

## **Resilient architecture (preliminary version)**

A. Casimiro, A. Bondavalli, A. Ceccarelli,  
A. Daidone, L. Falai, P. Frejek,  
F. Di Giandomenico, G. Huszerl, M.-O. Killijian,  
A. Kövi, E.V. Matthiesen, O. Mendizabal,  
H. Moniz, T. Renier, M. Roy

DI-FCUL

TR-07-19

September 2007

Departamento de Informática  
Faculdade de Ciências da Universidade de Lisboa  
Campo Grande, 1749-016 Lisboa  
Portugal

Technical reports are available at <http://www.di.fc.ul.pt/tech-reports>. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.



**Project no.:** IST-FP6-STREP- 26979  
**Project full title:** Highly dependable ip-based networks and services  
**Project Acronym:** HIDENETS  
**Deliverable no.:** D2.1.1  
**Title of the deliverable:** Resilient architecture (preliminary version)

<b>Contractual Date of Delivery to the CEC:</b>	31st December 2006
<b>Actual Date of Delivery to the CEC:</b>	22nd December 2006
<b>Organisation name of lead contractor for this deliverable</b>	FCUL
<b>Author(s):</b> António Casimiro (editor), Andrea Bondavalli, Andrea Ceccarelli, Alessandro Daidone, Lorenzo Falai, Peter Frejek, Felicita Di Giandomenico, Gábor Huszerl, Marc-Olivier Killijian, András Kövi, Erling V. Matthiesen, Odorico Mendizabal, Henrique Moniz, Thibault Renier, Matthieu Roy	
<b>Participant(s):</b>	4
<b>Work package contributing to the deliverable:</b>	WP2
<b>Nature:</b>	R
<b>Version:</b>	4.0
<b>Total number of pages:</b>	75
<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 is the first deliverable within this work package, a preliminary version of the resilient architecture.

The deliverable builds on previous results from WP1, the definition of a set of applications and use cases, and provides a perspective of the middleware services that are considered fundamental to address the dependability requirements of those applications. Then it also describes the architectural organisation of these services, according to a number of factors like their purpose, their function within the communication stack or their criticality/specificity for resilience.

WP2 proposes an architecture that differentiates between two classes of services, a class including timeliness and trustworthiness oracles, and a class of so called complex services. The resulting architecture is referred to as a “hybrid architecture”. The hybrid architecture is motivated and discussed in this document.

The services considered within each of the service classes of the hybrid architecture are described. This sets the background for the work to be carried on in the scope of tasks 2.2 and 2.3 of the work package. Finally, the deliverable also considers high-level interfacing aspects, by providing a discussion about the possibility of using existing Service Availability Forum standard interfaces within HIDENETS, in particular discussing possibly necessary extensions to those interfaces in order to accommodate specific HIDENETS services suited for ad-hoc domains.

# Table of Contents

<b>REFERENCES .....</b>	<b>4</b>
<b>1 EXECUTIVE SUMMARY .....</b>	<b>8</b>
<b>2 MIDDLEWARE LEVEL REQUIREMENTS .....</b>	<b>10</b>
2.1 MIDDLEWARE LEVEL REQUIREMENTS FOR THE SELECTED USE-CASES.....	10
2.2 SELECTED USE CASES FOR WP2 .....	16
2.3 IDENTIFICATION OF REQUIRED SERVICES.....	17
<b>3 ARCHITECTURE OVERVIEW .....</b>	<b>23</b>
3.1 RATIONALE .....	23
3.2 ARCHITECTURAL PRINCIPLES.....	25
3.3 REQUIREMENTS ON INFRASTRUCTURE AND LOW-LEVEL SUPPORT .....	27
<b>4 TIMELINESS AND TRUSTWORTHINESS ORACLES .....</b>	<b>28</b>
4.1 RELIABLE AND SELF-AWARE CLOCK.....	28
4.2 LOCAL AND DISTRIBUTED DURATION MEASUREMENT.....	31
4.3 TIMELY TIMING FAILURE DETECTOR.....	33
4.4 FRESHNESS DETECTOR.....	36
4.5 AUTHENTICATION .....	39
4.6 TRUST AND COOPERATION ORACLE .....	40
<b>5 COMPLEX SERVICES.....</b>	<b>44</b>
5.1 DIAGNOSTIC MANAGER .....	44
5.2 RECONFIGURATION MANAGER .....	47
5.3 QOS COVERAGE MANAGER .....	49
5.4 REPLICATION MANAGER .....	52
5.5 MOBILE AGENT MANAGER .....	53
5.6 INCONSISTENCY ESTIMATION MODULE .....	55
5.7 PROXIMITY MAP .....	56
5.8 COOPERATIVE DATA BACKUP.....	57
<b>6 THE HIGH AVAILABILITY SERVICES DEFINED BY THE SERVICE AVAILABILITY FORUM.....</b>	<b>60</b>
6.1 APPLICABILITY OF THE SERVICE AVAILABILITY FORUM (SAF) SPECIFICATIONS TO AD-HOC CLUSTERS .....	61
6.2 AVAILABILITY MANAGEMENT FRAMEWORK (AMF) AND THE APPLICATION INTERFACE SPECIFICATION (AIS) SERVICES .....	62
6.3 CONCLUSIONS ON THE APPLICATION OF AIS SERVICES INTERFACES IN THE HIDENETS ARCHITECTURE .....	71
<b>7 FINAL REMARKS .....</b>	<b>72</b>
<b>ANNEX I: FUNCTIONALITIES OF THE HA SERVICES DEFINED BY THE SERVICE AVAILABILITY FORUM .....</b>	<b>73</b>

## References

- [1] M. Radimirsch et al, Use case scenarios and preliminary reference model, EU FP6 IST project HIDENETS, deliverable D1.1, September 2006.
- [2] F. Cristian, C. Fetzer. The Timed Asynchronous Distributed System Model. *IEEE Trans. Parallel Distributed Systems*, pages 642–657, June 1999.
- [3] P. Veríssimo, A. Casimiro. The Timely Computing Base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930, 2002.
- [4] D. Mills. Network Time Protocol (Version 3) Specification, Implementation, 1992.
- [5] P. Veríssimo. On the Role of Time in Distributed Systems. *Proceedings of the 6th Workshop on Future Trends of Distributed Computing Systems (FTDCS'97)*. pages 316–321, October 1997.
- [6] P. Veríssimo, C. Almeida. Quasi-Synchronism: A Step Away from the Traditional-Fault-Tolerant Real-Time System Models. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, 7(4):35–39, 1995.
- [7] H. Kopetz. The Time-Triggered Architecture. *The First IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 1998.
- [8] P. Veríssimo, L. Rodrigues, A. Casimiro. CesiumSpray: a Precise and Accurate Global Time Service for Large-scale Systems. *Journal of Real-Time Systems*, 12(3):241–294, Kluwer Academic Publishers, November 1997.
- [9] J. Elson and K. Romer. Wireless sensor networks: a new regime for time synchronisation. *SIGCOMM Comput. Commun. Rev.*, 33(1):149–154, 2003.
- [10] F. Cristian. A Probabilistic Approach to Distributed Clock Synchronisation. *Distributed Computing*, 3(3):146–158, 1989.
- [11] F. Cristian, C. Fetzer. Probabilistic Internal Clock Synchronisation. *Symposium on Reliable Distributed Systems*, pages 22–31, 1994.
- [12] G. Alari, A. Ciuffoletti. Implementing a Probabilistic Clock Synchronisation Algorithm. *Real Time Systems*, 13(1):25–46, July 1997.
- [13] T. Chandra, S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
- [14] P. Veríssimo, A. Casimiro, C. Fetzer. The Timely Computing Base: Timely actions in the presence of uncertain timeliness. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 533–542, New York, USA, June 2000.
- [15] A. Casimiro, P. Veríssimo. Using the Timely Computing Base for dependable QoS adaptation. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, pages 208–217, New Orleans, USA, October 2001.
- [16] T. Renier, E. Matthiesen, H-P. Schwefel, R. Prasad. Inconsistency Evaluation in a Replicated IP-based Call Control System, *The 3rd International Service Availability Symposium*, Helsinki, May 2006.
- [17] R. Olsen, H-P. Schwefel, M. Hansen. Quantitative analysis of access strategies to remote information in network services, *Globecom'06*, San Francisco, November 2006.
- [18] I.-E. Svinnset et al., Report on resilient topologies and routing – preliminary version, EU FP6 IST project HIDENETS, deliverable D3.1. December 2006.
- [19] C. Fetzer, F. Cristian. A Fail-Aware Datagram Service. *Proceedings of the 2nd Annual Workshop on Fault-Tolerant Parallel and Distributed Systems*, Pages 55-69, 1998.
- [20] A. Casimiro, P. Verissimo. Timing Failure Detection with a Timely Computing Base. *3rd European Research Seminar on Advances in Distributed Systems*, 1999.

- [21] C. Almeida, P. Verissimo. Timing Failure Detection and Real-Time Group Communication in Quasi-Synchronous Systems. *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy, June 1996.
- [22] A. Bondavalli, E. De Giudici, S. Porcarelli, S. Sabina, F. Zanini. A Freshness Detection Mechanism for Railway Applications. *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 292- 301, March 2004.
- [23] P. Verissimo, L. Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.
- [24] ITU-R Recommendation TF.460-4: Standard-frequency and time-signal emissions. Annex 1.
- [25] D. Powell. Failure mode assumptions and assumption coverage. In *Proceedings of the 22nd IEEE Annual International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 386-395, Boston, USA, July 1992.
- [26] A. Bondavalli, L. Simoncini. Failure Classification with respect to Detection, in Esprit Project N.3092 (PDCS), 1st Year Report, IEEE-CS, Los Alamitos, CA (USA), May 1990.
- [27] D.P. Siewiorek, R.S Swartz. *Reliable Computer Systems: Design and Evaluation*. A.K. Peters, 1998.
- [28] M. Pizza, L. Strigini, A. Bondavalli, F. Di Giandomenico. Optimal discrimination between transient and permanent faults. In *Third IEEE International High-Assurance Systems Engineering Symposium*, pages 214–223, 1998.
- [29] G. Mongardi. Dependable computing for railway control systems. In *European Dependable Computing Conference (DCCA-3)*, pages 255–277, Mondello, Italy, 1993.
- [30] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, F. Grandoni. Threshold-based mechanisms to discriminate transient from intermittent faults. *IEEE Transactions on Computers*, 49(3):230–245, 2000.
- [31] L. Spainhower, J. Isenberg, R. Chillarege, J. Berding. Design for fault-tolerance in system es/9000 model 900. In *22<sup>nd</sup> IEEE International Symposium on Fault Tolerant Computing Systems - FTCS 22*, pages 38–47, Boston, Massachusetts, 1992.
- [32] A. Daidone, F. Di Giandomenico, A. Bondavalli. Hidden Markov Models as a support for diagnosis: formalization of the problem and synthesis of the solution, In *25<sup>th</sup> IEEE Symposium on Reliable Distributed Systems (SRDS 2006)*, Leeds, UK, October 2006.
- [33] 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.
- [34] 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 20th International Conference SAFECOMP 2001*, Budapest, Hungary, pages 238–248. Lecture Notes in Computer Science 2187. Springer, 2001.
- [35] A. Chohra, F. Di Giandomenico, S. Porcarelli, A. Bondavalli. Towards optimal database maintenance in wireless communication systems, in *Proc. World Multiconference on Systemics, Cybernetics and Informatics*, Vol. 1 Information Systems Development, Florida, USA, pages 571–576, July 2001.
- [36] B.T. Loo, A. LaMarca, G. Borriello. Peer-To-Peer Backup for Personal Area Networks. IRS-TR-02-015, UC Berkeley; Intel Seattle Research (USA), May 2003.
- [37] S. Elnikety, M. Lillibridge, M. Burrows. Peer-to-peer Cooperative Backup System. The USENIX FAST, 2002.
- [38] L.P. Cox, B. D. Noble. Pastiche: Making back-up cheap and easy. *Fifth USENIX OSDI*, pages 285–298, 2002.
- [39] L.P. Cox, B. D. Noble. Samsara : Honor Among Thieves in Peer-to-Peer Storage. *Proc. 19th ACM SOSR*, pages 120–132, 2003.

- [40] E. Sit, J. Cates, R. Cox. A DHT-based backup system. Technical report, MIT Laboratory for Computer Science, 2003.
- [41] S. Quinlan, S. Dorward. Venti: a new approach to archival storage. *Proceedings of the First USENIX Conference on File and Storage Technologies*, pages 89–101, Monterey, CA, 2002.
- [42] M. Landers, H. Zhang, K.L. Tan. Peerstore: better performance by relaxing in peer-to-peer backup. *Proceedings of the Fourth International Conference on Peer-to-Peer Computing*, pages 72–79, Zurich, Switzerland, 2004.
- [43] C. Batten, K. Barr, A. Saraf, S. Treptin. Pstore: a secure peer-to-peer backup system. Technical Report MIT-LCS-TM-632, MIT Laboratory for Computer Science, 2001.
- [44] J. Cooley, C. Taylor, A. Peacock. ABS: the apportioned backup system. Technical report, MIT Laboratory for Computer Science, 2004.
- [45] F. Junqueira, R. Bhagwan, K. Marzullo, S. Savage, G.M. Voelker. The phoenix recovery system: rebuilding from the ashes of an internet catastrophe. In *Proc. of HotOS-IX*, pages 73–78, Lihue, HI, May 2003.
- [46] B.F. Cooper, H. Garcia-Molina. Bidding for storage space in a peer-to-peer data preservation system. In *Proc. 22<sup>nd</sup> International Conf. on Distributed Computing Systems*, pages 372–381, Vienna, Austria, July 2002.
- [47] E. Hsu, J. Mellen, P. Naresh. Dibs: distributed backup for local area networks. Technical report, Parallel & Distributed Operating Systems Group, MIT, USA, 2004.
- [48] A. Muthitacharoen, R. Morris, T.M. Gil, B. Chen. Ivy: a read/write peer-to-peer file system. *SIGOPS Oper. Syst. Rev.* 36(SI): 31–44, 2002.
- [49] J. Kubiawicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao. Oceanstore : an architecture for global-scale persistent storage”. In: *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 190–201, 2000.
- [50] A.V. Goldberg, P.N. Yianilos. Towards an archival intermemory. *Proceedings of IEEE International Forum on Research and Technology Advances in Digital Libraries (ADL'98)*, IEEE Society, pages 147–156, 1998.
- [51] C. Randriamaro, O. Soyey, G. Utard, F. Wlazinski. Data distribution in a peer to peer storage system. *Journal of Grid Computing (JoGC), Special issue on Global and Peer-to-Peer Computing*, Springer, Lecture Notes in Computer Science, 4(3): 311–321, September 2006.
- [52] M.O. Killijian, D. Powell, M. Banâtre, P. Couderc, Y. Roudier. Collaborative backup for dependable mobile applications. *Proceedings of the 2nd workshop on Middleware for Pervasive and Ad-hoc Computing*, pages 146–149, 2004.
- [53] L. Courtès, M.O. Killijian, D. Powell. Storage Tradeoffs in a Collaborative Backup Service for Mobile Devices. LAAS Report 05673 & *Proceedings of the 6th European Dependable Computing Conference (EDCC)*, pages 129–138, Coimbra, Portugal, October 2006.
- [54] Service Availability Forum web site, <http://saforum.org>
- [55] Service Availability Forum, Application Interface Specification (SAF-AIS-B.02.01), [http://saforum.org/specification/AIS\\_Information/](http://saforum.org/specification/AIS_Information/)
- [56] I. Crnkovic, M. Larsson, editors. *Building Reliable Component-Based Software Systems*. Artech House Publishers, 2002.
- [57] F. Siqueira, V. Cahill. Quartz: A QoS architecture for open systems. *IEEE International Conference on Distributed Computing Systems (ICDCS 2000)*, pages 197–204, 2000.
- [58] I. Foster, A. Roy, V. Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *Proceedings of the 8th International Workshop on Quality of Service (IWQOS)*, pages 181–188, Pittsburgh, PA, June 2000.

- 
- [59] 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.
- [60] C. Aurrecochea, A. T. Campbell, L. Hauw. A survey of QoS architectures. *Multimedia Systems Journal – Special Issue on QoS Architecture*, 6(3):138–151, 1998.
- [61] 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.
- [62] M. Correia, P. Veríssimo, N. F. Neves. The design of a COTS real-time distributed security kernel. In *Fourth European Dependable Computing Conference*, Toulouse, France, October 2002.
- [63] M. Tuxen, Q. Xie, R. Stewart, M. Shore, L. Ong, J. Loughney, M. Stillman. Requirements for Reliable Server Pooling (rfc3237), IETF, January 2002.

# **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*”.

In HIDENETS we consider applications and use cases within communication environments made of ad-hoc/wireless multi-hop domains as well as infrastructure network domains, which raise a number of challenges to be addressed both at the architectural level and on the definition of the middleware services. These challenges include, among others, the intrinsic unreliability of communication, the potentially high number of users, calling for scalable solutions, and the openness of the environments, making systems subject to both accidental and malicious interferences.

To address these challenges and to achieve the general objectives of HIDENETS, the specific objectives of this work package, as stated in the Technical Annex, are:

- *To define new architectural paradigms and constructs targeted to i) deal with the uncertainty of the environments, providing the means to reconcile this uncertainty with the predictability that is sometimes required; ii) deal with specific classes of faults, detecting and recovering from them so as to keep the operation within certain specified levels of QoS*
- *To propose middleware solutions for enhancing resilience, availability and security in both domains of mobile ad-hoc communication and infrastructure-based communication. The proposed solutions have to accommodate a balanced harmonization among security/trustworthiness, availability/performance and reliability aspects. These solutions range from algorithms and higher layer protocols to actual services*

The main objective of this deliverable is to provide a preliminary description of the HIDENETS resilient architecture, which will be used as the basis for further activities that will be done in WP2 and in other project work packages. In particular, the deliverable describes initial ideas and on-going work with respect to 1) the definition of architectural solutions to address the identified challenges, 2) the definition of middleware services aimed at providing improved dependability, and 3) the definition of interfaces to be used by application developers. Given the preliminary nature of this deliverable, it is expected to further refine these initial ideas, which will be described in the final version of the HIDENETS resilient architecture deliverable, D2.1.2, in December 2007. Nevertheless, this deliverable sets the ground for the work to be carried out in WP2, namely the definition and development of the HIDENETS middleware solutions, which will be done in tasks 2.2 and 2.3. It also provides initial input, in the form of service definitions, for the design and testing activities carried out in WP5.

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

The work in WP2 started 3 months after the beginning of the project, according to the work plan and timetable defined in the Technical Annex. This deliverable reflects the progress achieved so far in WP2, more specifically in its first task (Task 2.1 – Resilient Architecture).

In order to define architectural constructs and middleware solutions to address the requirements of applications considered in HIDENETS, the initial part of the deliverable, Section 2, is focused on these requirements. More specifically, it builds on results from WP1, in particular on the applications and use cases that were defined and that are presented in deliverable D1.1, and provides a summary of the middleware level requirements for each use case, highlighting the main challenges that have to be addressed by the middleware services, such as the need to provide dependable quality of service (QoS), trustworthiness or availability. It then introduces the overall set of middleware and communication level services that have been identified as relevant in HIDENETS, which constitute the full HIDENETS node software architecture that is being defined in the context of both WP2 and WP3. Finally, Section 2 also explains how the services are grouped together.

Section 3 is then devoted to introduce and explain the architectural solution that is proposed to HIDENETS in order to enable the design of improved solutions for resilience and dependability. This solution is based on the principle of architectural hybridization, that is, on using an architecture in which different parts of the system enjoy different properties, like synchrony or trustworthiness. To better introduce the solution, the section provides an overview of related work in the area of distributed system models and approaches to deal with uncertainty. A brief overview of the possible impacts of adopting a hybrid architecture in terms of low-level infrastructure requirements is also provided in this section.

Sections 4 and 5 provide initial descriptions of the middleware services addressed in HIDENETS. In fact, given the hybrid architecture adopted, two classes of services are considered and each class is addressed in its own section. First, the services belonging to the class of timeliness and trustworthiness oracles are described, which include services that provide critical functionality with respect to timeliness or security aspects. These are typically simple services. The remaining services, described right after, are designated as complex services, and encompass services related to monitoring, reconfiguration and other management activities. The description of each service tries to include all information that might be relevant for upcoming activities in the course of the project. For instance, it includes input and output relations to other services and an overview of what needs to be known to construct the service. It also includes, for some services, initial descriptions of the interfaces to be provided. It is expected that these initial descriptions of the middleware services will serve as basis for the work to be done in the remaining two tasks of WP2. Naturally, it is also expected that as a result of these developments, the description of the services and their interfaces will be refined and improved. On the other hand, it is also expected that these initial descriptions might provide the necessary input for the work that will be developed in the context of WP5, that is, for modelling and testing activities.

The last part of the deliverable (Section 6) is concerned with the presentation of Service Availability Forum (SAF) standard interfaces, and a discussion of the applicability of these standards in HIDENETS to address the requirements of the considered use cases. Since SAF interfaces have been designed having in mind the operation in infrastructure based environments, a specific challenge in using these standard interfaces consists in verifying if, and how they can fit the specific needs and the characteristics of the operation in ad-hoc environments. Nevertheless, it is expected that existing SAF interfaces will be used whenever possible. HIDENETS applications or services that need to make use of functionality that is already defined by the SAF, should rather use the SAF interfaces than define that functionality once again in a different way.

## 2 Middleware level requirements

This section starts by summarizing the middleware level dependability requirements for the use cases that were identified in WP1. It therefore closely follows the text provided in deliverable D1 [1], revisiting the middleware level requirements for each use case. Then, for each use case, a list of fundamental challenges is provided.

In order to address these dependability requirements, it is necessary to rely on a number of services (e.g. for monitoring, for failure detection, for management) providing dependability related functionalities. Therefore, we then identify the required services for each application, which altogether are included in a complete view of the HIDENETS node software architecture. The HIDENETS node software architecture has been developed within WP2 and WP3 and includes not only the services provided as middleware level services, but all the services that should be present in a HIDENETS node.

### 2.1 Middleware level requirements for the selected use-cases

We start by reviewing the use cases considered in WP1. We i) list the applications involved in each of them, ii) describe the main middleware level requirements and iii) summarise the fundamental challenges for dependability that are associated to each use case. These middleware level requirements are specified in terms of a set of middleware level properties, which we now briefly review (for detailed descriptions of the use cases and middleware properties, please refer to deliverable D1.1 [1]):

- **Timeliness of data:** Refers to the freshness of data, ensuring that data is delivered to applications in a bounded amount of time and is thus up-to-date.
- **Logical consistency:** Ensures that the state of a set of replicas is consistent with each other.
- **Temporal consistency:** This property is essentially relevant in the context of real-time data representation. It ensures that at any point in time the value of some (real-time) entity stored at a replica is not too far apart from the real value of that entity at that same point in time.
- **Trustworthiness of data:** The concept of trustworthiness refers to the degree of confidence a service user may have that the service will perform as expected and, in particular, that it will satisfy a set of security properties. A trustworthy service is dependable with respect to security properties.
- **Robustness:** This specialised secondary attribute of dependability characterises systems that are dependable with respect to external faults. Therefore, the robustness of middleware solutions is especially meaningful when external faults constitute a relevant threat.
- **Message ordering:** Different properties with respect to message ordering can be considered. This includes FIFO, Causal, Total and Temporal ordering. All these properties are of more relevance in the context of group communication.
- **Completeness, accuracy and timeliness (of failure detection):** This property concerns the detection of crash failures. *Completeness* refers to the ability of a failure detector to detect every failure that occurs. *Accuracy* refers to the ability of the failure detector to not make mistakes, that is, wrongly detect failures when they do not occur. *Timeliness* refers to the ability of the failure detector to detect failures within given time bounds.

