DECENTRALIZED COOPERATIVE MANAGEMENT: A BOTTOM-UP APPROACH

Sam Michiels, Nico Janssens, Wouter Joosen, Pierre Verbaeten
K.U.Leuven, Dept. of Computer Sciences
Celestijnenlaan 200A, B-3001 Leuven, Belgium
sam.michiels@cs.kuleuven.ac.be

ABSTRACT

Network services must self-adapt dynamically in order to accommodate available resources, system errors, and changing requirements. This need for self-reliance will increase with the growing flexibility and heterogeneity that characterizes such network technologies as sensor (ad hoc) networking, grid computing, and pervasive computing. The downside of traditional distributed network management techniques is that they do not take into account the design of each individual network node. We propose a bottom-up approach, i.e. providing a solid and adaptable architecture at the core of each network node, and on top of it a coordination and management platform that exploits the underlying flexibility. This paper proposes a node-architecture for decentralized cooperative management, structured as three clearly separated planes: (1) a communication plane that is designed with adaptability in mind, (2) a coordination plane that enables individual nodes to cooperate, and (3) a management plane that monitors the communication plane and adapts it when necessary.

KEYWORDS

Network management, software architecture, component framework.

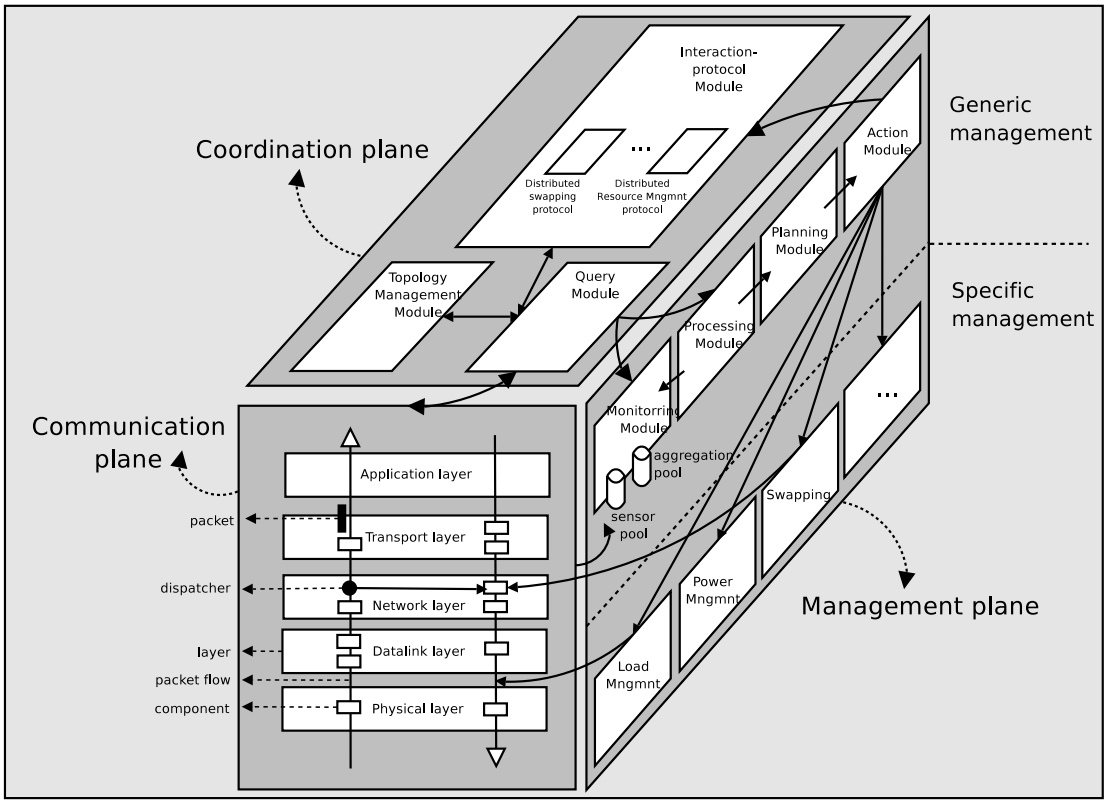
1. INTRODUCTION

Network management shows a trend from centralized paradigms (e.g. SNMP) to distributed hierarchical paradigms (e.g. mobile code inserted by a client) and, recently, strongly distributed (*decentralized*) cooperative paradigms (e.g. autonomous agents that cooperate to accomplish a common goal) (Martin-Flatin et al. 1999). In part, this trend is motivated by the increasing dynamism of ad hoc and sensor networks, in which centralized management is no longer possible.

