol, along with a deterministic function to extract from the result vector the correct speed to agree upon. In brief, the idea is to start a new agreement round at certain moments, when there is some state change (e.g., when a driver increases the speed of its car). The consensus is executed and when it finishes executing, the result is fed to the control algorithm of the Platooning application, which sets it as the optimum speed at which to drive. The vector exchanged is structured as an array of  $\langle \text{proposal}, \text{max} \rangle$  speed elements. Each participant adds to the vector an element with the proposed speed, and the maximum speed at which it is able to drive. At the end of the consensus, a deterministic function selects the maximum of the suggested speeds that is not greater than the minimum of the maximum speeds. This way, we can be sure that all correct cars agree on a value that each one of them is able to respect.

The applicability of the Intrusion-Tolerant Agreement service in this case is not trivial, since there are a number of assumptions that might be difficult to ensure in a real scenario, such as the knowledge and the existence of a fixed number of cars in the platoon. In the practical implementation, some assumptions had to be made in order to allow the use of this service, as will be reported in detail in deliverable D6.4. In brief, we can mention the need to assume that all cars in the platoon are within range, which in practice may require the use of a communication stack a multi-hop broadcast layer (an issue addressed in WP3 [42]). We can also mention the need for secure channels, which has been achieved by using IPSec and point-to-point communication, in a transparent manner to upper layers of the service.

As a final remark, we must say that we believe that the availability of this Intrusion-Tolerant Agreement service is interesting and relevant in the context of the considered ad-hoc communication domain. Providing this kind of functionality in these environments is challenging and this is probably why there are not many works dealing with the issue. However, this is certainly an important service for many applications (as demonstrated with the Platooning test bed), whenever there is the need for coordination. We believe that the work performed within HIDENETS has contributed to a better understanding of the problem and of the possible solutions.

## 9 Concluding remarks

This deliverable reports on the final solutions for improving the dependability and resilience of applications operating in the ad-hoc domain. In particular, from the initial ideas on a potential relevant set of services to be studied, designed and developed, we finally selected six services to be developed and implemented as complex middleware services: The Diagnostic and Reconfiguration Manager, the QoS Coverage Manager, the Replication Manager, the Proximity Map, the Cooperative Data Backup and the Intrusion-Tolerant Agreement service.

We opted for producing a self contained document, including relevant material from previous deliverables, but trying at the same time to select only the strictly important parts, the ones that any reader will need to be read to be able to understand all the important pieces of our work, from the objectives of the services to their actual implementation. For the same reason we added an introductory section, where we try to provide all the necessary motivation for the contents of this deliverable, and this includes the reasons why we distinguish between solutions for ad-hoc domains from solutions also using the infrastructure domain, the reason why we separate complex middleware services and oracle services and the reason why we selected this specific set of services. In the introductory section we also discuss the close relation between the implementation work that has been done and the work concerned with the development of the proof-of-concept applications.

We believe that in general the objectives of the work package have been achieved, as we were able to define an architecture that provides clear advantages over other “more standard”, homogeneous, architectures, and we were able to implement most of the services and show their positive impact on the performance/dependability characteristics of selected applications. Although all this is a global achievement of the project, the contribution of the work performed in this work package is certainly fundamental.

In some cases the developed and implemented services were subject to evaluation tests, to allow us to quantitatively characterise the improvements that might be achieved over other solutions. In the applicable cases, a summary of these results has been also provided in this deliverable, while the full set of results will be provided in WP4 deliverables, namely in D4.2.2.

From a practical point of view, we have been able to demonstrate selected parts of the HIDENETS architecture using separate test-beds. A fully integrated demonstration could perhaps be envisaged, but it would require a considerably larger effort. Such an integrated HIDENETS solution including all the developed services (which would possibly be provided as a coherent package and could be customised for the different applications), would raise additional complexity problems that would in some sense be orthogonal to the dependability objectives of the project (note also that complexity issues are addressed in other work packages in HIDENETS, namely in WP4). Achieving a fully integrated “HIDENETS package” can be a possible direction of future work. Another open issue concerns the development of additional services that could also be useful in some situations. We can mention, for instance, event-based communication services, which are typically interesting when considering anonymous and large-scale communication, and which could extend the basic point-to-point or broadcast communication services that have been used in HIDENETS (as provided by underlying and existing communication facilities). We can also mention configuration services, which would be even more relevant in the case of a configurable platform. Finally, we must say that it would certainly be possible to consider many other (complex) services. Our set of services is relevant for some applications, but as there are so many applications constantly being envisaged, there is constant need to investigate the adequate support for all these new applications. We may say that in HIDENETS we have provided the means to develop of some of these applications with improved dependability characteristics.