### 2.1.1 Use case: platooning

- Application of interest: *platooning (PT)*
- Middleware level requirements:
  - Timeliness of data: Data needs to be timely, i.e. fresh, meaning that the delay between generation and processing of the message at the receiver needs to be kept very small.
  - Logical consistency: The data distributed inside the platoon must not be modified. It is absolutely necessary that all following cars receive the same control data, even in case it is transmitted by any other platoon member. Data integrity is also important in this case.
  - Temporal consistency: As the transmitted control information may refer to rapidly changing entities, there must be mechanisms to ensure the consistency of the information made available to the application with regard to the actual value of the entities. An application should only use such control information if it knows that the information is temporally consistent, that is, if the error with regard to the real value is within some known and bounded error. For some types of information, this can be ensured by establishing a temporal validity bound (that is, an interval during which the information is always temporally consistent).
  - Trustworthiness of data: The data exchanged must be strictly limited to the platoon. Only the platoon members shall be authorised to send and to receive control information, and the follower cars must only process data provided by other members of the same platoon. Besides authenticity, also confidentiality may be important in this case, to prevent unauthorised entities to have access to the data being exchanged.
  - Robustness: Required. The reaction to a failure should in any case be fail safe, i.e. not inject additional traffic threats into the road network.
  - Message ordering: Probably not an issue, but depends on how consistency among the platoon members is achieved. The availability of communication services with total ordering properties may simplify the achievement of consistent views.
  - Completeness, accuracy and timeliness (of failure detection): If a failure occurs, the detection needs to be 1. timely (highest priority), 2. complete, 3. accurate (lowest priority).
- Fundamental challenges:
  - 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. HIDENETS needs to investigate the sources of delay in the networking part and to consider appropriate means for timeliness.
  - It is essential that all messages sent out can reach all addressed platoon members. This relates to throughput performance as well as to transmission errors. HIDENETS needs to consider means to ensure sufficient throughput and a reliable communication link. The availability of reliable broadcast or multicast services can also be very useful in this case.
  - A worst case situation is if two large platoons driving in opposite directions meet. In this case, the amount of radio resources needed by the two platoons increases in the affected geographic area and needs to be guaranteed for each of the platoons. HIDENETS needs to develop appropriate means to detect such events and to re-distribute the available radio resources quickly and efficiently.
  - Any information inside the platoon needs to be trustworthy. False messages may lead to traffic accidents. HIDENETS needs to develop trust mechanisms.
  - A platooning application provides a reasonable 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, HIDENETS needs to find adequate solutions to support the adaptation of the application in a dependable way.

### 2.1.2 Use case: Infotainment and work with highly mobile terminals

- Applications of interest: *Video conference (VC)*, *Online Gaming (OG)*, *Audio and video streaming (AVS)*, *Streaming data (SD)*, *Non-interactive data communication and messaging (NIDC)*, *Interactive data (ID)*
- Middleware level requirements:
  - Timeliness of data: The requirements are variable depending on the specific application. However, in general it should be assumed that an upper bound on the timeliness of data has to be set, in order to configure time-related parameters.
  - Logical consistency: Not an issue, since in general communication is client-server based. However, some applications might be fully distributed, like online gaming, thus requiring some consistency, even if weak forms of consistency, among the participants.
  - Temporal consistency: In general, not an issue.
  - Trustworthiness of data: Depending on the nature of transmitted data, there might be privacy and/or confidentiality requirements. It might also be necessary to ensure that data is received from a trusted sender.
  - Robustness: This is seen as an important property, but not a critical one. In general, when it is not possible to handle faults then service provisioning will simply be interrupted and restarted later. Satisfying a certain Service Level Agreement, that is, a certain quality of service is the most important.
  - Message ordering: In general it should be ensured that communication takes place through FIFO connections.
  - Completeness, accuracy and timeliness (of failure detection): Since the use case typically involves only client-server connections, the issue is typically to timely detect the failure of a peer, without compromising the accuracy of the detection.
- Fundamental challenges:
  - This use case does not include very time critical applications. The fundamental issue is the preservation of a certain quality of service, and ensuring that failures do not affect the application in a way that is perceived by the user. Nevertheless, timeliness issues must be considered. This is true, for example, for information provisioning, on-line gaming and teleconference applications.
  - Ensuring data consistency between clients and server is important in some applications. For instance, for applications that require document uploads or data synchronisation it is necessary to ensure some kind of atomicity of operations despite failures, so that the application terminates with a complete result. Inconsistency between the different versions may cause difficulties. The same is true for web-shopping applications.
  - Privacy, confidentiality and authenticity in the ad-hoc domain are important properties that should be secured.
  - Effective QoS specifications and adaptation strategies are required.

### 2.1.3 Use case: Car accident

- Applications of interest: *Black-Box (BB)*, *Emergency communications (EC)*.
- Middleware level requirements:
  - Timeliness of data: There are timeliness requirements in both applications considered. In EC it is necessary to secure, for instance, the promptness of notifications, that is, ensuring that notifications are delivered within a certain amount of time. In BB it is necessary to make sure that the data instances stored securely are sufficient to reflect the state of the original car over time, even if some fragments of the data are lost.
  - Logical consistency: Unlike the infotainment use case, this use case considers logical consistency requirements due to the BB application. The backed up data must be consistent, i.e. the data reaching the fixed server, or the cooperating cars, should not be modified. Only the producer must be able to write the data, the other entities have no need for write (or even read) access on the data. We can also talk about integrity of the data. Inconsistent failures might occur when the original information at the provider side differs from the data copied at the participating cars.
  - Temporal consistency: In the BB application it is important that the information disseminated in the ad-hoc domain and stored in the infrastructure domain reflects the real situation. Within a timeframe, only the most up-to-date data needs to be backed-up securely. However, this is probably not a critical requirement (due to the inherent off-line use of information).
  - Trustworthiness of data: In the BB application, 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. There are also authenticity requirements to consider in EC applications.
  - Robustness: The integrity and the availability of the black-box data should be satisfied even in the presence of accidental or malicious threats, taking into account different types of failure modes, including timing failures and value failures and different types of faults, including external faults. This is true, in general, for all the applications considered in this use case.
  - Message ordering: Again due to the offline use of information, simple time-stamping of each data fragment should be sufficient for ensuring that the data can be retrieved in the correct order. This allows, in particular, to secure FIFO order.
  - Completeness, accuracy and timeliness (of failure detection): node failure detection mechanisms should be implemented at different levels to ensure that whenever faults occur, the corresponding errors are detected and signalled to ensure that they are properly handled. In the EC applications it is important to detect failures as soon as possible (to enable quick recovery procedures), but without compromising the accuracy of failure detection. Whereas in BB, failures of cars don't necessarily have to be detected.
- Fundamental challenges:
  - Timeliness issues are especially challenging in EC applications, like Emergency vehicle warning and online notification to hospital. They are also important for the BB application.
  - Concerning EC, quality of service issues are also relevant, to secure minimum quality levels in multimedia related applications that are considered in this use case. It may be important to use QoS adaptation and reconfiguration strategies. Techniques for differentiated service quality, differentiated resilience, highly resilient routing and differentiated rerouting may help to achieve this.
  - Availability of the black box information: this is the main goal of the BB application, the information availability must be maximised despite faults.
  - There are a number of security related challenges involved in the BB application. Confidentiality and privacy: the black box information should be accessible only from authorised 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. Integrity of the black box information: 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.

#### 2.1.4 Use case: Assisted transportation

- Applications of interest: *Traffic flow control (TFC)*, *Floating car data (FCD)*, *Traffic sign extension (TSE)*, *Hazard warning between vehicles (HzW)*, *Maintenance-software updates (MSU)*.
- Middleware level requirements:
  - Timeliness of data: Data should be provided in a timely manner, where the timeliness requirement depends on the considered application. The quantity and frequency of timely data needed from an application to deliver the correct service clearly depends on the application. For example, in HzW the delay should in general be kept very small to allow timely reactions (typically in the order of seconds or even less), while in FCD the timeless requirements are less binding ( $\approx 1$  minute for the data of the next 10 km ahead,  $\approx 10$  minutes if the distance is within 100 Km,  $\approx 1$  hour within 500 km).
  - Logical consistency: The applications considered in this use case do not require explicit replication of data. Therefore, logical consistency is not a fundamental requirement. It is however important to note that in Distributed FCD (DFCD) or in HzW several cars may disseminate different information, which will be handled and processed by the application. The information must be checked, condensed and cleaned in order to secure that all cars have a consistent view.
  - Temporal consistency: Temporal consistency requirements are important in this use case since it includes applications dealing with real-time data that is continuously changing over time, thus possibly becoming invalid (temporally inconsistent) after a certain interval of time. This is the case of FCD, HzW and TSE.
  - Trustworthiness of data: In general, for all the applications in this use case (FCD, TSE, HzW, MSU) it needs to be assured that the data stems from a car which is trustworthy and not from a malicious source. For safety reasons, this is especially important in the case of the HzW and the MSU applications.
  - Robustness: It characterises systems that are dependable with respect to external faults. Therefore, the robustness of middleware solutions is especially meaningful when external faults constitute a relevant threat. In TSE, the reaction to a failure should be, as a last resource, that the traffic sign stops transmitting (that is, switches to a fail-safe state) rather than issuing wrong information. In HzW the reaction to a failure should in any case be fail safe, similarly to the TSE application, i.e. not inject additional traffic threats into the road network.
  - Message ordering: There are no specific message ordering requirements.
  - Completeness, accuracy and timeliness (of failure detection): All the failure detection requirements are important in this use case. However, their ranking is not clear, since it depends not only on the specific application, but also on how the application is built. For instance, in FCD it is more important to detect all failures than to detect them in a timely way. On the other hand, for applications involving safety issues, like HzW, the timeliness of detection is more important.
- Fundamental challenges:
  - In this use case the main objective is to assist the driver on the road. Therefore, the information provided by the several applications must be accurate (with respect to the real situation) and consistent (with respect to the information provided to other drivers). Therefore, ensuring both logical and temporal consistency requirements is one of the most important challenges.
  - There is a trade-off between providing fresh information and saving communication resources. One interesting research challenge is to find solutions that address this trade-off, ensuring the

required consistency properties with parsimonious use of network resources. This may involve adaptation strategies and the use of event based communications in replacement of periodic dissemination of data.

- In this use case, ensuring that the information is produced by trustworthy entities is particularly important for safety reasons. This implies not only the need to authenticate the parties involved in some application, but it may involve the need for additional solutions in order to verify their reputation.

### 2.1.5 Use case: Brigade communication

- Application of interest: *Mobile mission control centre (MMCC)*.
- Middleware level requirements:
  - Timeliness of data: The application has only standard requirements regarding the timeliness of data. Each member of the brigade has to receive actual data. Most of the data to be transferred are static or changing with a very low frequency. Strict requirements may arise when data on movement scheduling or coordination have to be communicated.
  - Logical consistency: The application has only standard requirements in logical consistency. The mission control centre has to maintain a consistent view of actual data over the brigade.
  - Temporal consistency: The application has only standard requirements in temporal consistency. The caching function requires standard functions to remove outdated data. The communication delays within the brigade should not endanger the consistency of the typically infrequently changing mission data.
  - Trustworthiness of data: The application requires authorisation and authentication to ensure that the ad-hoc network is kept restricted to selected participants.
  - Robustness: As the mission control centre is a single point of failure in the communication between the restricted network and the infrastructure, the feasibility of the application depends on its robustness.
  - Message ordering: The application has no special requirements with respect to message ordering.
  - Completeness, accuracy and timeliness of failure detection: The application has no special requirements against the applied failure detection techniques.
- Fundamental challenges:
  - The devices of a road reconstruction brigade are exposed to an increased risk of accidental faults. (This is equally true for police or fire brigades.) The treatment of faults of the centre and that of faulty communication lines offer research challenges. The restricted nature of the ad-hoc network may provoke intrusion attempts of any kind. Through tampering with mission data, one may attempt to impede a mission.
  - Content failures are to be taken into account in the case of faults of the communication channels, of the caching, of the planning/scheduling of moves or of the traffic control. Timing failures can arise from faults in the caching strategies. Measures are to be taken to detect communication channel and member outages, and data damages. Inconsistent failures might occur when accidental faults happen, but malicious faults – caused by an intruder – can turn into well designed consistent failures.
  - Caching and proxy services should be further investigated. A generally flexible but secure way of identification of potential brigade members has further security related aspects. The dependable use of the untrusted outsiders in communicating traffic control data and confidential data raises further dependability questions. The use of the outsiders implies routing aspects as well (they should not route messages too far from the working area). The mission control centre has to efficiently transmit high-priority-small-size data and low-priority-large-size data (e.g.

engineering information) as well. From the bandwidth differences between the local ad-hoc network and the dedicated link to the infrastructure domain, other networking issues may arise.

### 2.1.6 Use case: Service discovery in ad-hoc networks

- Application of interest: *Ad-hoc service providers (AhSP)*.
- Middleware level requirements:
  - Timeliness of data: The application has only standard requirements in the timeliness of data.
  - Logical consistency: The application requires no special logical consistency properties.
  - Temporal consistency: The application's feasibility highly depends on the temporal consistency of the polling messages and their replies.
  - Trustworthiness of data: The application's feasibility highly depends on an authentication method that guarantees the non-repudiation of the messages of the requesters and offering providers.
  - Robustness: The application has no special requirements in robustness.
  - Message ordering: The application's feasibility may highly depend on the correct ordering of the reply messages. (It depends on the business logic of the requester that may rely on this ordering.)
  - Completeness, accuracy and timeliness of failure detection: The application has no special requirements against the applied failure detection techniques. Undetected communication failures endanger a fair competition if they affect a part of the ad-hoc network only.
- Fundamental challenges:
  - How to limit the broadcast to the cars near by to the geographic location of the service requester.
  - How to fairly distribute the messages and simultaneously avoiding a complete flooding of the communication channel. The fair distribution of the broadcast poll and fair transfer of the replies should be guaranteed even if competing service providers are included in the message transfers. The application can be ruined if the reply (and the offer of the quality of service within) cannot be enforced afterwards (it implies security issues).

## 2.2 Selected use cases for WP2

From the six use cases above, three of them are considered to be more relevant for WP2 and are therefore typically the ones considered in the remaining text of the deliverable. The choice is made considering the aspects that seem to be “more challenging” for the HIDENETS project, in particular because they involve timeliness, security or safety requirements, alone or in combination, that are particularly difficult to address in the considered environments using the already existing design and development methodologies and technologies.

The selected use cases for WP2 are the following:

- Platooning
- Car accident
- Assisted transportation

A brief overview of the main challenges involved in each of these use cases will reveal that they cover the several different aspects of dependability. In brief, the platooning use case involves relevant consistency and safety aspects, the car accident use case involves availability, confidentiality and privacy aspects, and the assisted transportation use case involves temporal consistency, safety, quality of service and authenticity aspects.

Quite clearly, when looking at all these use cases and applications as a whole, there are various different (and possibly conflicting) requirements that should be satisfied by the middleware supporting them. This is

challenging from the point of view of the software architecture. In particular, we believe that the following two design principles must be reflected in the architecture:

- **Service modularity:** Depending on the specific requirements of some application, it is only necessary to use a limited amount of middleware functionalities. Therefore, it is interesting to construct the middleware in a modular fashion, assigning to each module a specific functionality or service. This would allow, for instance, configuring specific solutions by composition of the strictly necessary services. Other advantages of composition are also described in the literature [56].
- **Service differentiation:** Middleware services will be accessed through well defined Application Programming Interfaces (APIs). However, not every middleware service should be made accessible to applications for general use. It may be more interesting, or even necessary, to differentiate services according to their criticality or specificity. This will provide a better isolation and shielding of the critical services, which will thus enable to more easily construct them with the necessary properties (typically timeliness, or security properties).

Interestingly, both principles seem recursive, since it is possible that the internal service construction is made by composing different modules, as well as differentiation may be observed within a single service, which would itself be composed of different parts providing services with differentiated qualities.

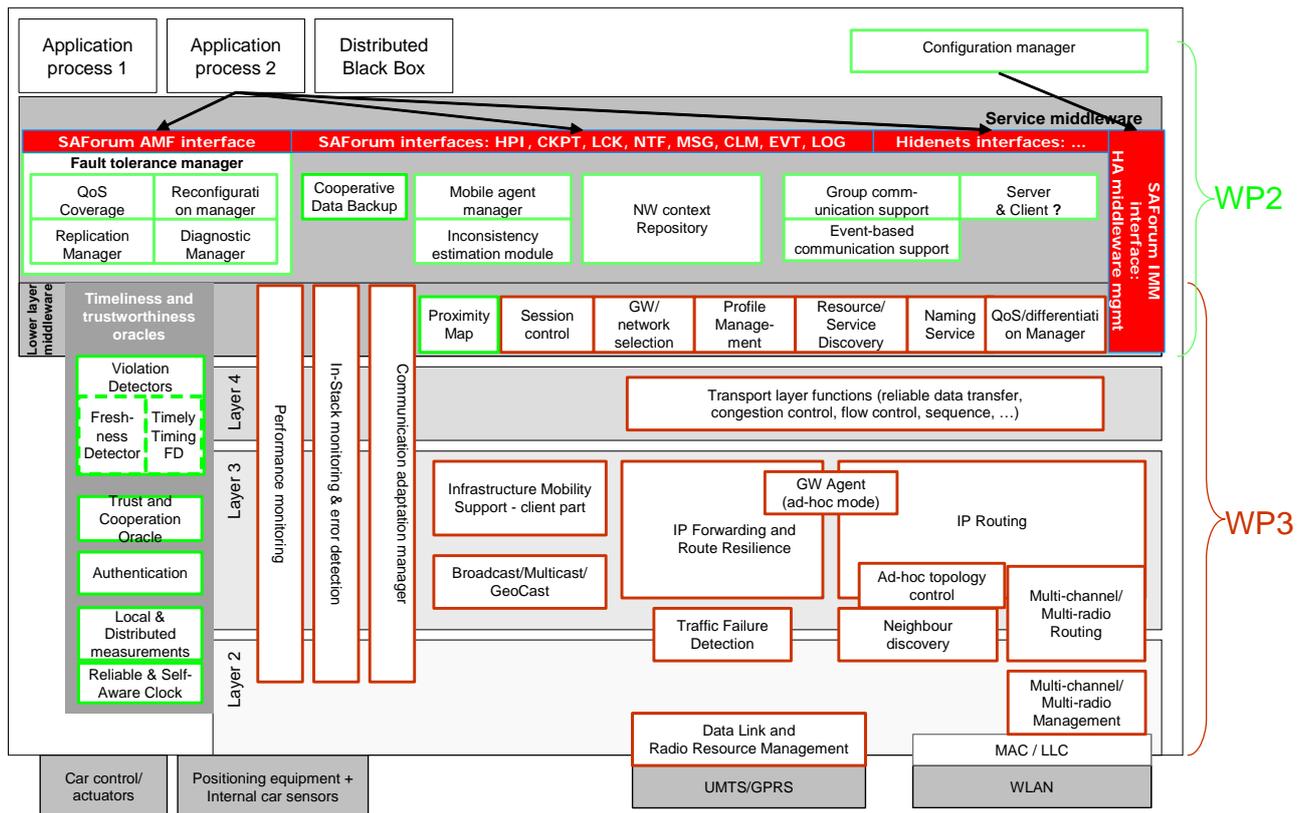
The following section provides an overview of the overall set of services that have been identified as potentially necessary in the HIDENETS node software architecture.

### 2.3 Identification of required services

As a result of the identified challenges for the middleware and communication levels, a node software architecture model is being developed by WP2 and WP3 together. This will show the main functional structure of a node (car/terminal) from the viewpoint of HIDENETS. In this document we will focus on the description of the service blocks that are relevant in the context of WP2. A detailed description of the node software architecture, including all the functional building blocks, can be found in [1]

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 1: Overall picture of HIDENETS node software architecture**

Figure 1 essentially illustrates the several building blocks that constitute a HIDENETS node, while providing also some information regarding the structuring of these building blocks within the node. The architecture reflected in the picture should be considered preliminary and it should be expected that adjustments are introduced in the course of the project. The layered structure is obvious in the picture, following the standard ISO communication layers.

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 blocks, like communication adaptation, include functionalities that are cross-layer, and therefore are represented by blocks spanning several layers. All these functional blocks will be addressed as part of WP3.

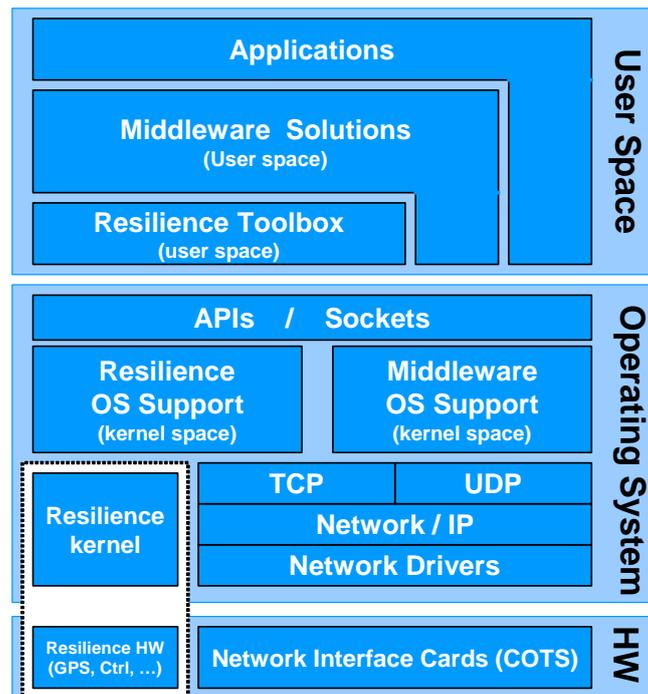
On the left of the picture, a group of functional blocks is represented outside the networking layers to explicitly convey the idea that these are special blocks, with a specific purpose, and therefore they are grouped as timeliness and trustworthiness oracles (the term ‘oracle’ being used here to convey the idea of a service that provides information one can trust). This separation is not purely logical. On the contrary, it rests on architectural principles that will be further detailed ahead, in Section 3.2. These are architecturally important blocks to achieve dependability improvements, in addition to the middleware services that we consider in HIDENETS. Because of that they are introduced in the context of WP2, in particular in the context of task 2.1. It must be said, however, that due to the nature of these building blocks, in which it is necessary to deal with strict timeliness and trustworthiness issues, their definition and development will be made essentially within task 3.3 of WP3. We also note that time- or security-related functionalities may also exist outside these oracles. In fact, applications may use standard system services, like clock or authentication services, but since these will serve ‘normal’ purposes, not specifically related to dependability, we do not include them in this figure.