Decentralized cooperative management requires each individual network node to cooperate and coordinate in order to reach a common global goal. This goal can be conceived as a collective intention, which is achieved by individual intentions in action (Searle 1990). For example, a collective intention to optimize network throughput for high priority traffic, can be realized by cooperation and coordination of individual nodes. One node may contribute to realize the collective intention by means of dropping low priority packets, while another node starts searching for a faster route.

Decentralized cooperative network management relies on a solid node architecture that enables cooperation and coordination between individual nodes (e.g. to realize resource control, load balancing, quality of service guarantees, etc). Each network node should be designed as an open platform, in order to react to various changes in the network (load, available bandwidth, delay), at application level (data priorities, quality of service), and in the system itself (available processing power and memory footprint, or battery level).

Figure 1. Overview of the architecture with the communication plane, the coordination plane and the management plane



We propose at the core of each network node a well-defined software architecture for building protocol stacks that are adaptable and manageable at run-time. Around this core, we provide management extensions that extract information from a running protocol stack, and deploy adaptations in it if needed. In addition, a coordination extension is provided to allow nodes to exchange information in a controlled manner.

This paper describes the next step in the evolution of our self-management architecture, i.e. providing generic support for decentralized coordination. In previous publications, we have concentrated on the core of our self-management architecture, i.e. the communication plane (Michiels 2003). We have developed a component framework that enforces an open architectural style, and allows for automatic building of protocol stacks based on an architectural description. Subsequently, we have extended the core with two different management platforms, one for load management (the DiPS+ Monitoring Architecture (DMonA) (Michiels et al. 2002, 2004a), and one for transparent and safe component hot-swapping (Customizable Protocol Stacks (CuPS) (Janssens et al. 2003)). Recently, we have developed a distributed platform for component hot-swapping (Janssens et al. 2004a). This paper bundles our experiences, and proposes a generic architecture for decentralized cooperative management.

The remainder of the paper is structured as follows. Section 2 discusses the architecture in detail, zooming in on the three planes that form the overall architecture. Section 3 shows how we have implemented the architecture. Section 4 discusses related work. Section 5 formulates the main conclusion.

2. THE ARCHITECTURE

We propose a software architecture for decentralized management of highly dynamic computer networks, which consists of three planes: a communication plane, a coordination plane and a management plane (see also Figure 1).

2.1 Communication plane

The communication plane of the architecture represents the networking sub-system of a network node, i.e. the protocol stack. The protocol stack is crucial for realizing self-adaptability of network nodes. It is responsible for processing and forwarding network packets. By consequence, end-to-end quality of network services is heavily influenced by the processing of each intermediate network node. Yet, it is impossible to develop a protocol stack that is able to efficiently handle the wide variety of services, devices, and quality-of-service requirements that are associated with current networks. Moreover, each individual protocol stack should behave in favor of the end-to-end behavior expected by the user.

By consequence, in order to optimize the processing of packets throughout the network, it is important that the protocol stack is designed with adaptability in mind. The communication plane, therefore, provides four key features that enable software adaptability: (1) modularization of the code in fine-grained and independent components, (2) separating concurrency code from functional code, (3) explicitly specifying the software architecture, and (4) providing messages as explicit first class entities.

These features have been translated in a combination of three architectural styles (see also Figure 1). The pipe-and-filter style implies that a protocol stack is built as a pipeline of components and filters. Different types of components exist for encapsulating functionality, dispatching strategies, and concurrency behavior. The layered architectural style reflects the protocol stack as being a number of connected layers. The blackboard style, finally, implies that packets are presented as first class entities. A packet can be annotated with meta-information to influence the behavior of components that process it downstream the component pipeline. In this way, components can pass information without having to know each other. This anonymous way of communication in combination with the filters makes components more independent and replaceable.

2.2 Coordination plane

The coordination plane provides a means for individual network nodes to interact. Interaction can be established in many different ways, depending on the stability of the network and its topology (wired, wireless, ad hoc), the available resources in the network nodes (routers, handheld devices, sensors), and the type of applications (multimedia, real-time). To support this diversity of features, the coordination plane defines a clear interface with the communication plane, and vice versa.