## References

- [1] M. M. Artimy, W. Robertson, and W. J. Phillips, "Connectivity in inter-vehicle ad hoc networks," Faculty of engineering, Dalhousie University 2004.
- [2] A. Avizeienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *In IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11-33, 2004.
- [3] G. Bolch, S. Greiner, d.-H. Meer, and K. S. Trivedi, *Queuing Networks and Markov Chains*: Wiley-InterScience, 2006.
- [4] M. Bozinovski, H.-P. Schwefel, and R. Prasad, "Algorithm for Controlling Transaction Consistency in SIP Session Control Systems," *IEE Electronics Letters*, vol. 40, pp. 209-211, 2004.
- [5] I. de Bruin et al, Specification HIDENETS laboratory set-up scenario and components, EU FP6 IST project HIDENETS, deliverable D6.2, October 2007.
- [6] T. Camp, J. Boleng, and V. Davies, "A survey of mobility models for ad hoc network research," *Wireless Communications and Mobile Computing (WCMC)*, 2002.
- [7] A. Casimiro, P. Verissimo. Using the Timely Computing Base for dependable QoS adaptation. In *Proceedings of the 20<sup>th</sup> IEEE Symposium on Reliable Distributed Systems*, pages 208–217, New Orleans, USA, October 2001.
- [8] A. Casimiro et al, Resilient Architecture (Final Version), EU FP6 IST project HIDENETS, deliverable D2.1.2, December 2007.
- [9] A. Casimiro et al, Service level resilience solutions for the infrastructure domain, EU FP6 IST project HIDENETS, deliverable D2.2, June 2008.
- [10] I.-R. Chen, G. Baoshan, S. George, and C. Sheng-Tzong, "On failure recoverability of client-server applications in mobile wireless environments," in *Reliability, IEEE Transactions*, vol. 54, pp. 115-122, 2005.
- [11] Z. D. Chen, H. Kung, and D. Vlah, "Ad hoc relay wireless networks over moving vehicles on highways," In *MobiHoc01: Proceedings of the 2nd ACM international symposium on Mobile ad-hoc networking & computing*, New York, NY, USA. ACM Press., pp. 247-250, 2001.
- [12] M. Correia, P. Verissimo, N. F. Neves. The design of a COTS real-time distributed security kernel. In *Fourth European Dependable Computing Conference*, Toulouse, France, October 2002.
- [13] M. Correia, N. F. Neves, P. Verissimo. From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures. *Computer Journal*. 41(1):82–96, January 2006.
- [14] L. Courtès, O. Hamouda, M. Kaaniche, M.O. Killijian, D. Powell, "Dependability Evaluation of Cooperative Backup Strategies for Mobile Devices", In *Proceedings of the 13<sup>th</sup> IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'07)*, Melbourne, Victoria, Australia, December 17-19, 2007.
- [15] D. R. Cox and H. D. Miller, *The Theory of Stochastic Processes*: Chapman and Hall Ltd., 1965.
- [16] L. Falai et al, Mechanisms to provide strict dependability and real-time requirements, EU FP6 IST project HIDENETS, deliverable D3.3, June 2008.
- [17] M.J. Fischer, N.A. Lynch, M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *JACM*, 32(2):374–382, 1985.
- [18] I. Foster, A. Roy, V. Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *Proceedings of the 8<sup>th</sup> International Workshop on Quality of Service (IWQOS)*, pages 181–188, Pittsburgh, PA, June 2000.
- [19] R. Gass, J. Scott, and C. Diot, "Measurements of in-motion 802.11 networking" in WMCSA '06: In *Proceedings of the Seventh IEEE Workshop on Mobile Computing Systems & Applications*, Washington, DC, USA, 2006, pp. 69–74.