Above these lower communication layers, and above the oracles, we consider the existence of a middleware layer, in which some building blocks are mainly targeted to interactions with the layers below (and therefore constitute a lower middleware layer), and some other building blocks are supposed to handle higher level interactions with the applications. A significant part of the lower middleware blocks is therefore addressed in the scope of WP3, while the remaining middleware services will be addressed and developed in WP2. The latter include several services concerned with node management (diagnostic, reconfiguration, replication, etc) and with communication support.

In the picture the application interfaces are also represented, including the standard interfaces defined by the Service Availability Forum and also the interfaces that will be defined by HIDENETS. This illustrates the idea that existing services and standards should be used whenever possible, instead of reinventing services or functionalities. However, for a matter of clarity and objectivity, the figure just represents the service blocks to be addressed within HIDENETS. All of these interfaces are used by applications, of which a Distributed Black Box application (illustrated at the top) is just an example (the description of this application is provided in deliverable D1.1 [1]).

The picture just described is clearly interesting from a completeness point of view, since it provides an idea of the complete set of services that are identified as relevant in the HIDENETS architecture. A more structured perspective is given in Figure 2, which provides a simplified view of the node architecture. In particular, it should be easy from this figure to identify the structure that was implicitly described in the text above:

- There is a **hardware layer**, which includes the ‘normal’ system hardware and, possibly, additional hardware to be used by the resilience kernel. Examples of additional hardware include a GPS antenna or a dedicated network card to be used by the oracles.
- There is an **operating system layer**, which includes networking functionalities and provides low level support to user space services. These might be resilience oriented OS support functionalities, like admission control or resource reservation, or standard OS support functionalities for generic middleware services.
- On the top there is a **user space layer** which includes the middleware solutions. The Resilience Toolbox is a part of this layer that is dedicated to resilience. We consider separated blocks for resilience oriented and non-resilience oriented solutions to provide sufficient room for services not specifically related to resilience objectives.



**Figure 2: Simplified view of node architecture with resilience services**

The concept of architectural hybridization is illustrated in Figure 2. The core of this concept is to differentiate between specialised oracles and other complex services, and let the oracles support the complex services. (As already mentioned, the oracles are enclosed in the resiliency kernel, whose definition is part of Task 2.1 of WP2. They will be implemented in Task 3.3 of WP3.) Therefore, the architecture encompasses a well defined part, i.e. the resilience kernel, which has its own synchrony and reliability properties, different from the properties assumed for the remaining system as further explained in section 3.2. Because of these better properties, the resilience kernel is specifically concerned with the provision of specialised support services, which must necessarily be services with reduced complexity.

In what follows, we briefly list the several services, or building blocks, that are grouped together as communication related services, resilience kernel services and middleware services. We also refer to the user interfaces, which play an important role both from a practical point of view, and also from a standardization point of view.

### 2.3.1 Communication related services

In this group of services we include the services that will be addressed in the scope of WP3 and that are related, in some sense, with communication functionalities. For a description of each of these services please refer to the HIDENETS deliverable D3.1 [18].

The functional blocks in this group (that are currently being considered in HIDENETS) are the following:

- Multi-channel / multi-radio management
- Multi-channel / multi-radio routing
- Neighbour discovery
- Ad-hoc topology control
- Routing
- IP forwarding and route resilience
- Broadcast/multicast/geocast
- GW agent

- Infrastructure mobility support – client part
- Traffic failure detection
- In-stack monitoring and error detection
- Performance monitoring
- Communication adaptation manager
- QoS and differentiation manager
- GW/network selection
- Profile management

### 2.3.2 Resilience kernel services

This group of services corresponds, in fact, to the timeliness and trustworthiness oracles. These are services that for a well known reason, be it more stringent requirements for timeliness or for trustworthiness, must be constructed with special care in a differentiated part of the system. This is the only possible way of ensuring that the properties secured by these services can be better than the properties of the remaining system services (including other security or timeliness related services), and therefore can be used to improve the overall dependability of the system.

The specific services included in this group are the following (they are described in Chapter 4) :

- Reliable and self-aware clock
- Local and distributed measurement
- Authentication
- Trust and cooperation oracle
- Freshness detector
- Timely timing failure detector

### 2.3.3 Middleware services

In this group we consider all the services that provide enriched functionalities to user applications, in particular related to dependability improvements. These are, in general, complex services, in the sense that they have to deal with large quantities of information, they have to manage the interaction with possibly many applications and different requirements at the same time, and also because they may need to execute complex algorithms and protocols.

The specific services included in this group are the following (they are described in Chapter 5):

- Diagnostic manager
- Reconfiguration manager
- QoS coverage
- Replication manager
- Proximity map
- Cooperative data backup
- Inconsistency estimation
- Mobile agent manager
- Network context repository
- Event-based communication support

- Group communication support
- Client-server communication support

### **2.3.4 Application Programming Interfaces**

When designing a new platform for a broader audience, the fitting choice of the APIs is fundamental. Proprietary interfaces can give the freedom and flexibility to create APIs that fully and (for the platform designer) comfortably reflect the internal architecture of the offered individual services. However, application designers prefer standard interfaces that support reusability of application concepts and code over multiple platforms with similar functionalities. To help the adoption of the future HIDENETS platform we favour standard APIs. The APIs for highly available services defined by the Service Availability Forum (SAForum) can provide some orientation for the design of standard HIDENETS interfaces

Because of the novelty of the HIDENETS concept, there are no standard APIs that could be directly used as they do not address all the special challenges identified in the HIDENETS use cases, namely those related to the operation in ad-hoc/wireless domains. The goal is to allow porting of the applications that were developed for standardised SAForum compliant high-availability platforms to the HIDENETS platform with the most possible minimal re-development efforts. It will be allowed by the reuse of the syntax and semantics of the standard SAForum interfaces wherever it is possible, and by the application of the SAForum interface definition guidelines where new special functions are to be covered. This method of interface development supports the re-use of common design patterns.

### 3 Architecture overview

#### 3.1 Rationale

There is ongoing work in the scope of WP1 in terms of the definition of a Reference Model for HIDENETS. This reference model establishes a number of dimensions that should serve as reference for the work to be developed in this work package. The aspects of synchrony and fault characterization are particularly important in face of the objectives of HIDENETS and given the challenges to be addressed. Therefore, we first review these challenges and existing literature and approaches in terms of distributed system models, before introducing the architectural principles proposed for the HIDENETS architecture.

##### 3.1.1 Fundamental challenges

As described in the HIDENETS Technical Annex, *HIDENETS addresses the provision of available and resilient distributed applications and mobile services with critical requirements on highly dynamic and possibly unreliable open communication infrastructures.*

The considered networking environments, consisting of ad-hoc/wireless multi-hop domains as well as infrastructure network domains, present a number of characteristics that raise significant challenges to be addressed in HIDENETS and, in particular, in the work to be developed in the context of WP2. These characteristics include:

- Unreliable communication due to the presence of wireless links;
- Loose reliability guarantees, due to the use of off the shelf, standard systems and components (COTS);
- Uncertain operational conditions, due to operation in open environments, with variable numbers of users or traffic flows, and due to highly dynamic network topologies;
- Variable failure modes, ranging from simpler accidental faults to malicious faults (attacks and intrusions), due to operation in open and weakly controlled environments.

Because of these characteristics, and having in mind the resilience objectives of HIDENETS, it becomes clear that some fundamental challenges in HIDENETS are to:

- Reconcile the needs for predictability and timeliness with the uncertainty of the environment;
- Improve reliability and availability despite the unreliability of the communication and of the system components;
- Ensure secure and trustworthy operation in the presence of non-trusted parties.

In HIDENETS we propose to address these challenges not only by designing new algorithmic solutions, new protocols and new services, but also by applying or devising innovative architectural solutions. We aim at opening new ways to address the antagonisms exposed in the above challenges, between what is needed and what can be assumed. This requires a special attention on the system models to be considered and used in the work, particularly concerning synchrony and faults. Synchrony models and fault models are indeed two of the several dimensions considered in the HIDENETS Reference Model for WP2 (see [18]), and these will be further addressed below.

##### 3.1.2 Distributed system models and middleware approaches

A *distributed system* is a system composed by a set of  $n$  processes (usually  $n > 1$ ) communicating by the exchange of messages. Several distributed systems *models* have been defined, in order to classify distributed systems instances in classes. A distributed system model can be defined in terms of four different and orthogonal models: *failure model*, *synchrony model*, *network topology model* and *message buffering model* [23]. In HIDENETS we are mainly concerned with the definition of two of them, which are typically the most important and differentiating ones: the *synchrony model* (also called *timing model*) and the *failure model*.

Synchrony models allow characterizing a system with respect to the existence of notions of *time* and *timeliness*. From this point of view, in the system model there might be assumptions about the timing behaviour of the distributed system, in particular on *communication times* and on *processes' reaction times*. The weakest model is the *asynchronous* model, also called *time-free*. A time-free model makes absolutely no assumptions related to time, which means that message delivery and process response times have no bounds. Furthermore, in time-free models there are neither clocks nor any other form of measuring time intervals. On the other extreme, the strongest model is the *synchronous* model, in which communication and processing delays are always bounded, and the bounds are known. Intermediate models are known as *partially synchronous* models. They do only the necessary time and/or timeliness assumptions that will allow satisfying some required properties with the appropriate dependability. For example, the Timed Asynchronous model [2] only assumes the availability of local clocks with a bounded rate of drift, and the Timely Computing Base (TCB) model [3] assumes that some parts of the system are synchronous while other parts may be asynchronous. The former is powerful enough to allow the detection of timing failures, while the latter also provides the means for a controlled and timely reaction to those failures. The use of partially synchronous models is attractive for the work in HIDENETS, since these models allow capturing the dynamics and heterogeneity of the operational environments considered in the use cases.

A distributed system can also be characterised by its *failure model* [25]. Failure mode assumptions are *assertions* on the types of errors that a component may induce in the system. A distributed system can be characterised under this point of view essentially along two directions: on the *process* failure modes and on the *communications* failure modes. In general we can identify the failure model along the *time domain* and the *value domain*. We should note that it makes sense to define failures on the time domain *only if* some notion of synchrony is defined in the model; for asynchronous systems no failures in the time domain can be defined, simply because there are no constraints in the timing behaviour of the system. In the *time domain* we have several classes: no failure, failure only by stopping, omission failure, early or late timing failure and finally arbitrary (also called undefined) failure. In the *value domain* we can identify the following classes: no failure, failure by non-code value (signalled), and finally arbitrary (non-signalled) failure.

The composition of the assumptions on the time domain and on the value domain allows obtaining the complete class of *failure mode*. Some classes of failure are of particular interest for the distributed systems:

- *process crash model*: in the time domain we consider only *stop*, and we consider no possible failure in the value domain;
- *arbitrary failure model*: in this class we have *arbitrary* failure both in the time and in the value domain.

A complete classification of the failure modes can be found in [25] and in [26]. In general, a larger class of considered failures implies a more difficult solution to the problem; a similar implication holds for the synchrony model: lighter assumptions on the synchrony of the distributed system imply more difficult solutions to the problems.

Independently of the particular synchrony or failure model that is assumed, the problem is that they will not be adequate for environments in which the synchrony or the failure modes are uncertain or unpredictable, as is the case of the environments considered in HIDENETS. Because of that, we believe that an interesting approach for HIDENETS is to follow a design philosophy based on the wormhole metaphor, introduced in [61]. This is a design philosophy for distributed systems with uncertain or unknown attributes, such as synchrony, or failure modes, which builds on the following two guiding principles:

- First, assume that uncertainty is not ubiquitous and is not everlasting— this means that the system has some parts which are more predictable than others and tends to stabilise.
- Then, be proactive in achieving predictability— in concrete, make predictability happen at the right time and right place.

These more predictable parts can be seen as shortcuts or wormholes, through which it is possible to do things much faster or reliably than apparently possible in the other parts of the system.

The wormhole 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 [62]. 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 [3]. 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 is interesting and useful in the context of HIDENETS; in fact it allows subdividing the system in two parts, one “simple and trusted” and one “complex”. The illustration of the concept has been already seen in Figure 2, where the wormhole takes the form of the *Resilience kernel*. The power of the wormhole model originates from the possibility of using a *different model* for the two *subsystems*, both for the synchrony and for the failure models: for example, we can consider the subsystem “simple and trusted” as synchronous with a crash fault model, and use lighter assumptions, as asynchronous with arbitrary faults of processes and communications, on the rest of the system. In particular, the idea of wormhole is at the basis of the architecture, and thus of the middleware, of the HIDENETS system.

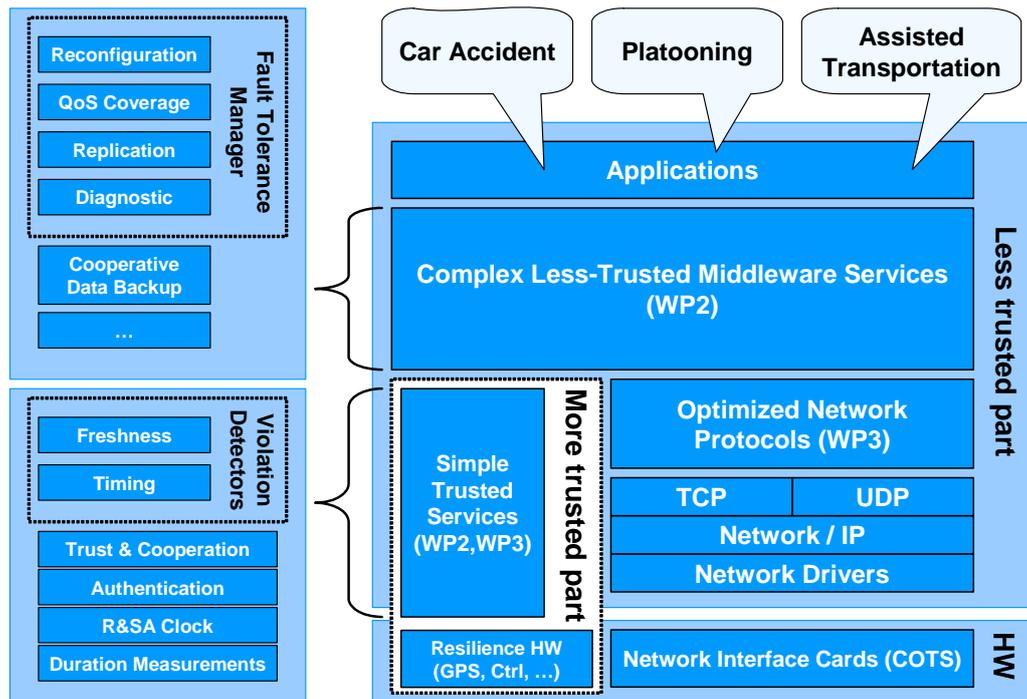
## 3.2 Architectural principles

As mentioned in Section 2.3, and as previously seen in Figure 2, in HIDENETS we consider the existence of specialised timeliness and trustworthiness oracles. In this section we address in more detail the architectural principles underlying this proposal for a HIDENETS system. We also overview specific middleware services that will be required to supporting the applications identified in Section 2.1.

As shown in Figure 3, the middleware can be seen as composed of two parts (hybrid model): a simpler and more trusted part and a complex and less trusted part. The more trusted part corresponds, as said above, to the resilience kernel depicted in Figure 2. It is important to mention that this part of the system is not simply a sub-part of the operating system kernel, but it can be a kernel on its own, with its own resources and, of course, with its own properties. The components belonging to the more trusted part (simply referred to as *trusted* from now on) are less vulnerable than the others, and then they can be used to provide more resilient services. This part can help the less trusted part for the construction of faster or more resilient solutions, or to obtain the required level of dependability at the application level.

### 3.2.1 “Simple trusted” middleware (timeliness and trustworthiness oracles)

This part of the middleware has to be designed as simple/small as possible, since it has to provide the basic trusted services. Note that this simplicity is fundamental to ensure increased predictability and trust, that is, making predictability happen in the right place, according to the design principles enunciated in the wormhole philosophy. The distributed services of this part use a *trusted network* to exchange messages with the trusted part of other nodes; this can be done by reserving some traffic channels, or by using a private network. Additional techniques can also be envisaged, and may be subject of research in the context of WP3. This network is more reliable than the payload one and it can be characterised with a more powerful model i.e. with a partially synchronous or even with a synchronous model with losses. However, full synchrony, or strict real-time behaviour, is typically very difficult to secure for the communication in the kind of ad-hoc environments that we consider in HIDENETS. Note that trusted services with a local nature do not need the existence of such trusted network. Therefore, it would be possible to envision a simplified implementation of the architecture, only composed of oracles providing services with a local nature, thus without the need for such additional resources. For instance, measuring the duration of critical local activities could be done by a local oracle.



**Figure 3: Architectural hybridization of a HIDENETS end system**

The main trusted services, to which we refer as *Timeliness and Trustworthiness Oracles*, are the following:

- A *Reliable and Self-Aware Clock*, which is a logical clock that we try to define as a global clock shared between the nodes; from a hardware point of view, it could be implemented getting the clock from a GPS module.
- A *Local and distributed duration measurement* service, that is able to measure the duration of arbitrary actions with known precision bounds.
- An *Authentication* service, which serves to secure the identity of the sender. E.g.: this service can provide mechanisms to sign flows of data.
- A *Trust and Cooperation* service that is able to evaluate locally the level of trust of neighbouring entities and to manage cooperation incentives, which is necessary for building cooperative services.
- *Violation Detectors*, which check if the assumptions introduced in the model are violated. Among violation detectors, the *Freshness detector* checks the *freshness* of every message of a data flow, with respect to application requirements. E.g.: an application can require to this service that messages of a data flow must maintain delays less or equal to a maximum values, and the middleware can control if this requirement is respected. The *Timely timing failure detector* checks if individual timed actions incur in timing failures, and does that as timely as allowed by the synchrony of the trusted part.

### 3.2.2 “Complex less trusted” middleware (complex services)

This part of the middleware is developed on top of a potentially asynchronous model. The idea is to construct the middleware services in this part in such a way that they will use and they will rely on the trusted part if and when required, but essentially for the execution of critical steps of their operation. The following complex services are currently being considered for development in the scope of HIDEENETS:

- *Diagnostic Manager*. It is in charge of monitoring critical components of the middleware and of lower levels in order to judge if everything is going well. E.g.: the Diagnostic Manager maintains information about the state of available/actually used network channels (ad-hoc or infrastructure); other services of the middleware or directly the applications can obtain this information in order to

switch between different kinds of connections.

- *QoS Coverage Manager*. It interacts with the applications using the services provided by the Diagnostic Manager. It has to assess if the application requirements are satisfied or not. E.g.: when the lower level communication guarantees decrease, the amount of video data sent in a video real-time communication can be decreased to maintain a constant video rate.
- *Reconfiguration Manager*, that includes the Software Reconfiguration Manager. The reconfiguration could be static (in presence of a particular class of fault the manager chooses a predefined strategy) or dynamic (many strategies are defined, the one used is chosen basing on the best predicted results on the dependability of the application).
- *Replication Manager*, is handling 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*, that 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*, that provides information on neighbouring nodes.
- *Inconsistency estimation*, is used as an input for the replication manager and reconfiguration manager. Inconsistency levels are estimated using network metrics and probabilities derived from the traffic and replication model descriptions.

### 3.3 Requirements on infrastructure and low-level support

As mentioned earlier, in order to use a design philosophy based on the wormhole metaphor it may not be sufficient to simply “look for” the necessary predictability within the system – specific measures may have to be taken in order to make this predictability happen when and where necessary, and this has to be done by construction.

In practice, this may be achieved by the use of specific hardware or specific (usually low-level) software solutions, to create and to secure the improved properties needed for the resilience kernel (Figure 3 illustrates this possibility). This is how we practically follow the architectural hybridization approach, introduced in Section 3.2, to ensure the required properties.

For example, a GPS device can be considered a specific hardware piece aimed at providing improved functionality and timing services. In this sense, from an architectural point of view this GPS device should be included in the trusted part of the system, and additional measures should probably be taken in order to ensure that the services provided by the GPS are kept trustworthy and cannot be affected by misbehaviours (accidental or intentional) on the less trusted part of the system.

The need for improved communication channels between distributed instances of oracle services (e.g., for distributed forms of timely timing failure detection), may also be addressed by resorting to specific or additional communication hardware and infrastructures, instead of using differentiation techniques or virtual channels over a single communication infrastructure. Depending on the criticality requirements of the envisaged application, this can be a practical solution.

Some additional considerations about specific requirements on infrastructure or low-level support can be found in the descriptions of the middleware services in the following sections.

## 4 Timeliness and trustworthiness oracles

This chapter addresses the several building blocks that constitute the resilience kernel. The descriptions presented here reflect the initial state of the work and therefore should not be understood as final descriptions of the functionality to be provided. On the other hand, at this stage of the work our main concern is with the objectives of the services, their purpose, rather than the way they will be made available. In this sense, each section provides a short description of each service and tries to identify its main objectives, the target problem that is addressed. In any case, other aspects of the service construction and operation, like the potential relations with other services, or what must be known by the system designer to construct the service, are also addressed for each of them.

### 4.1 Reliable and self-aware clock

#### 4.1.1 Short description

The *Reliable and Self-Aware Clock* (R&SA Clock) is a software component that provides an abstraction of the local clock. Its main task is monitoring the synchronisation level of the local clock with respect to a global time reference (e.g. UTC, Universal Time Coordinated, see [24]). The R&SA Clock provides information about time and it is aware of the current *precision level* of the information provided.

It can be used directly from the applications and from other higher level middleware services. The reply to a request of the current time is the following tuple:

- LocalClockTime;
- [MinUTCTime; MaxUTCTime].

LocalClockTime, MinUTCTime, MaxUTCTime are in the format Day/Month/Year, HH:MM:SS:MICROSECONDS.

The provided information is such that the actual real-time (UTC time) is within the interval [MinUTCTime; MaxUTCTime], and LocalClockTime is the most probable value of actual UTC time. LocalClockTime is always within the interval [MinUTCTime; MaxUTCTime].

The dimension of the interval [MinUTCTime; MaxUTCTime] is a dynamic variable: it changes over time; in fact it depends on the synchronisation mechanism(s) used and on the current quality of such synchronisation. The interval is usually *symmetric* respect to the medium (LocalClockTime value); the behaviour of the most used external clock synchronisation mechanism is in fact to synchronise the local clock with a given level of accuracy (*Accuracy*). In this case we have:

- $\text{MinUTCTime} = \text{LocalClockTime} - \text{Accuracy}$
- $\text{MaxUTCTime} = \text{LocalClockTime} + \text{Accuracy}$

In general we can imagine to use synchronisation mechanisms characterised by asymmetric interval [MinUTCTime; MaxUTCTime].

The main objective of the R&SA Clock is to provide to the rest of the system: i) a local view of a global distributed time reference and ii) information on the *quality* of the local view of clock.

### 4.1.2 Targeting problem

In distributed, open, dynamic pervasive systems like HIDENETS, many applications may have critical aspects to deal with, in order to provide a dependable (e.g.: safe) service. Examples of such aspects are: i) temporal order delivery (for example, the physical time of sensor readings in data fusion process); ii) temporal consistency; reduced and reliable transmission delay. These applications are usually time-dependent and intensively use timestamps. Timestamps can be obtained by reading the local clocks of the nodes of the distributed system. Time measurements can be obtained through these timestamps. Clock synchronisation is thus a fundamental process, and it is impossible to ensure it a-priori in systems like HIDENETS. Clocks, due to their imperfections, drift from real time, and their drifts may vary over time due to several causes. There is thus the need of synchronizing the clocks to each other or to a time reference, in order to enforce and maintain accuracy and precision bounds [5]. Usually, systems ([2], [6]) assume *worst-case bounds* that are necessary constraints that allow distinguishing unreliable biased data due to poorly-synchronised clocks, from reliable data collected when clock synchronisation is good. However these bounds are usually pessimistic values, far from the medium case.