The advantages of providing the coordination plane as a clearly separate plane in the overall architecture include: (1) it is clearly identifiable as an individual entity, (2) a node can be equipped with a particular coordination plane depending on the circumstances, while reusing the core communication platform, and (3) ultimately, it can only be plugged onto the core communication plane when there is need for distributed coordination.

The second advantage (i.e. being able to install different coordination planes while reusing the core) is highly relevant since different adaptations require different interaction patterns. Janssens et al. (2004) propose distributed software agents as a means to detect network bottlenecks in mobile ad hoc networks. Yet, a different coordination protocol is required to allow replacing components in a protocol stack in a transparent, safe and consistent manner (i.e. without loosing any packets) (Janssens et al. 2004b). Tuduce and Gross (2004) describe a coordination platform for resource monitoring in ad hoc networks. This platform is heavily based on dynamic clustering of neighboring nodes.

The coordination plane consists of three main modules: the topology management module, the interaction-protocol module, and the query module. The topology management module is responsible for maintaining an up-to-date view on the network neighborhood. It can achieve this, for instance, by dynamically installing a cluster of neighboring nodes, and tracking when nodes leave or join. Or, it can delegate the work to software agents that travel the local network and dynamically update nodes with topology information. Which technology is used best depends on the dynamics of the network and the capabilities of the individual nodes. The topology management module cooperates with the query module to exchange messages between network nodes.

The interaction-protocol module encapsulates coordination protocols, which are offered as services to the management plane for deploying actions on multiple nodes. The interaction-protocol module allows for adding a customized coordination protocol for each specific type of action. The distributed hot-swapping protocol, for instance, defines the steps to replace a component at multiple nodes atomically. The distributed

resource management protocol allows for deploying customized usage patterns in upstream network nodes. For example, when a node decides to drop low priority packets because it is (almost) overloaded, it can notify upstream nodes so that they can adapt their behavior (e.g. start dropping these packets earlier in the path, or bypass the overloaded node by changing their route to the packet's destination). In both cases, the interaction-protocol must ensure that adaptations are deployed by multiple nodes in an atomic and consistent manner. The interaction-protocol module cooperates with the query module to exchange protocol-messages between nodes.

The query module is responsible for sending and receiving messages between network nodes. It handles incoming queries and delegates them to the appropriate entity for further processing, and it constructs a specific query when triggered by the topology management module or the interaction-protocol module. Three types of queries are recognized: status update queries (e.g. (aggregated) information on battery level, experienced load, topology updates, etc.), behavior adaptation queries (e.g. strategies for differentiating between user types, protocols, packet lengths, etc.), and control queries (e.g. queries to maintain a cluster). A trigger to send an outgoing query can be initiated by (1) the topology module autonomously, in order to update its view on the network neighborhood, or (2) another query that is received via the communication plane or (3) a third party (e.g. a software-agent or an administrator). The query module transforms each of these three trigger types into a query, which is transferred over the network via the communication plane.

The query module handles incoming queries as follows. Control queries are delegated to the topology management module for further handling. Status updates and queries for behavior adaptation are delegated to the management plane. Status updates are added to the aggregation pool for further processing, while queries for behavior adaptation are directly delegated to the processing entity, which interprets these queries as adaptation hints. The next section elaborates on this in more detail.

2.3 Management plane

The management architecture differentiates between a general management plane, and one or more specific management sub-planes, depending on the intended type of management. In this way, components in the general management plane can be reused for all types of management, while an application developer must only provide a minimal amount of specific components.

2.3.1 Feedback control loop

The general management plane is responsible for (1) collecting run-time information about the status of the local system and its neighborhood, (2) processing this information to decide on the most appropriate action to be taken, (3) planning the deployment of an action to preserve the system in a correct and consistent state, and (4) the deployment itself.

Separating the general management plane has multiple advantages. First of all, by extracting the management code from the individual mechanisms to actually manipulate the running protocol stack, it becomes much easier to reason about management. In addition, the general management plane is a convenient level to associate with each other different sources of information. For instance, if the monitoring module detects overload of a specific component area in the protocol stack, one solution might be to filter out packets for the overloaded area. Another more effective solution could be, however, to replace the component area by another more efficient version. The choice between solutions is made at the general management level, while deployment of a solution is delegated to a specific management sub-plane.