- [20] R. Gass, J. Scott, and C. Diot "CRAWDAD trace set cambridge/inmotion/tcp (v. 2005-10-01)" Downloaded from <http://crawdad.cs.dartmouth.edu/cambridge/inmotion/tcp>, Oct. 2005.
- [21] M. B. Hansen, R. L. Olsen, and H.-P. Schwefel, "Probabilistic models for access strategies to dynamic information elements," *To appear in Performance Evaluation*, 2007.
- [22] A. Helal, A. Heddaya, and B. Bhargava, "Replication Techniques in Distributed Systems," *Kluwer Academic Publishers*, 1996.
- [23] T. Henderson, D. Kotz, and I. Abyzov, "The changing usage of a mature campus-wide wireless network" in *Proceedings of the 10<sup>th</sup> annual international conference on Mobile computing and networking*, 2004, pp. 187–201.
- [24] S. Kent and R. Atkinson. Security architecture for the internet protocol. IETF Request for Comments: RFC 2093, Nov. 1998.
- [25] M.-O. Killijian, D. Powell, M. Banâtre, P. Couderc, and Y. Roudier, "Collaborative Backup for Dependable Mobile Applications," *Proc. of 2nd Int. Workshop on Middleware for Pervasive and Ad-Hoc Computing (Middleware 2004)*, pp. 146-149, Oct. 2004.
- [26] D. Kotz, T. Henderson, and I. Abyzov, "CRAWDAD trace set dartmouth/campus/tcpdump (v. 2004-11-09)" Downloaded from <http://crawdad.cs.dartmouth.edu/dartmouth/campus/tcpdump>, Nov. 2004.
- [27] A. Kung et al. Security architecture and mechanisms for v2v / v2i. EU project SaVeCom, deliverable D2.1, Aug. 2007.
- [28] Y. Lee, "CNU wireless traces," Downloaded from <http://networks.cnu.ac.kr/measurement/cdma-1x-evdo>, 2006.
- [29] Y. Lee, "Measured tcp performance in cdma 1x ev-do network" Passive and Active Measurement Conference, 2006.
- [30] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, *Modeling with Generalized Stochastic Petri Nets*: John Wiley & Sons Ltd., 1995.
- [31] E. V. Matthiesen, T. Renier, and H.-P. Schwefel, "A new selection metric for backup group creation in inter-vehicular networks," *In 16th IST Mobile and communications summit*, 2007.
- [32] H. Moniz and N. F. Neves and M. Correia and P. Veríssimo. Randomized Intrusion-Tolerant Asynchronous Services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 568–577, June 2006.
- [33] H. Moniz, N. F. Neves, M. Correia, A. Casimiro, P. Verissimo. Intrusion Tolerance in Wireless Environments: An Experimental Evaluation. In *Proceedings of the 13<sup>th</sup> IEEE Pacific Rim Dependable Computing Conference*. December 2007.
- [34] R. L. Olesen, M. B. Hansen, and H.-P. Schwefel, "Quantitative analysis of access strategies to remote information in network services," *In Global Telecommunications Conference, GLOBECOM - IEEE*, 2006.
- [35] S. Porcarelli, M. Castaldi, F. Di Giandomenico, A. Bondavalli, P. Inverardi. A Framework for Reconfiguration-Based Fault-Tolerance in Distributed Systems, In R. De Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, LNCS. Springer-Verlag, 2004. also ICSE-WADS2003, Post-Proceeding of ICSE-WADS2003.
- [36] S. Porcarelli, F. Di Giandomenico, A. Chohra, A. Bondavalli. Tuning of database audits to improve scheduled maintenance in communication systems, in Computer Safety, Reliability and Security, *Proc. of the 20<sup>th</sup> International Conference SAFECOMP 2001*, Budapest, Hungary, pages 238–248. Lecture Notes in Computer Science 2187. Springer, 2001.
- [37] I. Porter, J.E., J. Coleman, and A. Moore, "Modified ks, ad, and c-vm tests for the pareto distribution with unknown location and scale parameters", *IEEE Transactions on Reliability*, 41(1):112–117, Mar 1992.
- [38] M. Radimirsch et al, Use case scenarios and preliminary reference model, EU FP6 IST project HIDENETS, deliverable D1.1, September 2006.

- [39] H. P. Reiser, A. Casimiro. Optimizing Byzantine Consensus for Fault-Tolerant Embedded Systems with Ad-Hoc and Infrastructure Networks. *Proc. of the 4<sup>th</sup> Int. Workshop on Dependable Embedded Systems (WDES'07)*, Beijing, China, October 2007.
- [40] “SAF, Service availability Forum” <http://www.saforum.org>.
- [41] F. Siqueira, V. Cahill. Quartz: A QoS architecture for open systems. *IEEE International Conference on Distributed Computing Systems (ICDCS 2000)*, pages 197–204, 2000.
- [42] I.-E. Svinnet et al., Report on resilient topologies and routing – final version, EU FP6 IST project HIDENETS, deliverable D3.1.2. June 2008.
- [43] K. S. Trivedi. “Probability and statistics with reliability, queuing and computer science applications”. John Wiley and Sons, 2002.
- [44] UMass Trace Repository, “UPRM wireless traces,” Downloaded from <http://traces.cs.umass.edu/index.php/Network/Network>, 2006.
- [45] P. Veríssimo. Travelling through wormholes: Meeting the grand challenge of distributed systems. In *Proc. Int. Workshop on Future Directions in Distributed Computing*, pages 144–151, Bertinoro, Italy, June 2002.
- [46] P. Veríssimo, A. Casimiro. The Timely Computing Base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930, 2002.
- [47] D. Xu, D. Wichadakul, K. Nahrstedt. Multimedia Service Configuration and Reservation in Heterogeneous Environments. *Proceedings of the International Conference on Distributed Computing Systems*, pages 512–519, 2000.