What kind of synchronisation mechanisms can be used in HIDENETS and what is the quality of such mechanisms? We can imagine, with regards to mobile nodes, the availability of GPS receiver; through GPS signal all subsystems may try to synchronise their clock to a unique temporal grid (UTC time). However, the availability of the GPS receiver can be large, but not complete: we can imagine that some cars are without a GPS receiver, some others can have a receiver of bad quality, the GPS signal can be temporarily unavailable (for example due to buildings/tunnels), ... All these situations can lead to **biased local visions of UTC**. With regards to servers/nodes in the infrastructure, we can imagine to use NTP (Network Time Protocol, [4]) as synchronisation mechanism; the quality of the synchronisation of such protocol is very variable, it depends on many factors: variability of transmission delays, stratum of the available servers, ... For both mobile nodes and infrastructure servers it can be useful to be aware of the actual quality of the synchronisation respect to the global time reference.

The R&SA Clock is a specific software component to support the timestamp of events; instead of a timestamp composed only by a temporal mark, this clock allows to obtain a more complete timestamping, composed by the *temporal value LocalClockTime* and by an indicator of the *quality* of the temporal value (the interval [MinUTCTime; MaxUTCTime]). In this way the components of the system are *aware* of the quality of the timestamps, and can use also this information, e.g. for the computation of distributed measurements.

HIDENETS reference system [1] is a dynamic system, composed by a quite large and variable number of (distributed) components; in systems with these characteristics the availability of a "smart" clock, like the R&SA Clock, that is i) capable to use different mechanisms for the synchronisation with the global distributed time reference and ii) aware of the current quality of synchronisation with such time reference, can be very useful in order to build dependable middleware services and applications. Applications of a HIDENETS system that require clock synchronisation and timeliness requirements can take advantage of the awareness of the quality of synchronisation. For example, let us consider distributed ordering algorithms: the better is the quality of synchronisation of nodes, the greater is the ability to order events and to establish precedence between distributed events. An important usage of the R&SA Clock is for distributed measurement estimation. The timestamps obtained through a R&SA Clock can be used instead of a direct reading of the local clock value; using these timestamps, characterised also by the interval [MinUTCTime; MaxUTCTime] (whose dimension depends on many factors), we can obtain *multiple estimates* of a distributed measure (e.g.: mean, maximum, minimum values).

### 4.1.3 Relevant applications/use cases

The R&SA Clock is relevant in each application in which it is necessary to address real-time constraints and to provide temporal consistency; in these applications the service provided by the R&SA Clock can help to *control* the system behaviour and to provide strict resilience and real-time requirements. These applications can be summarised as follow (the applications are ordered in the list with respect to the importance of the R&SA Clock in order to provide a dependable service):

- Assisted transportation
- Traffic Sign Extension
- Emergency communication
- Distributed black box
- Hazard warning between vehicles
- Unusual driver behaviour
- Interactive data
- Floating Car Data
- Online gaming application
- Mobile mission control centre

The R&SA Clock is relevant mainly for the following use cases:

- Platooning
- Car accident
- Assisted transportation.
- Brigade communication

### 4.1.4 Entities affected

The following physical entities are affected:

- Involved cars and the servers in the infrastructure
- Subsystems involved in the clock synchronisation (it depends on the methods used for the clock synchronisation method(s)):

### 4.1.5 Input from other building blocks

The R&SA Clock receives input mainly from the local physical clock (through an interface dependent on used hardware and OS), and it receives also information from the used clock synchronisation mechanism(s), in order to evaluate/estimate the current quality of the synchronisation.

### 4.1.6 Output to other building blocks

Output of the R&SA Clock can be used directly by the application and from other building blocks of both the complex top level of the middleware and the simple services in the “Timeliness and Trustworthiness Oracles” part. Here follow examples of the two usages:

- An example of a direct usage from an application is for the Floating Car Data application. The timestamp associated to some information can be obtained through the R&SA Clock. In this way, remote hosts that receive this information can know the interval of *UTC real-time* in which the information was generated.
- An example of usage from a middleware service is for the Distributed Duration Measurement service. This service uses the timestamps obtained through R&SA Clock. The usage of this service, instead of a direct reading of the local clock value, allows obtaining *multiple* estimation of a distributed measure (e.g.: mean, maximum, minimum values): this can be made through manipulation of the intervals [MinUTCtime; MaxUTCtime] associated to every timestamp.

### 4.1.7 What needs to be known to design the building block

To design the Reliable and Self Aware Clock it is necessary to know the clock synchronisation scheme/algorithm used to synchronise the local clock with UTC real time. The great need for time-dependent protocols has increased the attention about clock synchronisation algorithms: starting (usually) from the creation of a local virtual clock at each node of the distributed system, communication between nodes allows to execute internal (e.g. TTA, [7], developed for embedded applications), external (e.g. NTP, [4], a hierarchical/master-based algorithm) or hybrid internal/external (e.g. CesiumSpray, [8]) clock synchronisation protocols.

The choice of the clock synchronisation mechanism to use is based on the characteristics of the system in which it will be used. In large scale systems, e.g. Internet, hierarchical or master-based algorithms are preferred, since the high number of nodes and the distance among them make the fulfilment of a cooperative algorithm very hard. On the other hand, in WSNs (Wireless Sensor Networks) global time-scales and a-priori synchronisation mechanisms (that is, clocks are pre-synchronised when an event occurs) often are not the right choice. This is because of the requirements [9] of energy efficiency (energy spent synchronizing clocks should be as small as possible), scalability (large population of sensor nodes must be supported), robustness (the service must continuously adapt to conditions inside the network, even in case of network partitioning) and ad-hoc deployment (time synchronisation must work with no a priori configuration, and no infrastructure available). In a HIDENETS car scenario, a car can be assumed to have access to a GPS module, which can be used for car positioning in space and also in time (with a good quality of synchronisation with UTC time). This will be useful in order to be able to maintain a small dimension for interval [MinUTCTime; MaxUTCTime].

All cited systems, protocols and algorithms show a consensus about the fact that quality of clock synchronisation is a variable factor and is very hard to predict. Many causes may be responsible for variations, such as varying communication delays, propagation delays, inaccessibility of reference nodes, network partitioning, node failures, or any possible kind of failure in the clock synchronisation algorithm or in the clocks itself.

The R&SA Clock is used by many different services and applications; it is thus of uttermost importance to design this component as a *light* and *low-intrusive* oracle, especially for its usage in car subsystems, characterised by reduced resources.

## 4.2 Local and distributed duration measurement

### 4.2.1 Short description

The *duration measurement service* can be used to measure local or distributed durations with *bounded precision*. This building block provides routines to start and stop the measurement of activities taking place locally or in a distributed setting. Local measurements are usually associated with the execution of local functions, while distributed measurements are associated to distributed executions involving communication activities through some network.

Duration measurement is an important building block for other services, such as different kinds of failure detectors. It is particularly important for **timely** timing failure detection, in which case special care must be taken during the design, to secure that measurements are done in a *timely manner*. In fact, this is one of the reasons why we consider this service as an oracle, since it must be in a part of the system with specific properties (with respect to synchrony and security, for instance) in order to behave as expected (timely, trustworthy).

Unlike the *Performance Monitoring* service (described in deliverable D3.1 [18]), this building block should be as simple as possible, allowing other building blocks to benefit from its particular properties. The performance monitoring service, on the other hand, should be seen as a more generic service, able to provide diverse and more complete information about the performance of the activities taking place or being measured in the node.

### 4.2.2 Targeting problem

The ability to measure durations, or time intervals, is essential in every system and in every situation in which the notion of time is relevant, for instance because of timeliness requirements or even because of performance requirements (e.g. throughput requirements implicitly involve timeliness requirements). In practice it is usually simple to measure time intervals using the timestamps provided by a single local clock. However, to measure durations with certain *guarantees about the measurement precision*, and to do it in a more generic way, including the measurement of durations bounded by distributed events, it is certainly useful to have a service that might do that for us.

In HIDENETS it will be necessary to measure durations for multiple purposes. The required quality (i.e., precision) of the measurements will not be necessarily the same, depending on the specific applications. However, in order to address dependability concerns, it should be at least possible to estimate the precision associated to the measurements. This is one of the objectives of this service. On the other hand, in some cases (in particular when safety issues become relevant), measurement results must be available within bounded time, in order to allow for timely reactions when necessary. This requires synchronous behaviour and is only achievable if specific design solutions (using real-time techniques) are employed. This timely behaviour is another objective of this service.

The measurement of local durations is usually regarded as a simple problem, since the availability of any local clock with bounded drift rate is sufficient to do this measurement with an upper bound for the measurement error. However, in order to achieve a more precise notion of the error associated to the measurement, it is necessary to obtain timestamps from a clock that also provides some notion about that precision.

On the other hand, the measurement of distributed durations is regarded as a more interesting problem. It is in fact a generic problem of asynchronous distributed systems, which has also been addressed in the context of clock synchronisation or in the context of real-time communication. The seminal paper of Cristian about probabilistic clock synchronisation [10] has first formally presented the *round trip* duration measurement technique, on which several other works have built thereafter [11][12]. This technique is widely used to measure distributed durations, since it allows the measurement to be done with a *guaranteed upper bound* on the measurement error. Not surprisingly, this technique can be used to achieve internal clock synchronisation between a master and a set of slave nodes, such as in the Network Time Protocol, NTP [4]. In this case, one may say that the two problems – measuring distributed durations and synchronizing clocks – are equivalent problems.

However, with additional assumptions on the synchrony of the system, like the availability of external and global time sources, it is possible to devise external clock synchronisation protocols to synchronise the clocks in a system, without the need to resort to round trip or other duration measurement techniques. In such settings, it probably makes more sense to perform duration measurements using the synchronised clocks, assuming that they provide a suitable precision.

Therefore, both because of local measurement and also because of distributed measurements, in HIDENETS our proposal is to build the duration measurement service on top of the R&SA Clock. In the case of local measurements, this will enable the provision of a measurement service that in addition to the measured interval is also able to provide a notion of the precision (*precisionLevel*) of this interval. For distributed measurements, the service will provide exactly the same kind of functionality.

We should note that one additional advantage of constructing the local and distributed duration measurement on top of the R&SA Clock is that in this way the service becomes independent and does not need to be aware of the particular techniques that are used to achieve a global time notion. For example, it is irrelevant if the underlying R&SA Clock uses a GPS to provide its service, or if it uses a round trip technique when the GPS is not available.

In practice, the duration measurement service will provide an interface including the necessary functions to start and stop the measurement of some activity, be it a local one or a distributed one. Quite clearly, the service will need to keep track of on-going measurements, for which some form of request identification will be needed. When some measurement is completed, the service will provide the measured duration with a

granularity in the order of microseconds, as allowed by the underlying R&SA Clock service. Further to this duration, the precision of the measurement (which is an upper bound on the measurement error) will also be provided, allowing the application to determine upper and lower bounds for the duration. The error may vary over time, depending on the precision level provided at each moment by the R&SA Clock.

### 4.2.3 Relevant applications/use cases

This building block is implicitly relevant for applications in several use cases, like for example the assisted transportation and the platooning use cases. However, its relevance is more explicit when considering services or building blocks that will directly use the duration measurement service interfaces to collect measurements for the duration of local or distributed activities. This is the case of the freshness detector and the timely timing failure detector.

It is possible to say that this service is relevant for all the applications with timeliness or temporal consistency requirements. This includes, for instance, “Floating car data”, “Hazard warnings and information from other vehicles” or “Platooning”. In all these cases it is important to have a precise notion of the delays involved in the communication in order to provide correct service or even ensure some safety properties of the application.

### 4.2.4 Entities affected

This is a basic building block, providing a distributed service with an end-to-end semantics, and in this sense it should be present in every end node (both in cars and servers in the infrastructure), but not necessarily in intermediate entities (routers, switches).

### 4.2.5 Input from other building blocks

The R&SA clock service will be used as a basic building block for construction of the Local and Distributed measure service.

### 4.2.6 Output to other building blocks

This building block provides duration measurements that may be used for several purposes. Single measurements may be “observed” by the Timely Timing Failure Detector building block (see Section 4.3) in order to detect single timing failures (whichever the measured activity). Measurement of activities involving data transfers that require freshness can also be requested to this service by the freshness detector (see Section 4.4). Additionally, multiple measurements can be requested and collected for monitoring purposes, for example by the QoS coverage manager (see Section 5.3), which will keep histories of these measurements in order to perform statistical analysis, generate histograms and probability distribution functions or do other on-line monitoring activities based on them.

### 4.2.7 What needs to be known to design the building block

In the first place, in order to design this service it is necessary to know the exact interface provided by the R&SA Clock service. This will allow the service to obtain timestamps with known precision levels in a distributed manner and thus calculate the requested time intervals, also with known precision bounds. Since the service will be used to measure the duration of possibly many activities executed in parallel, it is necessary to keep track of all the ongoing measurements. Moreover, this information about ongoing measurements has to be made available to remote nodes in order to allow distributed measurements to be completed.

## 4.3 timing failure detector

### 4.3.1 Short description

In HIDENETS we consider several applications with timeliness requirements. This includes applications to whose quality of service requirements are specified in terms of temporal bounds or applications whose correctness and safety depend on the timely exchange of information. Because of that, it is necessary to monitor the timeliness of relevant *local activities* (involving the execution of local tasks) or *distributed*

*activities* (involving communication activities between different nodes), as a first step to detect timing failures and react to them.

The availability of a timing failure detector is hence very important in HIDENETS, since it can be used as a basic building block to observe and inform the applications or middleware services about the success or failure of *timed executions*. Moreover, it is also important to do such detection as timely as possible, as allowed by the synchronism of the infrastructure, in particular the one supporting the timeliness and trustworthiness oracles. The timeliness of detection is particularly important when prompt reaction to timing failures is necessary, for instance because of safety reasons. Note that because of the hybrid architecture assumed in HIDENETS, it is possible to enjoy better synchronism properties for the oracles than for the rest of the system. Therefore, not only the detection of the timing failures can be made in a more timely fashion, but also the reaction could be done more promptly and timely, if necessary, by implementing it in tight connection to the failure detection service, separately from the rest of the system.

In brief, this timing failure detection service works by “observing” the execution of single *timed actions*. A timed action is defined as follows. Given a reference real time instant  $t_A$  (start instant), an interval  $T_A$  and a termination event, a timed action is the execution of some operation, such that its termination event takes place within  $T_A$  from  $t_A$ . If the execution does not terminate within the specified bound, then a timing failure occurs and information about this timing failure is delivered to the interested application.

### 4.3.2 Targeting problem

Timing failure detection allows applications or middleware services relying on it to know if timed actions have been executed in a timely way or if they have incurred in a timing failure. In many works in the literature, failure detection refers to crash failures [13]. In HIDENETS we are especially interested in detecting timing failures, since these are the primary source of lateness, inconsistency or other more severe failures in the kind of uncertain operational environments that are considered. In both cases, the definition of failure detectors is made according to certain properties that these failure detectors should satisfy.

Typically, a failure detector is defined by stating its properties with respect to *completeness*, *accuracy* and *timeliness*. Completeness refers to the ability of a failure detector to detect every failure that occurs. In other words, a failure detector enjoying this property is not allowed to do false negatives. Accuracy refers to the ability of the failure detector to not make mistakes, that is, wrongly detect failures when they do not occur (false positives). Timeliness refers to the ability of the failure detector to detect failures within given time bounds.

There exists previous work on the definition of timely timing failure detection (TFD) services. In concrete, the work in [14] introduces the definition of a timely TFD that exhibits the following properties:

- *Timed completeness*: every timing failure will be detected within a maximum time interval ( $T_{TFDmax}$ ) from its occurrence.
- *Timed accuracy*: no correct timed action will be wrongly detected as a timing failure if it terminates before a specific time interval ( $T_{TFDmin}$ ) from the deadline.

From these definitions, it is clear that different instantiations of such a timely TFD can be considered. In one hand, the time bounds that characterise the failure detector depend on the concrete implementation. On the other hand, it is possible to manage some trade-offs, like the trade-off between the accuracy of the detection and the requirements imposed on the underlying infrastructure (in particular on the precision level of the duration measurement service), in order to obtain a service that fits better the requirements of a certain application.

In the case of HIDENETS, we have identified different applications with different requirements on the completeness, accuracy and timeliness of failure detection. For instance, timeliness of failure detection is particularly important in the platooning use case, for safety reasons, while it can be relaxed in applications where safety does not depend on timely reaction. Therefore, the timely TFD service to be used in HIDENETS should be able to address these different requirements.

An important aspect to take into account in the construction of the timely TFD service, which is very relevant for some applications, is how to ensure the timeliness of failure detection. Timeliness of detection is

important from a qualitative point of view, because it allows achieving a more precise notion of the state of the system at a certain moment of time. However, from a quantitative point of view it is important to ensure the timely detection of a failure when this failure has an impact on the safety of the system. For instance, in the hazard warnings and information from other vehicles application, a timing failure on the dissemination of a warning will make the application unaware of a potential danger (e.g., of an obstacle ahead), putting in risk the driver which believes, because the application says so, that there is no danger ahead.

The required timeliness of failure detection can be addressed, in the first place, because in HIDENETS we consider an architectural hybridisation approach, in which different parts of the systems can be built with different properties. This is also why this service is build as an oracle. On the other hand, in order to secure timeliness it is necessary to employ real-time techniques in the design and construction of the service. Such “real-time philosophy” will necessarily imply some restrictions on the availability of the service. This means that the service will have to be used parsimoniously, only when strictly necessary, for instance to detect the most critical timing failures. In the general case, when the timeliness of timing failure detection is not relevant, a different service should be used. For instance, one possibility is to use a service like the Timer Service defined within the SAF Application Interface Specification (see Section 6.2.11).

### **4.3.3 Relevant applications/use cases**

This building block is particularly relevant for platooning, and for some applications included in the assisted transportation use case, such as the hazard warnings between vehicles and the traffic sign extension applications.

### **4.3.4 Entities affected**

Timely timing failure detection will essentially be used for the timely detection of timing failures locally to a node (be it in the infrastructure or in the ad-hoc domain) and for the timely detection of timing failures in distributed activities involving nodes in the ad-hoc domain, within one-hop distance from each other.

### **4.3.5 Input from other building blocks**

The duration measurement routines provided by the duration measurement building block are necessary to build this timely timing failure detector module. Distributed detection requires information exchange among peer timing failure detection services. However, given the specific design requirements for this service, in particular concerning synchrony, communication activities must not be handled simply as another communication flow, but instead as a specific communication flow, through a dedicated communication channel. This issue is investigated in the scope of task 3.3.

### **4.3.6 Output to other building blocks**

This building block may be used by the reconfiguration and maintenance service, as well as directly by applications that need to take specific action upon the occurrence of a single timing failure.

### **4.3.7 What needs to be known to design the building block**

As described above, there exist previous solutions to the problem of timing failure detection, which will serve as reference work for the development of this building block. There are a number of questions still to be answered, in particular with respect to the specific interfaces that will be provided by this service. For instance, given the distributed nature intended for the service, it must be known which are the requirements in terms of distributed knowledge of failures, and how consistently should this information be provided to all interested processes. The development of specific solutions to address such kind of requirements will benefit from the knowledge of the network properties. Another aspect that also requires some knowledge about the underlying infrastructure concerns the possibility of attaching some timely execution functionalities to the timely timing failure detection. In fact, since the objective is to timely react to timing failures in some specific situations, it should be possible to attach these execution functionalities, which may depend on low level system mechanisms, such as schedulers and process communication services.

## 4.4 Freshness Detector

### 4.4.1 Short description

The Freshness Detector is the component able to check *temporal properties* related to freshness of connection-oriented data flows; it is a component able to detect violations of a particular class of real-time requirements.

Several distributed systems with real-time requirements need to use fresh information and, in addition, they need to detect the *freshness level* of available data. Message freshness detection is an instance of the more general problem of *failure detection* in which *timing failures* are considered (instead of more traditional class of crash failures, [13]). An example of a system for which it is necessary to check the freshness of the exchanged data is the C2CC (Car-to-Car-Communication) system. For instance, considering the Assisted Transportation use case, and the hazard warnings application in particular, we have requirements on the freshness of data received from the cars near ourselves. An old information (e.g. old data of a car position or velocity) that reaches the application level can bring to incorrect choices, that may eventually cause catastrophic failures. The level of freshness of the data received has to be assessed to guarantee the safety. In this perspective the Freshness Detector is a fundamental building block of the HIDENETS architecture.

Whereas the freshness level of a *single message* (i.e. a datagram message) can be obtained using the Timely Timing Failure Detector, the Freshness Detector can be used to instantiate a *logical connection* between two end-hosts with freshness requirements and to check the freshness of data exchanged on this connection. The connection can be unidirectional or bidirectional, and the predicates of freshness can be simple and complex. The Freshness Detector can also check violations of freshness of data exchanged between a group of processes (multicast and broadcast messages). The Freshness Detector is an end-to-end service: we can define complex freshness requirements, using logical predicates of simple requirements based on freshness as seen by single participants.

To note that different entities can have different real-time requirements: the Freshness Detector obtains these requirements directly from the involved entities. The same application or middleware component can require to the Freshness Detector to instantiate several data flows, and each dataflow can be defined with its specific freshness requirements. When the predicate on the freshness of the connection is not satisfied, the application has to be informed to allow proper reaction through its own fault management policy; therefore it is of uttermost importance to define the temporal behaviour of the service. The Freshness Detector must be able to inform the application of absence of fresh enough data in a timely way, respecting the real-time requirements of the application, which can be really strict. For example, let us suppose that an application, in order to guarantee the safety of the system, needs a data flow respecting a real-time requirement on the freshness of the data. This application can require to the Freshness Detector to instantiate a logical connection between the two end-hosts, and can define the predicate of freshness on this data flow. When the Freshness Detector detects the absence of fresh enough data on the flow, it has to react, informing the application, in a timely way (e.g.: respecting some predefined real-time limit). In other terms, from the application level prospective, it is always needed to be able to switch, in a limited and known time, to a fail-safe state.

### 4.4.2 Targeting problem

The detection of messages delayed more than a given threshold - called 'message freshness detection' - is an important requirement for many distributed critical real-time systems. The Freshness Detector is a specific software component to support the detection of the freshness of the exchanged messages. Freshness detection is an important task in several HIDENETS typical applications: this service detects if fresh-enough data are available, in order to allow the application/the middleware to react to the absence of fresh-enough information.

The Freshness Detector can evaluate the degree of freshness of the received data using the Distributed Duration Measurement service (Section 4.2): in particular, it is mainly used to evaluate the measure of the distributed interval between the *send* and the *receive* events for every single end-to-end one-way message. In this way it obtains an estimation of the freshness of every data received. To note that Distributed Measurement service uses the Reliable and Self-Aware Clock in order to evaluate these distributed measurements: using this service, the degree of precision of the measures can be directly estimated.

Distributed Duration Measurement gives also guarantees on the authenticity of the received data (the usage of this service prevents attacks on exchanged data).