The general management plane architecture provides four modules for monitoring, processing, planning, and deployment. The monitoring module is responsible for extracting information from the communication plane. It consists of a sensor pool and an aggregation pool. The sensor pool is an open list of sensors, each of which gathers information from the local node. The topology sensor, for instance, detects nodes joining or leaving the network. Sensors can be provided by the specific management layers, such as sensors for throughput, power level, network delay, available bandwidth, etc. The sensor pool is responsible for collecting information, which is distributed to other nodes in the network by the coordination plane. For each local sensor in the sensor sub-module, an aggregation sub-module exists in the aggregation pool which aggregates information from other nodes. Therefore, the aggregation module exports a management interface (`addStatusInfo(String name, SatusInfo info)`), which is called by the query module in the

coordination plane. Aggregation is not necessarily deployed in all nodes. The head of a dynamically created cluster, for instance, would be a good node to collect resource information from its neighbors.

The processing module is responsible for interpreting the collected information in the aggregation pool and deciding what kind of adaptation is required. Via its policy interface (`addBehaviorUpdate(String name, BehaviorUpdate hint)`), which is called by the query module in the coordination plane, specific hints can be injected to customize behavior of the processing module. These policies can use the management interface of each aggregation sub-module to interpret the collected information. The processing module autonomously pulls (aggregated) status information from the monitoring module, and processes this information to update its view on the system's status. When a change is required, a high-level description of the required adaptation is sent to the planning module.

The planning module is responsible for deciding when the suggested adaptation can be deployed in a safe and correct way. For example, this module checks possible system restrictions related to an adaptation, it decides when the system is ready to be adapted, it detects inconsistencies with previous updates, etc. The planning module may cancel or postpone proposed adaptations, for instance, when a previous adaptation has just been installed and the system's behavior is not yet stabilized. It may also add restrictions to the deployment of an adaptation (e.g. with respect to timing, memory usage, or the effect on packet throughput). When the planning module decides that a proposed adaptation can be deployed safely, it notifies the action module.

The action module translates the high-level policy decision, which was suggested by the processing module, to one or more specific actions. The action module manages a mapping between a policy decision and the action that is most appropriate to fire. Via its management interface, specific mappings can be added to the action module, as well as selection criteria to select an action when alternatives are available. These selection criteria can take into account the restrictions imposed by the planning module. For instance, when the processing module concludes that the measured load of the system has crossed a threshold, the action matrix maps this event to one or more appropriate actions. In case alternatives are available, the action module selects an action based on a selection strategy. In case of the overload example, possible actions are to install a customized scheduling strategy to drop low priority packets, or to replace a component by a better performing version. The latter may be more effective, but involves more overhead during component-swapping. These four modules together with the communication plane, create a local feedback control loop as follows. Local information is extracted from the underlying communication plane by the monitoring module and collected in the aggregation pool. This local and global information is processed by the processing module. When the planning module decides that the system is ready to be adapted, the action module can adapt the local communication plane in cooperation with one or more specialized management sub-planes. In addition to this local feedback control loop, a distributed control loop is created. Indeed, the action module can also initiate distributed actions on neighboring nodes by triggering the interaction-protocol module in the coordination plane. The results of these decentralized adaptations will be evaluated by the processing module when nodes send their updated information to an aggregation pool (e.g. in the cluster head).

2.3.2 Specialized management sub-planes

The management plane differentiates between different sub-planes, which can be classified in two categories: behavioral management and functional management. The former category adapts the underlying behavior of the communication plane without changing the functionality. In other words, it changes *how* the system performs, but not *what* the system does. The latter category customizes the communication plane by adding, removing, or replacing functional components, i.e. changing *what* the system does.

The main responsibility of each sub-plane is to translate the action specified by the action module into specific actions at the protocol stack level. In case of overload, for instance, the load management sub-plane translates the action of changing a packet scheduling strategy into calling a specific concurrency component in the component pipeline. Management sub-planes make extensive use of the architecture specification of the communication plane. Based on this specification, the hotspots to be changed can be identified and actions can be transformed into architecture manipulations.

Figure 1 shows three examples of specific management sub-planes. Load management and power management change the behavior of the communication plane when the system is respectively overloaded or running on low battery. The swapping sub-plane adapts the functionality of the communication plane by adding, removing or replacing components while the system is running.

3. PROTOTYPES

We have developed and evaluated prototypes of the management architecture presented in this paper. The DiPS+ communication architecture is used to validate the communication plane, DMonA and CuPS are used to investigate the management plane, and NeCoMan presents current research on coordination middleware.