When the Freshness Detection is used to inform the application on the absence of fresh-enough data, specific interfaces and real-time mechanisms must be used in order to guarantee a *prompt and timely reaction*. In HIDENETS this service is used also in safety-critical application: in the Assisted Transportation use case, for example, the absence of fresh-enough data, without a prompt reaction from the application, can bring to unsafe state for the system. It is thus important to obtain proper estimation of the freshness of available data (e.g.: for such applications, a pessimistic estimation, always greater or equal to the real freshness value, can be the proper one).

#### 4.4.3 Relevant applications/use cases

The Freshness Detector is relevant for all the applications in which we have timeliness constraints on received data and need temporal consistency. These applications can be summarised as follows (the applications are ordered with respect to the importance of the Freshness Detector in order to provide a dependable service):

- Traffic Sign Extension
- Assisted transportation
- Hazard warning between vehicles
- Unusual driver behaviour
- Interactive data
- Floating Car Data

The Freshness Detector is mainly relevant for the following use cases:

- Platooning
- Assisted transportation.

These use cases are relevant as the reference applications are based on the availability of fresh-enough data in order to provide a correct service. If no fresh-enough data are available, the application has to react in bound and known time in order to guarantee the safety of the application (e.g. both for the platooning and for assisted transportation, the absence of data about the position of the other participants have to be signalled and handled in a timely fashion).

#### 4.4.4 Entities affected

The check of the freshness is an end-to-end service, thus the affected physical entities are the two end-to-end hosts. The entities are thus the involved car(s) and/or server(s) in the infrastructure. No intermediate gateway/router is thus directly involved in the process of freshness detection.

#### 4.4.5 Input from other building blocks

The distributed measurements and their quality, in terms of an estimation of its precision (see Section 4.2), are obtained through the Distributed Duration Measurement service. The data obtained through this service are authenticated through the usage of Authentication service. In this way we prevent dangerous situations in which unauthenticated messages can give the wrong perception of the presence of fresh data.

#### 4.4.6 Output to other building blocks

Output of the Freshness Detector can be used directly by the applications and by other building blocks of the middleware:

- As an example in which the Freshness Detector is used by other higher level middleware services, we can think to the Diagnostic Manager. When this manager acts to obtain global view of a (sub)system state, it has to exchange messages about local state with other Diagnostic Manager in a distributed fashion. These messages must be fresh and their freshness level has to be controlled, in

order to not use too old data to build the local image of the state of the entire system. The freshness of the exchanged messages can be checked also in this case using the local Freshness Detector.

- As an example in which the Freshness Detector is used by other middleware, higher, services, we can think to the Diagnostic Manager. When this manager acts to obtain a distributed view of the global state of a (sub)system, it has to exchange messages about local state with other distributed Diagnostic Manager. This information must be fresh and its freshness level has to be controlled, in order to not use too old data to build the local image of the state of the entire system. The freshness of the exchanged messages can be checked also in this case using the local Freshness Detector.

#### 4.4.7 What needs to be known to design the building block

The problem of timeliness detection is quite well known and studied, and many solutions have been proposed in the literature [19][20][21][22]. Actually, despite the common root and the same abstract concepts, designing and implementing a safe and efficient solution for timeliness detection in a specific context like HIDENETS remains still a difficult and delicate task. In fact, the system and application constraints limit the solution space and the designer freedom, making known solution hard or even impossible to apply smoothly.

In the design of the Freshness Detector service, we use directly the information provided by the Distributed Duration Measurement service. The Freshness Detector service is used for safety-critical applications in HIDENETS; for this usage it will be important to design this building block in order to provide a *complete* detection of the timing failures in the reception of messages. As it is usual in the failure detection problem [13], a complete detection implies a detection that can be imprecise (e.g.: in our case, considering fresh-enough data too old by the Freshness Detector): since a too imprecise service becomes totally useless, we will have to find the point in which the service is complete enough to provide the safety, but also accurate enough to be profitably usable. Furthermore, the requirements of freshness of the applications must be deeply analyzed in order to define an interface of the Freshness Detector usable for definition of *complex predicates* on freshness of a data flow.

## 4.5 Authentication

### 4.5.1 Short description

The need for secure communication channels is common to several of the applications considered in HIDENETS. For instance, when retrieving information from a server in the infrastructure (e.g., to upgrade a software module in the car), it is fundamental to ensure that the correct server is being contacted, and therefore some kind of authentication is needed. The problem can in fact be solved using existing solutions, for instance using certificates managed by a public key infrastructure (PKI). Other techniques aimed at further securing the communication channels (e.g. SSL) can also be used and maybe incorporated with the help of this service.

Each car and server in the infrastructure can have a digital certificate issued and managed by a PKI. These certificates can then be used by the entities to authenticate each other, and establish communication channels with the necessary authentication guarantees.

The principle is that it is up to the applications to decide which technology or protocols they want to use to attain security properties in the communication. This service helps in this regard by allowing applications to retrieve something that helps proving that the entities are who they claim to be (i.e., their public-keys). The establishment of secure communication channels or the way the authentication is performed is a responsibility of the application. Some application may wish to use SSL to establish a secure channel, but other may prefer to use IPSec. This module should allow applications to maintain this flexibility by providing generic interfaces for setting up the secure communication channel.

The implementation of the authentication module as an oracle (by resorting to smartcard technology, for instance) is justified by the potential dependence of other oracles on the authentication service (e.g., Trust and Cooperation oracle). By providing this service as an oracle, it is ensured that the authentication module is kept safe from potential intrusions and, consequently, that the dependent oracles always receive correct authentication information about the identity of the entities. This way, the other modules can trust the information received by the authentication module, and continue to operate correctly, even if the entity is compromised.

### 4.5.2 Targeting problem

The ability to have authenticated and secure communication channels is a fundamental feature for several of the applications considered in HIDENETS. The authentication building block allows applications or middleware services to authenticate entities in a HIDENETS environment.

Considerable sensitive information is exchanged by the applications/use cases considered in HIDENETS. The corruption or leak of this information puts to risk the parties involved, and may lead to catastrophic consequences such as car accidents. The first step, provided by this service, is to guarantee that the communicating parties are who they really claim to be. If some car says it is A, then it has to show some proof that it is really A. The same is true for the servers in the infrastructure network. If some server says it is B, it has to present some proof to the car that it is B.

In practice, this can be accomplished by the use of digital certificates or similar technology. For instance, in the context of HIDENETS, it could be employed a public-key infrastructure (PKI) with a certification authority (CA) responsible for issuing X.509 certificates that bind public keys to entities. Assuming that every entity trusts this CA, and can verify its signature, then it can also verify that a certain public key belongs to any entity who claims so. Every component/entity that wishes to use this service should be provided in an absolute secure way with the public-key of the CA. One of the possible methods to achieve this is to hardcode the private-key in the component as part of the manufacturing process.

This service should provide a simple API with two simple calls that take as arguments a string that uniquely identifies an entity. Again, using the digital certificates example, the first call returns the entity's public-key, and it must proceed as follows:

- 1) Retrieve the associated X.509 certificate from the CA (suitable for C2IC), or the entity itself (suitable for both C2IC and C2CC).

- 2) Verify the signature in the certificate with the CA's public-key. This certifies that the certificate was really issued by the trusted CA.
- 3) Extract the entity's public-key from the certificate and deliver it to the application.

After the call is made, the application is then in possession of the public-key of the entity that wishes to authenticate. The public-key can then be used to verify the authenticity and integrity of any signed information that claims to be originated by the given entity.

The second call is similar to the first, but instead of returning just the public-key, it returns the entire X.509 certificate. The certificate can then be used by the application to use some standard protocol (e.g., SSL) to establish a secure communication channel with the entity.

### **4.5.3 Relevant applications/use cases**

This building block is implicitly relevant for several of the considered use cases. For example, in platooning it may be necessary to authenticate peer cars in a platoon or, at least, secure the authenticity of the information received from a platoon leader. Authentication is a fundamental service to address the requirements of the maintenance and software updates application, which needs secure channels with integrity properties to securely download the update.

### **4.5.4 Entities affected**

Both the cars and the servers in the infrastructure are affected by this building block. Depending on the nature of the application, the cars may need to authenticate the server in the infrastructure and establish secure communication channels. The infrastructure itself may also need to authenticate the car. For instance, if the car asks for some private information, the server in the infrastructure has to make sure the car is who it claims to be.

### **4.5.5 Input from other building blocks**

No input from other building blocks is required.

### **4.5.6 Output to other building blocks**

The authentication service will essentially be used by other oracle services. The idea is to use this service to authenticate other information provided by oracle services. For instance, this may include timestamps provided by the R&SA clock of freshness or timing failure indications provided respectively by the freshness and timing failure detection services. This service can also be used directly by applications or other middleware services.

### **4.5.7 What needs to be known to design the building block**

Computational resources of the entities involved, and the properties of the communication channels may be relevant knowledge that is needed to design this building block. The choice of cryptographic algorithms and parameters may need to be restricted based on this information.

The knowledge of the CA's public-key (or some equivalent secret in the case some other solution is applied) is needed by the entities who wish to use the service. This information must be distributed to the entities in an absolute secure manner.

## **4.6 Trust and cooperation oracle**

### **4.6.1 Short Description**

The trust and cooperation oracle (TCO) is a basic building block for cooperative services. A cooperative service emerges from the cooperation of entities that are generally unknown to one another. Therefore, these entities have no a priori trust relationship and may thus be reluctant to cooperate. In cooperative systems without cooperation incentives, entities tend to behave in a rational way in order to maximise their own benefit from the system. The goal of the trust and cooperation oracle is therefore to evaluate locally the level of trust of neighbouring entities and to manage cooperation incentives.

## 4.6.2 Targeting problem

Synergy is the desired positive effect of cooperation, i.e., that the accrued benefits are greater than the sum of the benefits that could be achieved without cooperation. However synergy can only be achieved if nodes do indeed cooperate rather than pursuing some individual short-term strategy, i.e. being rational. Therefore, cooperative systems need to have cooperation incentives and rationality disincentives. There are several approaches to this problem, some are based on micro-economy and others are based on trust. Typically, for micro-economic approaches, a node has to spend “money” for using a service and earns “money” for servicing other nodes. Regarding trust, one of the most common approaches is to use the notion of reputation, a level representing the level of trust that may be placed on a node, which can be computed locally by a single node, or collectively and transitively by a set of nodes.

## 4.6.3 Relevant applications/use cases

The relevant applications are expected to be the applications that implement a service by cooperation, e.g. peer-to-peer applications. In the case of HIDENETS, this is essentially the distributed black box application.

## 4.6.4 Entities affected

The following physical entities are affected:

- All cars involved in the cooperative application
- Gateway to the infrastructure

## 4.6.5 Input from other building blocks

The trust and cooperation oracle will use the proximity map service to discover the neighbouring nodes. The trust level and/or cooperation level of these neighbours will be evaluated.

## 4.6.6 Output to other building blocks

Depending on the particular option chosen for the implementation of the TCO, the output could be a list of nodes with a corresponding trust/cooperation level or simply a list of nodes that are considered trustable for cooperation. The Cooperative Backup Service will use the output produced by the TCO for establishing the partnerships during the backup.

## 4.6.7 What needs to be known to design the building block

In the literature, one can find many cooperation incentive schemes that are diverse not only in terms of the applications for which they are employed, but also in terms of the features they implement, the type of reward and punishment used, and their operation over time. A typical classification of cooperation enforcement mechanisms differentiates trust-based patterns from trade-based patterns. It makes a distinction between static trust that is about pre-established trustworthiness between peers, and dynamic trust that refers to reputation-based trust. In trade-based patterns, remuneration is the central notion and can be immediate, which is termed barter trade, or deferred, which is termed bond-based.

However, in our opinion, trust reflects the individual view of an entity about another entity’s trustworthiness. Whatever the incentive mechanism, an entity will have to ask itself whether it trusts another entity to cooperate with it. Trust establishment easily maps to reputation systems but may use remuneration systems as well. We thus prefer to classify cooperation mechanisms into: remuneration-based and reputation-based mechanisms.

### 4.6.7.1 Reputation systems

The estimation of reputation can be performed either centrally or in a distributed fashion. In a centralised reputation system, the central authority that collects information about peers typically derives a reputation score for every participant and makes all scores available online. In a distributed reputation system, there is no a central authority for submitting ratings or obtaining reputation scores of others. However there might be some kind of distributed stores where ratings can be submitted. Most of the time, in a distributed architecture, each peer estimates ratings autonomously. Each peer records ratings about its experiences with

other peers and/or tries to obtain ratings from other parties who have had experiences with a given target peer.

The centralised approach to reputation management is not fault-tolerant and furthermore not scalable. On the other hand, with the decentralised approach, it is often impossible or too costly to obtain cooperation evaluations resulting from all interactions with a given peer. This is the reason why reputation is generally based on a subset of these evaluations, usually obtained from the neighbourhood.

A reputation-based mechanism is composed of three phases:

1. *Collection of evidence*: Peer reputation is constructed based on the observation of the peer, experience with it, and/or recommendations from third parties. The semantics of the information collected can be described in terms of a specificity-generality dimension and a subjectivity-objectivity dimension.
2. *Cooperation decision*: Based on the collected information, a peer can make a decision whether it should cooperate with another peer. It will make its decision taking into account the other peer's reputation. There are various methods for computing peers' reputation: summation or average of the ratings, Bayesian methods, flow model based, etc.
3. *Cooperation evaluation*: The occurrence of interaction with a peer is conditional on the precedent phase. After interaction, a node must provide an evaluation of the degree of cooperation of the peer involved in the interaction. The reward is that the reputation of peers behaving cooperatively is raised accordingly. A peer with a bad reputation will be isolated from the functionality offered by the group of peers as a whole. The evaluation of the current interaction can convey extra information about other past interactions (piggybacking) that can be collected by the neighbouring peers.

This type of mechanisms has to cope with several problems due to node misbehaviour. Misbehaviour ranges from simple selfishness or lack of cooperation to active attacks aiming at denial of service (DoS), attacks to functionality (e.g., subversion of traffic), and attacks to the reputation system (liars). Counter-measures are to be designed to cope with these attacks.

#### 4.6.7.2 Remuneration systems

In contrast to reputation-based mechanisms, remuneration based incentives are an explicit counterpart for cooperation and provide a more immediate penalty to misconduct. Remuneration brings up requirements regarding the fair exchange of the service for some form of payment. This requirement generally translates to a more complex and costly implementation than for reputation mechanisms. In particular, remuneration based mechanisms require trusted third parties (TTP), such as banks, to administer remuneration of cooperative peers; these entities do not necessarily take part in the online service, but may be contacted in case of necessity to evaluate cooperation. Tamper proof hardware (TPH) like secure operating systems or smart cards have been suggested or used to enforce in a decentralised fashion the fair exchange of remuneration against a proof that the cooperative service was undertaken by a peer node.

A general-purpose remuneration-based mechanism comprises four main operations:

- *Negotiation*: The two peers often have to negotiate the terms of the interaction. Negotiating remuneration in exchange for an enhanced service confers a substantial flexibility to the mechanism. Negotiation can be performed either between participating peers or between peers and the authority.
- *Remuneration*: The remuneration can consist in virtual currency units (a number of points stored in a purse or counter) or real money (banking and micropayment), or bartering units (for instance quotas defining how a certain amount of resources provided by the service may be exchanged between entities). The latter can even be envisioned in the form of micropayments. Regarding real money, this solution assumes that every entity possesses a bank account, and that banks are enrolled in the cooperative system, directly or indirectly through some payment scheme. Issuing a check or making a transfer of money remunerates the collaborating peer.
- *Cooperation decision*: The peer in a self-organizing network is always the decision maker. During negotiation and based on its outcome, a peer can decide if it is better to cooperate or not.

- *Cooperation evaluation*: The interaction is controlled by a TTP, which can be centralised or distributed. In all cases, a conflict regarding negotiation, or remuneration is ultimately arbitrated by the TTP, even though this may possibly be a non-immediate process.

*Fair-exchange* is a desirable property of remuneration-based trust and cooperation mechanisms. A fair exchange should guarantee that at the end of the exchange, either each party has received what it expects to receive or no party has received anything. This property can only be attained by intricately integrating the remuneration operation with the application functionality. Fair exchange protocols rely on the availability of a trusted (and neutral) third party (TTP) caring for the correctness of the exchange. Two types of protocols should be distinguished: online protocols, which mediate every interaction through the TTP, which can lead to performance and reliability problems with the TTP constituting a bottleneck as well as a single point of failure; offline ones, also called optimistic fair exchange protocols, which resort to the TTP intermediation only if one of the parties wants to prove that the exchange was not fairly conducted.

#### 4.6.7.3 Challenges

As we just saw, there are a number of possible mechanisms to be considered for the implementation of a trust and cooperation oracle within HIDENETS. One of the next steps should thus be to analyze the specific requirements of the HIDENETS applications regarding cooperation and to choose the most appropriate approach between reputation or remuneration based mechanisms and their variants. Then, special care will have to be taken for evaluating the potential attacks, and counter-measures will have to be proposed and integrated to the implemented oracle.

#### 4.6.7.4 Specific requirements on infrastructure and low-level support

The TCO oracle will use remote calls to remote TCO oracle instances on other nodes. For identification of the nodes and/or for storing reputation or remuneration information, it could use specialised trusted hardware (such as a TPM-style platform) or a software-based solution possibly developed inside HIDENETS, such as the authentication service (as described in part 4.5).

## 5 Complex services

In this chapter we continue the description of the service blocks that we intend to address in HIDENETS, but now the focus is on complex services. This set of services is typically located at a higher level of the middleware, thus providing direct interfaces to the applications. Some of these services will need to handle specific application requests, for instance for replication or for quality of service monitoring and maintenance. Because of this direct interaction with the application, possibly implying the need to deal with high quantities of information, and also because the executed functions can be arbitrarily complex (e.g., requiring a lot of information to be gathered from other blocks and then possibly processing operations to be done on this data, such as with the diagnostic manager), these services should be constructed as ‘normal’ middleware services, on the less-trusted part of the system. Then, for the critical parts of their operation, they should use the services provided by the oracles previously described.

Similarly to the previous chapter, each of the following sections will shortly describe one of the complex services and will provide our initial view about the purposes, the interactions and the needs of each service. Along each section, an overview of the state of the art and related work is also provided.

### 5.1 Diagnostic Manager

#### 5.1.1 Short description

The Diagnostic Manager (DM) is in charge of managing all the activities necessary to judge if the system (or parts of it) is (are) working properly or not. The term “proper” has to be clarified: an unsuitable (improper) behaviour could be the result of: i) the manifestation of a fault affecting the component, which leads such component to depart from its functional specification, or ii) it could be determined by a change in the requirements of the application using the service provided by the component itself, or iii) by changes in the environment (e.g., system load) which leads to a change in the QoS provided.

The DM works on-line, basically performing the following activities:

- gathering data along time on the correct/deviated behaviours of the monitored components as perceived by appropriate error/deviation detection mechanisms the system is equipped with;
- diagnosing the status of the monitored components, on the basis of the collected information on the monitored component/subsystem. It is not practical, as soon as an error is observed, to declare the entire component failed and proceed to repair and replacement. Transients are the major causes of errors in hardware components, but they do not physically damage the component and they quickly disappear. Therefore traditional one-shot diagnosis is inadequate: an approach is needed, which collects streams of data about error symptoms and failure modes and filters them by observing component behaviour over-time. Typically, diagnosis algorithms are set up to discriminate acceptable from non-acceptable causes of malfunctions/misbehaviours of system components (e.g., transient from intermittent faults) adopting a threshold-based approach.

The DM judgements are necessary for the Reconfiguration Manager in order to select the proper system reconfigurations.

#### 5.1.2 Targeting problem

Diagnosis is a fundamental activity in fault tolerant systems: without the proper identification of faulty components and proper actions to handling with them, faulty components will increase along time, leading to defeat other fault tolerance provisions (e.g., error processing techniques) whichever be the redundancy level in place, ultimately causing (catastrophic) system failure. The HIDENETS framework addresses applications/scenarios with relevant dependability and QoS requirements, thus justifying the necessity to include a diagnostic manager among the middleware services.

Many solutions have been proposed in the literature for on-line diagnosis, mainly addressing the problem of discriminating between transient and permanent physical faults [27]. To this purpose, more or less sophisticated heuristics, based on thresholding schemes, e.g. [30], [31], [29] have been developed. They count errors, and when the count crosses a pre-set threshold a permanent fault is assumed. Another research

direction tackles the diagnosis problem in a probabilistic manner, with the construction of a formal framework based on well stated mathematical theories (Bayesian inference and Hidden Markov Models) [32].

The peculiar characteristics of the HIDENETS environment, like heterogeneity, mobility, dynamicity, as well as the fault categories addressed in the project will greatly impact on the diagnosis solutions to be devised.

Specifically, the HIDENETS environment is made up by two different categories of nodes:

- fixed nodes, which belong to the server infrastructure: they are wired interconnected and can have redundant resources;
- mobile nodes, which belong to “ad-hoc” domain: they are COTS devices provided with wireless capabilities in order to be able to connect to fixed nodes or to other mobile nodes.

Nodes inside the same category can be considered quite similar, but there are significant differences when comparing a fixed node with a mobile one:

- high vs. low amount of system resources: fixed nodes can do more and faster than mobile ones (they can benefit from redundant/replicated resources);
- wired vs. wireless connections: different speed and stability capabilities;
- fixed nodes are always in the system, mobile one are only sometimes in the system (they can have connection problems, empty battery, etc. or they simply can be intentionally turned-off);
- mobile devices should be reasonably cheap, to promote their diffusion.

On the basis of the specific node functionalities and capabilities, different fault classes could be identified, so it is necessary to develop specific diagnostic solutions able to assess the internal status of such different node (“local diagnosis”).

Moreover, HIDENETS system is highly dynamic and distributed, so each node interacts with a dynamic group of “reachable” nodes (mobile and fixed ones) along time. It is important for each node to be able to assess the status of nodes inside its group, because problems inside a node could negatively propagate into other nodes of the group. Some “global diagnosis” seems thus necessary, possibly organised so as to be resilient to Byzantine faults.

In order to evaluate the behaviour of a diagnosis mechanism, two metrics are defined:

- promptness: how quickly the diagnosis mechanism reveals the problem (given that there is the problem);
- accuracy: the ability to correctly discriminate each detected problem among all the envisioned ones.

Unfortunately, promoting promptness impairs accuracy and vice versa. In HIDENETS, diagnostic appliances will be set up to privilege one or the other of the above metrics (or a good balance of both), in accordance with specific requirements and available resources of the components under diagnosis (e.g., when the fixed infrastructure is involved, longer time can be taken by the distributed diagnosis to be highly accurate; on the contrary, in the ad-hoc domain, promptness of a distributed diagnostic judgement seems more relevant, because of the possibly high mobility of the involved components).

### 5.1.3 Relevant applications/use cases

The DM, being part of the Fault Tolerance management, is useful for all applications and use cases where resilience requirements are relevant.

For example, considering the “Assisted Transportation” use case, some applications like “Traffic Sign Extension” or “Hazard Warnings” require that faulty devices inside a traffic light or onboard of an emergency vehicle do not generate traffic or hazard signals when it is not needed. In the “Car Accident” use case, in order to guarantee to the “Access Medical Expertise” application an available and reliable communication channel between the emergency vehicle and the hospital, it is appropriate to diagnose all the

available communication resources in order to select the most appropriate one for the emergency requirements.

#### 5.1.4 Entities affected

The DM may affect many system entities, spanning from servers inside the infrastructure to software components/services inside a device in the “ad-hoc” domain. Of course, diagnostic solutions need to be adapted to the specific characteristics of the system entities it applies to.

#### 5.1.5 Input from other building blocks

To accomplish its task with reference to a monitored system component, the DM mainly needs to acquire the following information regarding the monitored component: i) error/deviation information from error/deviation detector, ii) failures to provide requested QoS.

DM may also receive directives from the application level (e.g., the application level could ask the DM to take decisions on the basis of specific requirements, e.g. favouring accuracy despite of promptness).

Therefore, inputs are required from the following modules:

- “Error Detection”, in order to receive specific error signals coming from various components in the node. 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. “In-stack monitoring & error detection” developed in WP3 constitutes a primary source of error detection.
- “Performance Monitoring” (developed in WP3), in order to get information about specific services in the node;
- “Traffic Failure Detection” (developed in WP3);
- “QoS Coverage Manager”, in order to get information about the QoS policies inside the node;
- services from the Timeliness and Trustworthiness oracles, in order to be able to diagnose time faults or attacks.

Specific faults and related criticality have to be defined in order to better focus the diagnosis mechanisms. 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 they are present, and to avoid detecting a non-existing problem).

#### 5.1.6 Output to other building blocks

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

- the “Reconfiguration Manager”, which has to properly reconfigure the system selecting the reconfiguration strategy more suitable 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 device soft reset;
- the application level, which may need to be informed about changes in system state. Example: during the execution of an application, some external software components could be diagnosed as corrupted, thus forcing the application to notify the inability to perform some requested actions;
- the maintenance centre, to take the appropriate actions to physically isolate and repair the component diagnosed as faulty;
- the “QoS Coverage Manager”, which has to perform some probabilistic analysis basing on diagnostic data and performance historical data.

### 5.1.7 What needs to be known to design the building block

The design phase of the DM needs to know all the following:

- basic characteristics of the components to which diagnosis has to be applied, mainly its fault/deviation model on the basis of which a diagnostic judgement is issued;
- the fault model of the environment in which the diagnostic mechanism has to operate. This information is necessary to understand which “protection” provisions the diagnosis has to take to be a resilient component itself;
- error/deviation mechanism capabilities and specifications (emission rates of false positives and of missed negatives), in order to know when error/deviation messages are signalled and how much they are trustworthy;
- available resources and computational capabilities of devices that need diagnosis capabilities, in order to select the proper diagnosis mechanisms.

## 5.2 Reconfiguration Manager

### 5.2.1 Short description

The “Reconfiguration Manager” (RM) component is part of the “Fault Tolerance Manager” and it is specifically in charge of deciding both when some reconfiguration is needed and which reconfiguration policy has to be applied. The RM, on the basis of information coming from DM and from “QoS and Differentiation Manager”, has 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 after the occurrence of some deviations from expected QoS;
- also, decide the application of pre-defined preventive maintenance policies, possibly alerting the operator service in order to physically repair/replace modules which fail to satisfy the maintenance tests and, in such a case, implementing the appropriate reconfiguration of the involved subsystem.

The RM component works on-line, basically performing the following activities:

- gathering information about the status of the monitored components from the “Diagnostic Manager”;
- gathering information about the required QoS levels from the “QoS Coverage Manager”;
- selecting the proper reconfiguration of components/system (if diagnosis or QoS adaptation determines that it is necessary); the choice of the reconfiguration action is guided by the expected benefit of applying it, obtained through a quantitative evaluation support implemented as part of WP4;
- triggering the proper actuators to put in place the selected reconfiguration;
- triggering efficient preventive maintenance operations, again decided on the basis of an evaluation support to compare several possible alternatives (as suggested in [34] and [35]), and implemented as part of WP4.

### 5.2.2 Targeting problem

Reconfiguration is a fundamental activity in fault tolerant systems, because dynamically reacting to diagnosed problems by performing proper system reconfiguration increases system availability and dependability; reconfiguration activities are particularly relevant in the HIDENETS scenario, where the high dynamicity of the environment requires frequent arrangements of the available resources.

Reconfiguration may be triggered as a reaction to either the presence of diagnosed faults or to diagnosed changes in the environment, therefore reconfiguration effectiveness is primarily influenced by correct diagnosis and secondly by the selection of the proper reconfiguration. Deciding the appropriate reconfiguration policy among several alternatives requires some evaluation support to assess the relative

benefit of each of them, with respect to defined criteria (e.g., with respect to the MTTF measure, or to the time necessary to perform the reconfiguration, or to some more complex performability measure). To better fit with the specificities of the system conditions at the time the reconfiguration is to be applied, the evaluation support should be applied on-line.

Reconfiguration can be performed according to different approaches; the extremes are described in the following and intermediate solutions are possible as well:

- static approach: a set of reconfiguration strategies are defined at system design time, and statically associated to specific patterns of faults/deviations of a number of system components. This association is performed through a look-up table, which is accessed on-line to retrieve the appropriate reconfiguration strategy. The problem arises of how to sort out from the situation where the fault/deviation pattern, as assessed by the DM subsystem, is not listed in the look-up table (situation not foreseen beforehand).
- dynamic approach: many strategies are applicable for the same diagnosed scenario, the choice of the reconfiguration to apply is performed on line through a proper evaluation support, which is fed with the specific system and environment conditions at the time the reconfiguration action is triggered by the DM subsystem. This is of course more accurate than the previous approach, but more costly in terms of time and resources needed to perform the on-line choice. Evaluation relying on the model-based approach looks a promising direction to this purpose; however, it requires methodological advancements to cope with complexity and fast solution methods, to make the approach feasible in practice. So, although very appealing, the dynamic selection still requires deep research investigations. Its possible implementation will be investigated in HIDENETS, with the on-line evaluator support realised as an activity of WP4.

In complex distributed systems reconfiguration is typically approached following a hierarchical approach [33]. This seems appropriate also in the context of HIDENETS, where node reconfiguration could be organised at two levels:

- reconfiguration local to a node, in order to either resist to local diagnosed faults or to better exploit local available resources;
- reconfiguration at multi-node level, to better manage faults and resources at system level. Global reconfiguration, possibly performed inside a group of both fixed and mobile nodes, promotes efficient reorganisation of system resources, taking into account higher level information on the whole involved group of components, thus overcoming the restricted vision at the single node.

The reconfiguration activity is expected to involve also the underlying communication levels, thus requiring the interaction with the “Communication Adaptation Manager”. However, it has to be reminded that the “Communication Adaptation Manager” is in charge of managing the communications for optimisation or local problems, without being necessarily triggered by the middleware level “Reconfiguration Manager”. This observation is at the basis of the existence of both services in the HIDENETS architecture.

### 5.2.3 Relevant applications/use cases

The RM is part of the Fault Tolerance management, so it is useful for all applications and use cases where resilience requirements are relevant.

For example, in the “Car Accident” use case, in order to guarantee to the “Emergency Communication” application an available communication channel between the emergency vehicle and the hospital, it is appropriate to reconfigure the system in order to give to the application the sufficient amount of system resources, possibly depriving other applications of some resources. In the “Car Accident” use case, the “Black Box” application needs to properly reconfigure system resources in order to send to fixed nodes as much data as possible, possibly avoiding indirect connections between a mobile node and a fixed one.

## 5.2.4 Entities affected

The RM subsystem may affect both physical system entities (fixed or mobile nodes, physical devices inside nodes, etc.) and software components/services, depending on the criticality and the appropriateness of applying reconfiguration actions. Obviously reconfiguration mechanisms need to be adapted to the specific characteristics of system entities it applies to.

## 5.2.5 Input from other building blocks

The RM needs input from the following modules:

- “Diagnostic Manager”, in order to get diagnostic information about the status of modules/services on which reconfigurations can be made;
- “Performance Monitoring”, in order to get information about specific services in the node;
- “QoS and differentiation Manager”, in order to request new QoS configurations inside the node;
- “Communication Adaptation Manager”, in order to know what is done and what could be done about communications;
- (possibly) services from the Timeliness and Trustworthiness oracles, in order to be able to reconfigure within specific temporal bounds;
- inputs from the evaluator support, in case of on-line evaluation of the reward associated to each doable reconfiguration strategy.

## 5.2.6 Output to other building blocks

The output of the RM is taken by the actuators the system is equipped with, in order to put in place the selected reconfiguration strategy (reconfiguration activities involving communication matters are performed by the “Communication Adaptation Manager” module, thus requiring some interaction with it). Output is also expected to be redirected to the operator service, which assists the physical repair/replacement of modules diagnosed as faulty.

## 5.2.7 What needs to be known to design the building block

In order to design the RM it is necessary to define the following elements:

- the system topology, including the relationship among components/services, availability of replicated resources, and how to tune components and services involved in the reconfiguration strategies. These are necessary elements to perform reconfiguration actions;
- which reconfiguration strategies are appropriate for each envisioned diagnosed scenario, possibly giving some indications about cost and effectiveness of each reconfiguration, in order to have more elements to make the choice;
- the preventive maintenance actions, from which RM will decide a convenient and efficient composition of them to define a preventive maintenance strategy;
- criteria (metrics of interest) to base reconfiguration and preventive maintenance selection on.

## 5.3 QoS coverage manager

### 5.3.1 Short description

The specific quality of service (QoS) requirement of an application must typically be translated into system tractable requirements. QoS requirements usually concern various observable or measurable communication parameters, like message delays, jitter, packet loss or bandwidth. On the other hand, in open communication environments, such networks parameters tend to vary with time, and it is often difficult to ensure a constant QoS over periods of time. Therefore, QoS contracts or Service Level Agreements (SLAs) should be able to reflect and encompass this uncertainty, which can be done, for instance, by associating a “probability of satisfaction”, or a “coverage” value, to given communication parameters. This opens new possibilities for

managing variations on the available QoS. Applications may rely on the provision of guaranteed “coverage” instead of (more difficult to satisfy) guaranteed bounds.

The QoS coverage manager 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 do that, the QoS coverage manager will need information concerning the performance of the network and, in particular, it will need to use histories of measured parameters to perform statistical calculations and characterise the probability distribution functions of these parameters. In addition, other information concerning the state and configuration of the system and the networks may be used to determine the possible adaptation alternatives.

The specific probabilistic approaches to be used by the QoS coverage manager, on which depends the dependable estimation of the coverage provided to applications, will be subject of investigation in HIDENETS. The basic idea is to collect performance and network information mapped in timing variables to perform probabilistic analysis in order to build probabilistic distribution function *pdf* that represents the actual distribution of a timing variable.

QoS requirements of applications or other services are specified to the QoS coverage manager through a pair  $\langle bound, coverage \rangle$ , where *bound* is a time bound and *coverage* is the probability of this bound to be secured. In order to manage the requests, the QoS coverage manager will encompass two main internal activities: a QoS coverage evaluation activity, and a QoS coverage maintenance and degradation activity. The former will be devoted to applying the monitoring strategies and output probabilistic information characterizing the state of the environment. The latter will be in charge of analyzing this information, verifying if application requests are satisfied, and provide the necessary indications of QoS variations when needed.

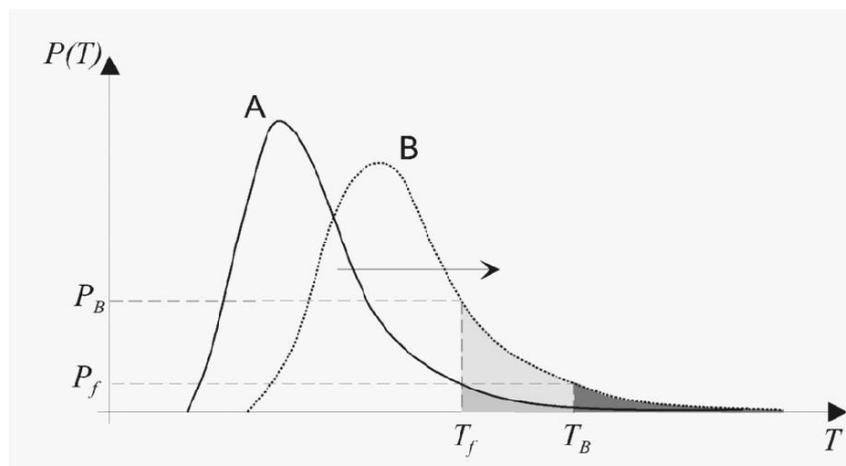
### 5.3.2 Targeting problem

The 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. Classical approaches for QoS provision assume that resource reservation is possible [57][58][59]. 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.

There are several QoS mechanisms that may be considered when dealing with QoS related architectures [60]. However, the QoS coverage manager is focused essentially on monitoring and maintenance aspects, with a design that takes into account the uncertain behaviour of the environment.

Dependable monitoring information about the timeliness of the environment could possibly be used as input to other QoS related mechanisms, thus improving their functionality. For instance, admission testing mechanisms need to compare available resources with requested ones, and so they could possibly use information provided by the QoS coverage monitoring mechanisms to improve the knowledge about available resources, measured in terms of timeliness.

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 these 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 rigorous information provided by some underlying 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.



**Figure 4: Example of variation of distribution with the changing environment**

The concept of dependable adaptation has been presented in previous work [15], 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 on a modification of the calculated pdf. This is illustrated in Figure 4, which depicts two distinct intervals of observation of a timing variable. While in a first moment, characterised by pdf A, the cumulative probability of violating time bound  $T_f$  is given by  $P_f$ , when the state degrades and the distribution changes, being characterised by pdf B, the same time bound could be violated with a higher probability, given by  $P_B$ . The objective of adaptation is then to choose another 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  $T_B$ .

This approach is however general and can be applied in HIDENETS. However, there are a number of open issues, as pointed out in [15], that still need to be addressed in order to obtain practical solutions. In particular, in HIDENETS it will be necessary to understand the behaviour of the applications (typical intervals of operation) and environments over which they execute (specific characteristics that may be used for the probabilistic characterisation). Furthermore, there are also several open issues with respect to the specific probabilistic approaches that should be used, which will be the subject of further work in the context of WP2 and WP4.

### 5.3.3 Relevant applications/use cases

Most relevant applications are all those that are able to execute correctly despite occasional faults (message losses or late messages), provided that a certain coverage of the assumed bounds is guaranteed, even if this implies assuming new bounds when necessary. For instance, multimedia applications like Video Conference, On-line Games, and Audio and video streaming, are able to dynamically adapt some characteristics of the transmitted data, as a means to secure some required level of coverage (a bounded probability of failure). In the assisted transportation use case, some applications have temporal consistency requirements. It may be possible to construct these applications by assuming that the temporal consistency requirements are secured with a certain probability (coverage), which should remain stable during operation. In this case, the QoS coverage manager can be used to keep track of these requirements and inform the application when they can no longer be secured. Then, the applications would need to adapt its execution to use new timing bounds and keep the coverage. For instance, in the traffic flow control application, a change in a timing bound would be reflected on the time taken to provide updated information (but which would be provided with the same accuracy).

### 5.3.4 Entities affected

The service can be made available to entities both in the ad-hoc domain (cars) and in the infrastructure (servers).

### 5.3.5 Input from other building blocks

The QoS coverage manager will basically interact with the diagnostic manager in order to use diagnostic data and performance historical data that is needed for probabilistic analysis.

### 5.3.6 Output to other building blocks

This building block will basically provide a service to applications.

### 5.3.7 What needs to be known to design the building block

In order to build the QoS coverage manager service it is necessary to have a clear view about a number of aspects, but in particular about:

- The characteristics of the operational environments (e.g., types of networks, possible interferences) on which the service will be used
- The characteristics of the applications that will operate and use the service (e.g. traffic generation patterns, adaptation abilities)

## 5.4 Replication manager

### 5.4.1 Short description

Most often when high availability is achieved through service replication and therefore also state sharing, the state sharing is done on dedicated hardware (fixed network interconnect)[16]. Moving highly available services into the wireless ad-hoc domain means that the state sharing will be achieved using the wireless interfaces. In order to make the state sharing more adaptable the fixed preconfigured cluster will be made dynamic so that members of the cluster will be selected as opportunities present themselves.

The replication manager is a tool that is used to take care of state sharing between replicated stateful services. A stateful service is one where the server needs to keep a state of the ongoing communication with its clients as opposed to a service where the server receives all the information that is required to perform its task with each incoming request. The main focus of the replication manager is on providing state replication for services provided by nodes in the ad-hoc domain. There is functionality in the replication manager to do sophisticated selection of replica candidates based on measured network metrics and replica selection policies defined for each application. The replication manager makes use of a distributed cluster formation algorithm which will select replica candidates so that the state replication can be done satisfying the desired replication scheme. Examples of such replication schemes can be to spread the replicas over a large area or only using one hop neighbours for state backup target. The replication manager differs from the cooperative data backup (see Section 5.8) in the way that it places the whole copy of the service state on a small set of backup nodes that can potentially be used to continue a service in case of a failure. Whereas the cooperative data backup is spreading small parts of the collected data onto a larger number of nodes in order to be able to reassemble the data at a later stage.

### 5.4.2 Targeting problem

The scenario where the replication manager will be useful is in the ad-hoc domain. It is intended to be used with a service provided in the ad-hoc domain by other ad-hoc nodes. Each car is running a set of services. Some of these services are stateful and need to share their states with failover candidate services in other cars to gain dependability. The replication manager is controlling the division of dynamically formed state sharing groups of servers into smaller subsets of these groups for performance enhancement. State update subsets are small groups of servers that share the same state. The motivation for having this component is to manage state sharing subsets which will reduce state update overhead and reduce risks of inconsistency. The replication manager itself is a distributed process. It is responsible for dynamically reconfiguring the subset

memberships of the individual servers. The replication manager will inform the servers of their new subset memberships and the servers will then start sharing state updates.

The replication manager will contribute to minimisation of state inconsistency and improved dependability.

### **5.4.3 Relevant applications/use cases**

The replication manager is developed focusing on the requirements from services provided by nodes in the ad-hoc domain. Further investigations will reveal if it is advantageous. The replication manager is relevant with client-server applications where the server is replicated. This holds not only for the ad-hoc domain but also for the infra structure domain. The replication manager is relevant for assisted transportation (floating car data). Moreover the replication manager can be used by stateful gateway nodes in the ad-hoc domain to share their state with backup gateway nodes.

### **5.4.4 Entities affected**

The entities that are affected by the replication manager are the nodes where the state replication mechanisms (e.g. state-sharing algorithm, dissemination protocol) are deployed since decisions inferred by the replication manager impact how the state replicas are disseminated.

### **5.4.5 Input from other building blocks**

This component requires input from the network context repository [18] and the proximity map. The replication manager will need positioning input from a localisation device (GPS).

### **5.4.6 Output to other building blocks**

The distributed algorithms will send signals to dynamically negotiate and update state-sharing subgroups membership. The replication manager sends group membership updates to the fault tolerance manager and reconfiguration manager for fail-over purposes.

### **5.4.7 What needs to be known to design the building block**

Knowledge of the neighbouring nodes both in the network and the nodes in close physical proximity is needed in order to setup a local replica group for service replication. The service replication groups are created on the fly according to the requirements of the application. The requirements of the individual applications are specified at deployment time and the subset creation policies are adapted accordingly also at deployment time. The subsets are formed dynamically as the topology changes. The current state of the art within service replication within dynamically maintained groups is e.g. Reliable Server Pooling[63] which was developed for dynamically changing group memberships of servers in fixed networks..

### **5.4.8 Items for discussion**

The parts that we are interested in are the group member selection part. We are going to develop distributed algorithms to form backup clusters for service replication. The actual state replication algorithm will be very simple.

## **5.5 Mobile agent manager**

### **5.5.1 Short description**

A mobile agent is a piece of software that traverses in the network. On one hand, it gets local information from the nodes it is visiting and makes decisions about the next step for its own activity. On the other hand, it may affect the visited node by leaving information or trigger local activities. Mobile Agent Manager is a software component supporting the execution of a mobile agent. The main functionalities of the mobile agent manager can be summarised as follows:

- Providing interfaces to the mobile agent for retrieving/editing local information, and for trigger local activities.
- Mobile agent generator.

- Mobile agent failure detection: a mobile agent may die out due to communication error, network separation, the mobile agent failure need to be detected (possibly in a hop by hop manner).
- Mobile agent failure recovery: a node detecting a mobile agent failure should be able to recover the agent by generating a replicate of the failed one.
- Mobile agent terminator: determine when the agent should be terminated, then terminate it simply by discarding it.

### 5.5.2 Targeting problem

Mobile agent manager is a specific software component to support the execution of mobile agents.

### 5.5.3 Relevant applications/use cases

The mobile agent is relevant for the following applications/use cases:

- Car accident, Assisted transportation,
- Restricted ad-hoc networks with mobile proxy,
- Limited broadcast query to a public ad-hoc networks.

These use cases are relevant as they share the same property: the destination is known not in terms of address, but in terms of property. For example, in the case of limited broadcast taxi query to a public ad-hoc network, the destination can be any taxi with certain properties (e.g., number of seats).

### 5.5.4 Entities affected

The following physical entities are affected:

- All cars involved in the mobile agent propagation
- Gateway to the infrastructure

Although a mobile agent may traverse in different networks (ad-hoc and infrastructure networks), the entity that need to have the mobile agent management component are limited to the nodes in the ad-hoc networks and the gateway to the infrastructure (e.g., access points). As for the other entities in the infrastructure network, the mobile agent can be transmitted as normal data packet.

### 5.5.5 Input from other building blocks

Although not all the functionalities are required in all nodes that support the execution of mobile agent, all such nodes are required to be able to provide the necessary information for the mobile agent to make decisions. This information can be obtained from the network context repository described in [18].

### 5.5.6 Output to other building blocks

As output to other components, a mobile agent may leave information gathered along the journey, or trigger some components. As an example for trigger, when a mobile agent for post-accident warning reaches the desired warning area and finds an access point, it may terminate itself and trigger the access point to broadcast warning message as broadcasting by the access point is more efficient. It is quite difficult to tell what are the building blocks to which a mobile agent may output, as it depends on the design the mobile agent (e.g., how light the agent should be?), the security issues (what type of activities can a mobile agent trigger), and so on. These issues will become clearer during the design phase of this service.

### 5.5.7 What needs to be known to design the building block

The use of mobile agents is increasingly explored by an expanding industry, and the first commercial systems (e.g., Aglests, Voyager, Concordia) are now available. Most of these systems are based on Java for the programming of agents.

The most critical issue about the realisation of mobile agent techniques is probably security concerns: protecting hosts from malicious agents as well as protecting agents and agent-based applications from malicious hosts.

To design a mobile agent, the following information is needed to be known:

- Purpose of the mobile agent
- Requirement of the application (QoS, dependability, security)

One example is a mobile agent for post-accident warning, which is required to reach the area 400 meters from the accident location within 2 seconds. Another example could be a taxi query to a public ad-hoc network, in which a user can specify the preference of the desired taxi, maximum time to wait, and so on.

## **5.6 Inconsistency estimation module**

### **5.6.1 Short description**

Stateful applications are applications that manage states about ongoing communications. Each application requires a specific type(s) of logged information – SIP session state, interactive game checkpointing, etc. – which depends on what the logs are used for: billing-related logs store different information about the communications than logs used for SIP server failover situations.

For dependability purposes, replicated stateful servers have to share service state so that if the primary server fails, the service can be continued at a backup server without losing all information related to previous communications. This information needs to be propagated timely so that the service can be continued at a backup server with up-to-date, or consistent, state values. If the state value is inconsistent when the state is read, dependability levels might be impacted because this can lead to, e.g., dropping ongoing SIP sessions (inconsistent session state) or revenue losses for the operator (inconsistent charging state).

The goal of the inconsistency evaluation module is to quantitatively estimate inconsistency for the replicated, stateful, HIDENETS services – possibly for a given replicated server and at any given time. Several inconsistency evaluation frameworks have been proposed for the infrastructure domain, among which are [16] and [17]. The main foreseen contribution will be the extension/adaptation of those frameworks (or the creation of a new framework if needed) for HIDENETS services in the ad-hoc domain.

### **5.6.2 Targeting problem**

The inconsistency estimation module helps determine whether the replicated service at a server is up-to-date (at a given time). The knowledge about inconsistency levels at a given time can be used by other HIDENETS services (e.g. replication manager) to failover the service more efficiently by picking an optimal replicated server or, typically in the distributed black box scenario, to retrieve a consistent image of the road and traffic conditions at accident time.

### **5.6.3 Relevant applications/use cases**

This component is relevant to all use cases with replicated stateful services that can use the knowledge about inconsistency levels to optimise performance and/or dependability. Typically, those services are deployed in the infotainment use-case.

### **5.6.4 Entities affected**

It is not clear yet how, and therefore where, this module is expected to operate. Most likely, inconsistency will be estimated, and therefore implemented, at the replicated servers (ad-hoc and infrastructure domains) and at the replication manager, which uses this information to pick the appropriate backup server for failover or state retrieval in general (c.f. current replication manager description for illustration). In the RSerPool resilient architecture, such an entity is called Name Server [18].

### **5.6.5 Input from other building blocks**

Inconsistency levels greatly depend on network characteristics and traffic models [16][17]; therefore, information provided by the network context repository [18] is essential in the inconsistency estimation process.

### 5.6.6 Output to other building blocks

The inconsistency levels estimated by this component can be used by the fault tolerance manager, and more specifically the reconfiguration manager subcomponent, to choose among the replicated servers and pick one as a backup only if it has consistent service state or if the probability for the service state to be consistent is above a certain threshold. The same applies with the cooperative data backup module.

The replication manager can also use inconsistency as an input to logically split a large set of replicated servers into subsets.

### 5.6.7 What needs to be known to design the building block

In order to evaluate inconsistency properly, we first need to carefully define what inconsistency is. There are many definitions of inconsistency, because each stateful application has different requirements in terms of state replication and state retrieval. For example, for one application those requirements imply that inconsistency is the probability that all states are identical at a given time, while another application might require that only the state replica used by a read process is consistent with the expected state value (i.e. the value read is not identical to the value written when the last state update was committed). There exist a huge number of replication techniques available (e.g., c.f. literature about consensus algorithms, concurrency and commitment protocols, and database technologies) that can be used in all types of application scenarios. Therefore, we need those specific requirements in order to define, and then evaluate, inconsistency for a given application.

Also, we need some knowledge about the protocol stacks used in each use-case considered and the information relevant to the fault tolerance manager so that it can react optimally (depends on the (set of) solution(s) picked in the scope of HIDENETS).

## 5.7 Proximity map

### 5.7.1 Short description

The goal of the proximity map service is to provide a map of neighbouring<sup>1</sup> nodes (along with estimated distance, position and speed). Thus, the proximity map represents the local knowledge a node has about its vicinity. This can vary according to wideness (the number of communication hops represented on the map) and according to accuracy (how often is the map updated? Does it use a reactive or a proactive protocol?).

### 5.7.2 Targeting problem

For many applications, and especially for cooperation-based applications, a node needs to interact with its neighbours. Furthermore, the quality of service that may be provided by a given component can vary according to the vicinity, e.g. the quantity of neighbours, their density, etc. It is then necessary to formalise this view of the vicinity into a proximity map.

### 5.7.3 Relevant applications/use cases

This service is relevant to the distributed black box. It should also be relevant to all applications that run in the ad-hoc domain.

### 5.7.4 Entities affected

The proximity map needs to interact with all the nodes in the vicinity and potentially with some servers in the infrastructure domain.

---

<sup>1</sup> The notion of neighbourhood here refers to geographic vicinity and not networking vicinity.

### 5.7.5 Input from other building blocks

The proximity map service is a rather low-level service and thus will essentially use information from the link and/or network layers. If positioning information is made available by some block, then it might also be used as input.

### 5.7.6 Output to other building blocks

The proximity map service will provide a position and an evaluated accuracy for each neighbour node, according to a given distance specified by the calling building block.

### 5.7.7 What needs to be known to design the building block

This building block will provide an abstraction of the neighbouring attainable network of entities. Thus it will be highly dependent on the network technology.

For ad-hoc networks, proposed neighbour discovery protocols are used to provide routing in such networks, and thus 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 proximity map evolves only when a particular route is no longer available.

In the HIDENETS context, the mobility model is geographically constrained when compared to other ad-hoc scenarios: most neighbours will be either running in the same direction or in the opposite direction. The proximity map will thus provide two separate lists of neighbours, a short-term interaction list (cars running in the opposite direction) and a long-term interaction list (cars running in the same direction). In a more general setting, it will provide a list of attainable neighbours with information on their position and relative speed.

#### 5.7.7.1 Challenges

Approximating the distance, position and speed of a discovered neighbour is a challenging task. A possible yet approximate solution would be to measure transmission power/energy needed to communicate between nodes at regular intervals.

Another possibility is to couple the node discovery algorithm with a positioning device (e.g. a GPS receiver). It is worth noting that these two approaches may be combined to improve accuracy of measurements and to cope with situations where one or the other approach cannot be used, like when GPS satellites are not available.

#### 5.7.7.2 Specific requirements on infrastructure and low-level support

In the context of car-to-car communication, we plan to use only one-hop communication (point-to-point, without routing) over a broadcast-type communication medium.

We will need to have access to low-level parameters of the communication device to evaluate the distance and the direction of a discovered peer.

We could also need to have access to a positioning device such as a GPS receiver.

## 5.8 Cooperative data backup

### 5.8.1 Short description

The problem of cooperative backup of critical data consists essentially in: discovering storage resources in the vicinity (using the proximity map service), negotiating a contract with the neighbouring cars for the use of their resources (using the trust and cooperation oracle), handling a set 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.

### 5.8.2 Targeting problem

The goal of the cooperative data backup service is to backup critical data despite failures of the data owner and of the nodes storing the critical data for the data owner. The service also needs to be resilient to denial-of-service attacks and other type of non-cooperation attacks such as data retention.

### 5.8.3 Relevant applications/use cases

The most relevant application within HIDENETS is the distributed black box.

### 5.8.4 Entities affected

The cooperating nodes, i.e. in our context the cars.

### 5.8.5 Input from other building blocks

The cooperative data backup will heavily use the proximity map to discover the nodes with which it can cooperate, and the trust and cooperation oracle to assess confidence in those nodes.

### 5.8.6 Output to other building blocks

The cooperative data backup can be seen by other building blocks as a safe repository for critical data.

### 5.8.7 What needs to be known to design the building block

Cooperative backup are inspired by both cooperative file systems and file sharing systems. Most are concerned with the problem of cooperative backup for fixed nodes with a permanent Internet connection. To our knowledge, there are only two projects looking at backup for portable devices with only intermittent access to the Internet: *FlashBack* [36] and *MoSAIC* [52][53].

#### 5.8.7.1 Peer-to-peer Backup Systems for WANs/LANs

The earliest work describing a backup system between peers is the one of Elnikety *et al.* [37], referred as Cooperative Backup System (*CBS*). Regarding the functions of a backup system (resource localisation, data redundancy, data restoration), this system is quite simple. First, a centralised server is used to find partners. Second, incremental backup, resource preservation, performance optimisation were not addressed. However, various types of attacks against the system are described. We will come back to this later.

The *Pastiche* [37] system and its follow-up *Samsara* [39], are more complete. The resource discovery, storage, data localisation mechanisms that are proposed are totally decentralised. Each newcomer chooses a set of partners based on various criteria, such as communication latency, and then deals directly with them. There are mechanisms to minimise the amount of data exchange during subsequent backups. *Samsara* also tries to deal with the fair exchange problem and to be resilient to denial-of-service attacks.

Other projects try to solve some limitations of the *Pastiche/Samsara* systems, or to propose some simpler alternatives. This is the case for *Venti-DHash* [40] for instance, based on the *Venti* archival system [41] of the *Plan 9* operating system. Whereas *Pastiche* selects at startup a limited set of partners, *Venti-DHash* uses a completely distributed storage among all the participants, as in a peer-to-peer file sharing system. *PeerStore* [42] uses a hybrid approach to data localisation and storage where each participant deals in priority with a selection of partners (like *Pastiche*). Additionally, it is able to perform incremental backup for only new or recently modified data. Finally, *pStore* [43] and *ABS* [44], which are inspired by versioning systems, propose a better resource usage.

Based on the observations that worms, viruses and the like can only attack machines running a given set of programs, the *Phoenix* system [45] focuses on techniques favouring diversity among software installations when backing up a machine (e.g., trying to not backup a machine that runs a given vulnerable web server on a machine that runs the same web server). The main added value is here in the partnership selection.

In [46], the authors focus on the specific issue of resource allocation in a cooperative backup system through an auction mechanism called bid trading. A local site wishing to make a backup announces how much remote space is needed, and accepts bids for how much of its own space the local site must “pay” to acquire that remote space.

In [47], the authors implement a distributed backup system, called *DIBS*, for local area networks where nodes are assumed to be trusted: the system ensures only privacy of the backed up data but does not consider malicious attacks against the service. Since *DIBS* targets LANs, all the participating nodes are known *a priori*, partnerships do not evolve, and no trust management is needed.

#### 5.8.7.2 Cooperative File Systems

As mentioned earlier, a backup system (static data files, single writer) can be implemented on top of any file system (mutable data files, multi-writer). There exist a number of peer-to-peer general file systems such as Ivy [48], OceanStore [49], InterMemory [50], Us [51], etc. We briefly present here two of them for the sake of the comparison although they are outside the scope of this survey.

*Us* [51] provides a virtual hard drive: using a peer-to-peer architecture, it offers a read-only data block storage interface. On top of *Us*, *UsFs* builds a virtual file system interface able to provide a cooperative backup service. However, as a full-blown filesystem, *UsFs* provides more facilities than a simple backup service. In particular, it must manage concurrent write access, which is much more difficult to implement in an efficient way.

*OceanStore* [49] is a large project where data is stored on a set of untrusted cooperative servers which are supposed to have a long survival time and high speed connection. In this sense we consider it as a distributed file system using a super-peers approach rather than a purely cooperative system. The notion of super-peers relates to the fact that peers are specifically configured as file servers (with large amount of storage) that can cooperate to provide a resilient service to non-peer clients.

#### 5.8.7.3 Mobile Systems

The *FlashBack* [36] cooperative backup system targets the backup of mobile devices in a Personal Area Network (PAN). The nature of a PAN simplifies several issues. First, the partnerships can be defined statically as the membership in the network changes rarely: the devices taking part in the network are those that the users wear or carry. Second, all the devices participating in the cooperative backup know each other. They can be initialised altogether at configuration time so there is no problem of handling dynamic trust between them. For instance, they may share a cryptographic key. *MoSAIC* [52][53] is a cooperative backup system for communicating mobile devices. Mobility introduces a number of challenges to the cooperative backup problem. In the context of mobile devices interacting spontaneously, connections are by definition short-lived, unpredictable, and very variable in bandwidth and reliability. Worse than that, a pair of peers may spontaneously encounter and start exchanging data at one point in time and then never meet again. Unlike *FlashBack*, the service has to be functional even in the presence of mutually suspicious device users.

#### 5.8.7.4 Requirements on infrastructure (hw/comm) and low-level (sw/OS) support

The cooperative backup service will heavily call the Proximity Map to identify neighbouring cars and the Trust and Cooperation Oracle Service to enforce cooperation between the participating cars.

## 6 The High Availability Services defined by the Service Availability Forum

Basic high availability requirements of HIDENETS services in both the infrastructure and the ad-hoc domain should be supported by existing services where possible instead of reinventing such services. Even if implementations of basic High Availability (HA) services do not yet exist for a given platform architecture, the HIDENETS services should be designed such that they can make use of the service when it becomes available. The Service Availability Forum (SAF) [54] has defined standard services and APIs for environments with extensive high availability requirements. The SAF services can provide a good base to satisfy HA requirements within HIDENETS, although there are some issues with the applicability of the SAF services in an ad-hoc cluster, as described in 6.1. If a HIDENETS service depends on a basic service not yet covered by the SAF specifications, it should fill the gap by its own implementation, but it is advised that the APIs to access the new basic service are designed according to the guidelines for the SAF services.

Each SAF service is described in a dedicated standards specification. Specifications which have reached a 'B' release are stable in the sense that the documented APIs must not be changed in an incompatible way. Specifications with an 'A' release may still be subject to significant and, especially, incompatible changes.

The SAF standards cover specifications belonging to one of two groups:

- the Application Interface Specification (AIS) [55] defines an *Availability Management Framework* (AMF) and *AIS services*. Each of the AIS services provides a set of highly available resources to its clients.
- the Hardware Platform Interface (HPI) defines APIs for monitoring and controlling hardware resources. The SAF HPI will not be considered further in this document.

The AMF and AIS services are assumed to exist in a cluster. The AMF provides redundancy control for registered redundant application components in a given cluster. It monitors component errors and controls component failover in error situations. It is possible that a component and its redundant counterparts reside on the same node, but the major use case would be that the components are distributed over several nodes.

The AIS services provide a single system image of their managed resources, i.e. clients have no knowledge on which node(s) the resources are kept. The resources remain accessible after node failures. This concept makes the AIS services suitable for applications which cooperate on cluster-wide visible resources. Any HIDENETS application or service of this type is a candidate user of the respective AIS service if it operates on one or more of the following resources:

- cluster nodes (i.e. HIDENETS nodes)
- checkpoints (for storing and retrieving application states)
- message queues or queue groups (for m:1 or m:n communication)
- events (for m:n communication with publishers and subscribers)
- locks (for synchronisation)
- the information model of the AIS services or others (for system management)
- logs and notifications (for system state recording)
- named objects (for storage and lookup)
- timers (for the control of timed operations)

The AIS defines the interfaces to these resources resp. to the services which manage them as C APIs.

## 6.1 Applicability of the Service Availability Forum (SAF) Specifications to ad-hoc Clusters

The properties of clusters hosting implementations of the AMF and the AIS services have been discussed under many aspects in the SAF, e.g. for virtualised architectures with several logical nodes on the same hardware, i.e. on the same physical node. However, it has not been examined to what extent AMF and the services are appropriate for being used in ad-hoc clusters. The following items describe the main issues with ad-hoc clusters regarding the AIS cluster membership.

### 6.1.1 Configured Nodes and Member Nodes

The AIS cluster model is described in the specification of the SAF Cluster Membership Service, one of the AIS services: The cluster consists of *configured nodes*, while some of them may be *member nodes* at a given time. This notion suggests a cluster with a static configuration in which dynamic membership changes of node can take place – a model which is well suitable for an infrastructure cluster.

For an ad-hoc cluster, the above terms need a specific interpretation: If an ad-hoc-cluster is considered as a set of (somehow cooperating) nodes at a given time, then any node which enters or leaves the SAF cluster membership is regarded as a configured node. But there may be nodes in the cluster that never enter the SAF cluster membership. It is implementation dependent whether the service has knowledge about them and reports their existence or not.

### 6.1.2 Membership in Multiple Clusters

An ad-hoc node may exist in more than one cluster at the same time. While the SAF Cluster Membership Service APIs were designed for a single cluster, SAF node names are qualified with a cluster name. Other “nodal” objects in the AIS, i.e. objects that exist on a node or are closely related to a node (e.g. a checkpoint replica), can be associated with a given cluster. However, the AIS does not go into the subject of a node being a member in multiple clusters at a time. It is easiest to logically separate all objects from different clusters so that they are only accessible within the cluster where they have been created. This would mean that all objects - not only the nodal ones – need to have additional cluster qualifiers in their names. It is the responsibility of the AIS services to protect their objects against unauthorised accesses.

### 6.1.3 Dependency between AIS Applications and the SAF Cluster Membership Service

A configured node which is not a member of the cluster should not run an application that makes use of any of the AIS services. This is because all AIS services have a local share on the node where the application client is running, and services on member nodes are not assumed to synchronise with services on non-member nodes. This assumption implies the existence of a SAF Cluster Membership Service on each node which hosts AIS applications.

### 6.1.4 Conclusions on the applicability of the SAF AIS to ad-hoc clusters

Because of the special properties of an ad-hoc cluster, not all elements of the SAF AIS are applicable in their already existing form. It is still a target of current research activities within the HIDENETS project, whether a SAF AIS compatible HIDENETS platform can be defined in such a form, that applications that do not make use of the special ad-hoc properties can run on both the standard SAF AIS and the HIDENETS platform without changes. Because of the above enlisted issues the two platforms will have internal differences. The ad-hoc clusters related parts of the HIDENETS platform has to comply with the guidelines of the SAF standards. The other parts have to be able to act as parts of a SAF AIS implementation.

## 6.2 Availability Management Framework (AMF) and the Application Interface Specification (AIS) Services

The AIS specification makes no statements about interdependencies between AMF and the services, but it is obvious that some services may well provide base functionalities for others, e.g. AIS services might be aggregated in redundancy groups maintained by the Availability Management Framework. The following sections cover the properties of each service, while Annex I. demonstrates the same services by enlisting the functionalities of their main methods.

### 6.2.1 Availability Management Framework

#### 6.2.1.1 Short Description

The Availability Management Framework is a general approach for high availability needs in environments which run redundant components. Its goal is to ensure application availability by detecting component failures and shifting service load from failed components to sane components.

The Availability Management Framework is responsible for ensuring the availability of workloads assigned to *service units* within a cluster. The workloads are called *service instances*. A service unit may e.g. have an active or a standby role for a given service instance. A service unit is associated with a node or it may even exist outside of a cluster. The redundancy relations of service units are defined in *service groups*. The Availability Management Framework supports the following redundancy models for service groups:

- 2N (1 active, 1 standby for all service instances)
- N+M (N active, M standby for all service instances. For a given service instance, there is 1 active and at most 1 standby)
- N-Way (N active or standby for all service instances. For a given service instance, there is at most 1 active and possibly more than 1 standby)
- N-Way active (N active for all service instances. No standby. For a given service instance, there is possibly more than 1 active)
- No Redundancy (1 active for a single service instance only. No standby)

An *application* may consist of one or more service groups. Service units are aggregates of *components*. Likewise, service instances are aggregates of *component service instances*.

The Availability Management Framework controls its components by the following means

- error detection
- active and passive monitoring support
- health checks

If an error is detected or reported, the Availability Management Framework runs attempts to recover from it according to settings in the configuration or through the API. The possible recovery actions are

- Restart the failed component
- Failover of the failed component
- Switchover or failover of the affected node (switchover is more graceful than failover)
- Node failfast (fast reboot)
- Restart of the application (might impacted several nodes)
- Cluster reset

The cluster reset will not be applicable to an ad-hoc cluster.

### 6.2.1.2 Relevant applications/use cases

All applications with redundancy aspects are candidates for running under the control of the Availability Management Framework. This is especially true for the service applications in the infrastructure domain.

The objects maintained by the Availability Management Framework must be configured, and some of them are nodal objects. It is obvious that such objects cannot be configured for specific nodes, but the configuration must be a *virtual* one in the sense that the associations of objects to nodes become runtime properties.

Example: A node hosting a service unit of a 2N service group enters a cluster. If no other instance of the service unit is detected running it makes the local service active. If a single active instance of the service unit is detected running it makes the local service standby. If two redundant instances are already running, the local service unit is either stopped or prevented from starting.

It is important to note that the configuration of AMF entities and especially virtual configurations are not covered by the Availability Management Framework standard at the time of writing.

### 6.2.1.3 Entities affected

If an error is recovered locally on a node, only that node is affected. If an error is recovered by a failover between entities on more than one node, all nodes within the corresponding service groups are affected.

### 6.2.1.4 Relation to other building blocks

Complex HIDENETS services can be configured as service units within the Availability Management Framework given that they are intended to provide redundancy within service groups.

Redundant services handled by the diagnostic manager and the reconfiguration manager may as well be handled by the Availability Management Framework.

The Availability Management Framework provides availability control for all monitored services - even non-redundant services can be monitored, but automatic recovery actions in case of service failures are limited to restarts.

### 6.2.1.5 Standards Status

The Availability Management Framework is standardised in the SA Forum document SAF-AIS-AMF-B.02.01.

## 6.2.2 **Cluster Membership Service**

### 6.2.2.1 Short Description

The Cluster Membership Service provides information about the current cluster configuration and the nodes that are members of the cluster. The cluster consists of a set of configured nodes.

Some or all of the configured nodes are member nodes at a given time. The Cluster Membership Service is the authority that determines which nodes are members of the cluster and which nodes aren't. The set of member nodes at a given point in time is referred to as the cluster membership, or simply membership. The Cluster Membership Services does especially offer an asynchronous tracking interface to inform registered applications about cluster membership changes instantly.

### 6.2.2.2 Targeting problem

The Cluster Membership Service support cluster-aware applications that need to know the configuration of a cluster and that need to be informed on any cluster configuration change. They use this information to adapt their own states according to the new situation in the cluster.

If there are dependencies between cluster-aware applications, then it is essential that application state changes due to cluster configuration changes happen according to the dependencies. However, the sequencing is not enforced by the Cluster Membership Service and must be achieved by other means.

### 6.2.2.3 Entities affected

Any node which is a member of a cluster in the sense of CLM must run an instance of the Cluster Membership Service. The Cluster Membership Service needs to run on each node which hosts any other SAF service.

The Cluster Membership Service is suitable for both the infrastructure domain and the ad-hoc domain. The special aspects of deploying the Cluster Membership Service in an ad-hoc cluster have been mentioned above.

### 6.2.2.4 Relation to other building blocks

This service is appropriate for applications that act independently from others, need to cooperate with peers on other nodes and have to know the states of the other nodes. An application that is monitored by the Availability Management Framework doesn't usually need to be cluster aware.

### 6.2.2.5 What need to be known to design the building block

The Cluster Membership Service is standardised in the SA Forum document SAF-AIS-CLM-B.02.01.

## 6.2.3 Checkpoint Service

### 6.2.3.1 Short Description

A *checkpoint* contains application-specific data partitioned in *sections*. It is a cluster-wide entity designated by a unique name. A checkpoint is made highly available by *replication*. The Checkpoint Service is responsible for the handling and the replication of checkpoints. The application component requests the creation, the update and the deletion of checkpoints which are replicated according to configured or application-defined rules. Checkpoints are typically used to store application data which are needed when the service must be transferred from a failed component instance to a sane redundant component instance. The sane component uses the checkpoint to synchronise itself to the last sane state of its failed counterpart such that the service can be resumed seamlessly.

Checkpoints can be set up for synchronous or asynchronous update. Synchronous updates provide better reliability at the cost of performance. The association of checkpoints with their hosting nodes is either defined in a configuration, or it is left to the applications by creating *collocated* checkpoints. A replica of a collocated checkpoint is created on each node where such a checkpoint is opened by an application.

### 6.2.3.2 Targeting problem

The Checkpoint Service supports the failover of applications with no or minimal service loss.

### 6.2.3.3 Relevant applications/use cases

All stateful redundant applications have the need to store their intermediate states in checkpoints.

### 6.2.3.4 Entities affected

The Checkpoint Service should be available on each node that hosts a redundant application.

Any node hosting a client of the Checkpoint Service needs to run a node-local server counterpart.

The Checkpoint Service is well suitable for deployment in the infrastructure domain. Collocated checkpoints are generally suitable for the deployment in the ad-hoc domain as well. If the use of non-collocated checkpoints in ad-hoc clusters is desired, then virtual configuration mechanisms are needed, like for AMF entities.

### 6.2.3.5 Relation to other building blocks

Other building blocks are expected to act as clients of the Checkpoint Service. Especially the components monitored by the Availability Management Framework are likely to use the Checkpoint Service.

The Checkpoint Services provides the access interfaces for all checkpoints and the management of all checkpoint replicas. The Checkpoint Service stores its own runtime states in the database maintained by the

Information Model Management Service. These data can e.g. be accessed by the Replication Manager for informing its own clients.

#### 6.2.3.6 Standards Status

The Checkpoint Service is standardised in the SA Forum document SAF-AIS-CKPT-B.02.01.

### 6.2.4 **Message Service**

#### 6.2.4.1 Short Description

The Message Service provides *queues* and *queue groups* for the m:1 resp. m:n communication between Message Service clients within a cluster. A queue consists of the collection of separate *priority areas* to hold data of different priorities. It can be opened and read by at most one client at a time, but its existence is not bound to the existence of its opening client. The queue may be marked as being *persistent* or as having a *retention duration*. Such queues are left intact after being closed.

A queue group is a collection of queues which are addressed as a single entity. A queue may be a member in more than one queue group. Messages sent to a queue group are distributed to its members according to a *policy* associated with the queue group. There are unicast policies and multicast policies meaning that messages sent to a queue group are directed either to one or to several member queues.

A message sent to a queue has the following properties:

- type – identifies the purpose of the message (application defined)
- version – messages of the same type might have different versions (application defined)
- size – message size in bytes (application defined)
- sender name – optional identification of the message sender (application defined)
- send time – time stamp (in nanoseconds) provided by the Message Service
- priority – one of four message priority numbers (application defined)

A receiving process may either wait for the reception of a message, or it may be informed asynchronously when a message arrives. The next message to be obtained from a queue is always the one with the highest priority. Messages of the same priority are received in the order in which they have been placed in the queue.

If a process closes a queue and another process, perhaps on another node, re-opens the queue without clearing its contents, it can continue processing messages at the point where the closing process had stopped. Switchovers from one process to another can thus happen without loss of data. Message senders are not aware which process receives messages from the queue or queue group that they send to.

#### 6.2.4.2 Targeting problem

The Message Service provides resilient communication resources for registered clients. Resilience is achieved by the following properties:

- the existence of a queue or queue group is not coupled to the existence of its creator or owner
- queue groups are automatically reorganised when a member queue is deleted or becomes unavailable

#### 6.2.4.3 Relevant applications/use cases

All applications with the need for resilient communication are potential users of the Message Service.

#### 6.2.4.4 Entities affected

Any node hosting a client of the Message Service needs to run a node-local server counterpart. Since no special configuration is needed, the Message Service is suitable for both the infrastructure domain and the ad-hoc domain.

#### 6.2.4.5 Relation to other building blocks

Though it is not expected that the entire communication within a HIDENETS network will use the abstractions provided by the Message Service, a few building blocks are potential clients of the Message Service. For example, the Freshness Detector might evaluate the provided message time stamps against the node-local time to judge if the received message is still usable.

Any message communicated via a queue or queue group maintained by the Message Service can be seen as an output of this service to other building blocks.

#### 6.2.4.6 Standards Status

The Message Service is standardised in the SA Forum document SAF-AIS-MSG-B.02.01.

### 6.2.5 Event Service

#### 6.2.5.1 Short Description

The Event Service permits an m:n communication between event *publishers* and event *subscribers*. It supports the distribution of information (by the publishers) to a set of “interested” applications (the subscribers), which can select this information according to specified filter criteria. Communication takes place over *event channels*. Multiple publishers and subscribers can communicate over the same event channel. The Event Service imposes no meaning of the data contained in an event apart from the following header attributes:

- cluster-wide unique identifier
- name of the publisher
- publishing time
- pattern (publisher-defined event metadata by which subscribers may filter events)
- event priority
- retention duration (time for which the event is kept available after being published)

Events and event channels are cluster-wide resources. If an event has a non-zero retention duration it can be inspected by subscribers which open the corresponding channel after the event has been published.

Beyond event retention, the Event Service satisfies the following communication properties for events communicated over a channel:

- Best effort delivery – the events should be delivered to the subscribers by best effort, thus, it may happen that only a subset of the subscribers get the event while the others do not.
- At most once delivery – The Event Service must not deliver the same event for a particular subscription of a particular subscriber multiple times.
- Event priority – Events are published with a certain priority. High priority events are delivered to subscribers ahead of low priority events. In case of overflow, low priority events are discarded from the subscriber queues to make room for high priority events.
- Event ordering – At a given priority level, events sent by a given publisher are received by subscribers in the order in which the publisher published the events.
- Event completeness – A complete event is an event that has been accepted by the publish event call; it contains the entire event data provided by the caller (if any), all event attributes provided by the caller or supplied by the Event Service when publishing the event. The Event Service guarantees that a process subscribing for events either obtains complete events or no event at all

To reduce the number of received events, an application is able to define *event filters*. To be able to apply the filters the users of the event channel, i.e. the publishers and subscribers, must share the same conventions regarding the number of events patterns being used, their ordering and contents, as well as their meanings.

#### 6.2.5.2 Targeting problem

The Event Service supports applications that exchange information between publishers and subscribers. An event channel can be viewed as an electronic bulletin board.

#### 6.2.5.3 Relevant applications/use cases

The blackboard application (see deliverable D1.1 [1]) is a use case where the Event Service could provide the major architectural building block.

#### 6.2.5.4 Entities affected

Any node hosting a client of the Event Service needs to run a node-local server counterpart. Since no special configuration is needed, the Event Service is suitable for both the infrastructure domain and the ad-hoc domain.

#### 6.2.5.5 Relation to other building blocks

Implementations of the Notification service may use the Event Service as a base mechanism.

#### 6.2.5.6 Standards Status

The Event Service is standardised in the SA Forum document SAF-AIS-EVT-B.02.01.

### 6.2.6 **Lock Service**

#### 6.2.6.1 Short Description

The Lock Service provides cluster-wide *lock resources* and the ability to set or release locks on them. Locks are used to synchronise accesses from competing processes or nodes to shared resources.

The Lock Service provides a simple lock model supporting two locking modes for exclusive access and shared access. All implementations must offer synchronous and asynchronous calls, lock timeout, trylock, and lock wait notifications. Implementations may optionally offer the additional features of deadlock detection and lock orphaning. A special Lock Service API allows an application to check whether optional features are supported.

The locks provided by the Lock Service are not recursive. Thus, claiming one lock does not implicitly claim another lock; rather, each lock must be claimed individually.

#### 6.2.6.2 Targeting problem

The Lock Service is a general mechanism for synchronisation of concurrent activities in the cluster.

#### 6.2.6.3 Relevant applications/use cases

All applications with node-spanning synchronisation needs are potential users of the Lock Service.

#### 6.2.6.4 Entities affected

Any node hosting a client of the Lock Service needs to run a node-local server counterpart. Since no special configuration is needed, the Lock Service is suitable for both the infrastructure domain and the ad-hoc domain.

#### 6.2.6.5 Relation to other building blocks

The need for using the Lock Service might not be obvious when the functionality of an application or a service is roughly defined. However, this is likely to change during the design phase when concurrent activities and their synchronisation become an issue.

#### 6.2.6.6 Standards Status

The Lock Service is standardised in the SA Forum document SAF-AIS-LCK-B.02.01.

## 6.2.7 Information Model Management Service

### 6.2.7.1 Short Description

The Information Model Management Service provides the information base of all objects handled by services attached to it. It is intended as a repository especially for the SAF services, but not restricted to them. It keeps information about various objects belonging to the attached services, e.g. the configuration and runtime attributes of service units, checkpoints, message queues, etc. Unlike the other SAF services, the Information Model Management Service has two sets of interfaces:

- the *object management interface* (OM-API) is used by management clients to retrieve, create, modify or delete objects maintained in the information model.
- the *object implementer interface* (OI-API) is used for the communication to modules implementing the objects maintained in the information model. Each SAF service acts as implementer for its own set of objects.

The Information Model Management Service can be seen as a gateway between management clients and the attached services with both “northbound” and “southbound” interfaces. Clients may create, retrieve, modify or delete objects. Simple object accesses may be bundled in transactions to allow more powerful operations. Any administrative operation beyond object accesses is also passed by the Information Model Management Service between the management client and the targeted service. An example of an administrative operation is the LOCK operation of the AMF, by which certain AMF objects can be taken out of service.

### 6.2.7.2 Targeting problem

The Information Model Management Service provides the information repository for all services which are attached to it.

### 6.2.7.3 Relevant applications/use cases

All applications or services with the need for providing and, respectively, accessing a cluster wide repository of objects, are potential users of the Information Model Management Service.

### 6.2.7.4 Entities affected

Any node hosting a client of the Information Model Management Service needs to run a node-local server counterpart.

### 6.2.7.5 Relation to other building blocks

The Information Model Management Service could serve as a base for the Network Context Repository.

### 6.2.7.6 Standards Status

The Information Model Management Service is preliminarily standardised in the SA Forum document SAF-AIS-IMM-A.01.01.

## 6.2.8 Log Service

### 6.2.8.1 Short Description

The Log Service provides interfaces through which applications can act as *loggers*. They can log events of different categories (alarms, notifications, system information, application information) into cluster-wide resources maintained by the Log Service, the *log streams*. There may be many streams for application information, but only one for any of the other categories. The Log Service takes care that the data from the streams are stored in *log files*.

Log retrieval APIs are not (yet) defined. Log files contain the logged records in human-readable format.

### 6.2.8.2 Targeting problem

The main task of the Log Service is the storage of logged information for diagnostic or any other purposes.

### 6.2.8.3 Relevant applications/use cases

The Log Service provides a general purpose mechanism for storing logs from different sources in a cluster-wide repository. As such, it is applicable to a variety of potential clients.

### 6.2.8.4 Entities affected

Any node hosting a client of the Log Service needs to run a node-local server counterpart.

### 6.2.8.5 Output to other building blocks

Implementations of the Notification service may use the Log Service as a base mechanism.

### 6.2.8.6 Standards Status

The Log Service is preliminarily standardised in the SA Forum document SAF-AIS-LOG-A.01.01.

## 6.2.9 Notification Service

### 6.2.9.1 Short Description

The Notification Service provides APIs to work with *notifications* in a *producer*, *subscriber* or *reader* role. A notification is a data structure which describes an important event (in its natural meaning, not in the one of the SAF Event Service) during the lifetime of an HA cluster. There are five notification types, according to event types being defined in ITU X.73x:

- alarm
- state change
- creation or deletion of an object
- change of an object attribute
- security alarm

Each notification has additional type-dependent parameters, but there is a set of common parameters for notifications of all types, among them a time stamp and IDs by which notifications can be correlated.

Producers are the sources of notifications, while subscribers and readers are their consumers. Subscribers use a push-interface for retrieving notifications, i.e. when a notification occurs it is forwarded to its subscribers. In contrast, readers use a pull interface. They retrieve the notifications actively from a persistent notification log. Subscribers and readers may specify filter criteria for the notifications that they want to see.

Besides applications, some SA Forum services may also generate notifications. These notifications are service specific and they can be found in the “Alarms and notifications” section of the specifications.

### 6.2.9.2 Targeting problem

The Notification Service implements a persistent log of notifications and access methods for producers, subscribers and readers.

### 6.2.9.3 Relevant applications/use cases

All applications that log notifications with the parameters expected by the Notification Services, are potential users of the service in a producer role. Subscribers and readers are usually applications which need to react to notifications.

### 6.2.9.4 Entities affected

Any node hosting a client of the Notification Service needs to run a node-local server counterpart.

### 6.2.9.5 Relation to other building blocks

Implementations of the Notification service may use the Log Service or the Event Service as a base mechanism.

The Notification Service could serve as a base for clients of the Diagnostic Manager as notification producers, while the Diagnostic Manager itself could act as a notification subscriber.

### 6.2.9.6 Standards Status

The Notification Service is preliminarily standardised in the SA Forum document SAF-AIS-NTF-A.01.01.

## **6.2.10 Naming Service**

### 6.2.10.1 Short Description

The Naming Service provides the storage and the retrieval of named objects.

The association between a name and an object is called a *binding*. Bindings are valid within a *context*. The same name can be used for different objects in different contexts, because the bindings are not the same. There are default contexts: the cluster-wide context (the binding is valid in the entire cluster) and the node-local context (the binding is only valid on the node where it has been created), but applications may create their own contexts in which they want to establish their bindings.

### 6.2.10.2 Targeting problem

The Naming Service is a general purpose service for the access to application-defined named objects which are stored as highly available resources.

### 6.2.10.3 Relevant applications/use cases

An application which deals with named objects available in various contexts may choose to use the Naming Service

### 6.2.10.4 Entities affected

Any node hosting a client of the Naming Service needs to run a node-local server counterpart.

### 6.2.10.5 Relation to other building blocks

There are various candidate applications or services which may use the Naming Service. For example, the Authentication or the Trust and Cooperation oracles might have their named objects managed by the Naming Service.

### 6.2.10.6 Standards Status

The Naming Service is defined in a preliminary standard proposal which is close to reach the A.01.01 stage.

## 6.2.11 Timer Service

### 6.2.11.1 Short Description

The Timer Service provides a mechanism by which client processes get notified when a *timer* expires. A timer is a logical object that is dynamically created and represents either absolute time or a duration (i.e. a an interval relative to a time reference point). Any absolute time or duration in the SAF specifications is defined in units of nanoseconds.

The Timer Service provides two types of timers: *single event* timers and *periodic* timers. Single event timers will expire once and are deleted after notification. Periodic timers will expire each time a specified duration is reached, and the process is notified about the expirations. Periodic timers have to be explicitly deleted by invoking a corresponding API.

Timers remain available when their creating process terminates, but they are node local resources. The propagation of a timer to another node has to be implemented by the use of another service, e.g. the Checkpoint Service.

### 6.2.11.2 Targeting problem

The Timer Service supports applications that need to set up timers and get informed when they expire.

### 6.2.11.3 Relevant applications/use cases

See above

### 6.2.11.4 Entities affected

Any node hosting a client of the Timer Service needs to run a node-local server counterpart.

### 6.2.11.5 Relation to other building blocks

The Timer Service has close relations or even overlapping functionalities with other HIDENETS services which operate on timers, i.e. the duration measurement and the timely timing failure detector. The reliable and self-aware clock service could provide the base for an implementation of the Timer Service.

### 6.2.11.6 Standards Status

The Timer Service is defined in a preliminary standard proposal which is close to reach the A.01.01 stage.

## 6.3 Conclusions on the application of AIS services interfaces in the HIDENETS architecture

It is a requirement for the HIDENETS platform to provide standardised access to its services since it greatly improves the development speed, and portability of applications. Thus, we chose the SAF AIS standardised specification collection that contains a wide range of service interfaces and also provides the guidelines for the development of further HIDENETS specific interfaces. So as it was stated in Section 6.1.4 the ad-hoc clusters related parts of the HIDENETS platform have to comply with the guidelines of the SAF standards, while other parts have to be able to act as parts of an SAF AIS implementation.

## **7 Final remarks**

This document is the first deliverable of WP2, and was developed within the efforts of task 2.1, Resilient Architecture. It provides a preliminary description of the architectural approach for achieving resilience in HIDENETS, presenting also an initial view on the overall HIDENETS node software architecture (a work that is being done by WP2 and WP3 together). It also provides preliminary descriptions of several services that are understood as relevant and necessary in HIDENETS, which were divided into two categories: Oracles services and Complex services. Finally, the deliverable also addresses initial work with respect to interfacing issues.

Given the preliminary nature of this document, our objective was essentially to provide a rather broad overview of the main ideas to be worked on in the course of WP2, including both the architectural aspects and the service aspects. A more in-depth and probably more complete discussion of the architectural approach and architectural solutions can be done in a better way in the final version of the Resilient Architecture deliverable (D2.1.2, to be delivered in December 2007).

The first part of the deliverable builds on results from WP1, namely on the several applications and use cases that have been identified and presented in deliverable D1.1. Chapter 2 provides a summary of the fundamental middleware requirements involved in each of the six use cases, which is essentially complete and stable, and then, following a top-down approach, it introduces the overall view of the HIDENETS node software architecture, which includes all the services that were identified as potentially necessary to address the application requirements with improved resilience. It is expectable that this view may evolve as a result of further work in the concrete definition of HIDENETS services and also along the joint work of WP2 and WP3 around the definition of the HIDENETS node software architecture.

Chapter 3 focuses on the architectural solutions and architectural support for the resilience solutions to be developed in tasks 2.2 and 2.3. The presented ideas are tightly connected to the HIDENETS reference model being developed in the context of WP1. In this sense, task 2.1 will continue working towards improving and refining the definition of the models and architectural solutions used in HIDENETS, thus providing also input for the HIDENETS Reference Model. Particular work that still has to be done in task 2.1, concerns the investigation and discussion of the specific ability of the proposed architecture to fulfil the requirements imposed by diagnosis, adaptation and reconfiguration strategies, at a satisfactory level with respect to the applications needs.

Chapters 4 and 5 provide initial descriptions of HIDENETS services. These are not comprehensive descriptions and should be understood as starting points for the work to be done in the remaining of the project, in particular in WP2 and WP3. Furthermore, the chapters are not closed, in the sense that more services might be found as necessary and added to the final version of the document. Each description tries to include already known information regarding possible dependencies and relations with other services, which is in fact on-going work, also in cooperation with WP3. Quite naturally, the descriptions also include some discussion of related work in order to put the work that has to be done in context.

At this point, and in this document in particular, there is not a separation of services in terms of their appropriateness or specific purpose for ad-hoc or for infrastructure based environments. We expect that this separation will be made in some cases, but probably not for all the services. This will be an issue for further research in WP2.

Interestingly, this separation of services will also be relevant for clarifying the potential relations and the possibilities of integration of HIDENETS services with existing standards for High Availability service interfaces defined by the Service Availability Forum. These standards and some initial work with respect to this integration are presented in Chapter 6 and are expected to be further developed along with the definition of the HIDENETS services.

## **Annex I: Functionalities of the HA Services Defined by the Service Availability Forum**

### **Common functionalities**

- initializing/closing the association between the application and the service, registration of the various callback functions, getting service specific handles, invoking pending callbacks
- freeing memory allocated by the service

### **Availability Management Framework (AMF) specific functionalities**

- registering/unregistering a component with the AMF, getting component names
- instantiating/terminating/killing a component
- setting/getting HA state of a component, requesting/confirming the removal of a component service instance
- starting/stopping health checks for a component, requesting/confirming health checks
- starting/stopping passive monitoring of specific errors
- starting/stopping/closing tracking changes of a protection group, detecting changes
- reporting errors and recommending recovery, clearing errors
- responding to AMF requests

### **Cluster Membership service (CLM) specific functionalities:**

- obtaining cluster membership information and tracking notifications of changes in the cluster, retrieving node membership

### **Checkpoint service (CKPT) specific functionalities:**

- opening/closing/deleting a checkpoint, confirming an asynchronous checkpoint opening, setting a local checkpoint to be the active one, setting the retention duration of a checkpoint, retrieving the status of a checkpoint
- creating/deleting/deallocating sections in a checkpoint, setting the expiration time of a section, initializing/deleting a section iterator, iterating over checkpoint sections
- writing data from memory to checkpoint, copying it back, overwriting single sections, propagating active checkpoint content to the other replicas, acknowledgement of completing propagation

### **Message service (MSG) specific functionalities:**

- opening/closing/deleting message queues in a cluster, acknowledgement of completing asynchronous opening, retrieving message queue status, setting retention time of message queues
- creating/deleting message queue groups, inserting/removing queues in groups, tracking queue group membership
- sending/retrieving a message, acknowledgement of message delivery, notifying about received message (ready for retrieval), cancelling all blocking retrieval calls
- invoking a queue for both the message transmission and the reply receipt, replying to such messages

**Event service (EVT) specific functionalities:**

- opening/closing/deleting an event channel, acknowledgement of completing asynchronous opening
- initializing/publishing/deallocating an event, setting/retrieving event attributes, subscribing/unsubscribing on an event channel, delivery notification for an event, clearing retention time of published events

**Lock service (LCK) specific functionalities:**

- opening/closing a distributed lock resource, requiring/releasing a lock, acknowledgement of completing asynchronous open/acquisition/release, informing a process about new requests on a held lock, purging orphan locks on a resource

**Information Model Management service Object Management API (IMM/OM) specific functionalities:**

- creating/deleting new configurations/classes, getting a class definition
- initializing/finalizing a search, iterating over matching objects
- initializing/finalizing an object accessor (to fetch attribute values), getting assigned attribute values
- initializing/releasing ownership handle, setting/deleting/clearing administrative owner of a set of objects
- initializing a configuration change bundle (CCB), requesting a CCB to create/modify/delete an object, applying/cancelling CCBs on the information model
- invoking an administrative operation on an object, acknowledgement of completing asynchronous admin operation invocation

**Information Model Management service Object Implementer API (IMM/OI) specific functionalities:**

- registering/unregistering a process as an object implementer (OI), setting/unsetting the OI of a class/object-set
- creating/deleting a runtime object, updating runtime attributes of an object, acknowledgement of an update operation
- enabling OI to validate and cache a create/modify/delete object request, notification about the end of a CCB, informing OI about applied/aborted CCBs
- letting an OI to perform an administrative operation, informing IMM service about the result of an administrative operation

**Log service (LOG) specific functionalities:**

- opening/closing a log stream, logging a new record to a log stream, acknowledgement of completing asynchronous opening/logging, setting enabled log severity level for a process

**Notification service (NTF) specific functionalities:**

- allocating/deallocating memory for different notifications (object create/delete, attribute/state change, alarm) and for notification structures, sending a notification
- allocating/deallocating memory for different notification filters, getting/deallocating localised textual description for a notification, retrieving a pointer in a notification structure
- subscribing/unsubscribing for notifications, delivering a notification, notifying about discarded notifications

- initializing/deallocating a notification iterator, iterating over the notifications

**Naming service (NAM) specific functionalities:**

- opening/closing/deleting a naming context, setting the retention duration of a context, acknowledgement of completing asynchronous opening
- creating/modifying/changing/looking up/deleting a binding
- subscribing/unsubscribing for monitoring changes to a binding, notifying about changes
- initializing/deallocating a context iterator, iterating over the bindings of a naming context
- getting implementation specific limits (number of contexts of a node/cluster, ...)

**Timer service (TMR) specific functionalities:**

- starting/rescheduling/cancelling a timer, retrieving the remaining time of a timer, skipping the next expiration of a periodic timer, getting attributes of a timer, getting current absolute time, getting clock tick count for a client, notifying a process about timer expiration