3.1 DiPS+ communication architecture

We have developed a protocol stack component framework that enforces the architecture of the communication plane as described in Section 2.1. In addition, we have implemented a builder tool that interprets an architecture specification, and automatically builds a protocol stack from it. The component framework is referred to as DiPS+ (Distrinet Protocol Stack) (Michiels 2003). It has been used in many case studies, ranging from Master theses to industrial prototypes (PEPiTA project 2001, SCAN project 2003).

3.2 DMonA and CuPS management planes

We have developed a load management architecture, called DMonA (DiPS+ Monitoring Architecture). DMonA allows for monitoring the throughput of the underlying communication plane, and adapting its behavior if local bottlenecks are detected. A local bottleneck is an area where many more requests arrive than can be processed during some period of time. When a bottleneck has been detected, DMonA tries to solve it, for instance, by inserting a customized packet scheduling strategy, or by favoring specific component areas to speed up areas that are about to free some crucial resources. The DMonA prototype has been used to monitor local throughput of a RADIUS authentication server and optimize its behavior for specific groups of users (Michiels et al. 2004a, 2004b). In addition to DMonA, we have developed a platform for transparent and safe component hot-swapping, called CuPS (Customizable Protocol Stack). CuPS allows to add, remove or replace components in a running protocol stack, without losing any packets if this is required. Replacing a component involves manipulation of the communication plane architecture in multiple places in order to block all incoming packet flows during the replacement, and to allow state transfer from the old to the new component (Janssens et al. 2003).

3.3 Necoman Coordination middleware

The NeCoMan (Network Consistency Management) middleware is a generic distributed interaction-protocol responsible for safe addition, replacement and removal of point-to-point services among programmable nodes (Janssens et al. 2004a). The main contribution of NeCoMan lies in its ability to improve the effectiveness of the deployment process. This is achieved by customizing the deployment process depending on the properties of both the network service to be deployed and the underlying execution environment. By automatically selecting the most effective deployment scenario, the performance penalty of safe distributed service deployment is minimized. NeCoMan has been developed on top of the DiPS+/CuPS node architecture. Yet, it is independent from the underlying node architecture. Adaptations of the underlying protocol stack are expressed in terms of creating, removing, linking and unlinking components. By consequence, the NeCoMan protocol can also be used in combination with adaptable component-based protocol stack architectures other than DiPS+.

4. RELATED WORK

Akyildiz and Kasimoglu (2004, p. 362) have proposed a reference protocol stack architecture for wireless sensor and actor networks. The architecture presented in this paper fits in this high-level view and is already implemented and validated in part. We see three domains that are highly relevant from this perspective: reflective middleware, software architectures and component frameworks, and self-managing systems. The reflective middleware research group at Lancaster University proposes a complementary approach to decentralized cooperative management from a reflective middleware perspective. Blair, Coulson, and Grace

(2004) argue that "there is great potential in developing future systems that are *vertically integrated* and can be seamlessly inspected and adapted as a unified pool of component-based functionality. [...] In the future, we see the middleware [...] merging into the underlying operating systems and network infrastructures." This confirms the idea of close collaboration between coordination and management middleware, with the underlying communication platform. While they are working top-down, we are following a bottom-up approach (starting from a solid communication platform, and then going up to the middleware).

The Rainbow project at Carnegie Mellon (Garlan et al. 2004a) proposes an architecture-based, dynamic self-adaptation framework that monitors and incrementally adapts a target running system using the system's architectural model. Adaptations are based on changes in the architecture specification, and then translated to the underlying implementation. The Ninja architecture (Gribble et al. 2001) defines a distributed hierarchical system architecture with well-engineered workstation clusters, heterogeneous client devices, and active proxies in the network to manipulate data streams. The architectural models in Rainbow and Ninja specify the distributed system's architecture rather than the software architecture of an individual node as, for instance, the DiPS+ protocol stack architecture. Rainbow and Ninja heavily depend on a stable and hierarchical network infrastructure with proxies, routers and clusters. These kinds of models are less usable in highly dynamic ad hoc networks, in which the topology and the characteristics of both nodes and communication links continuously change. This paper proposes a solution for network management in case no hierarchical system architecture is available.

During the last five years, a considerable research community has grown around the topic of self-managing systems (Garlan et al. 2002, 2004). The concept of a feedback control loop is generally accepted in this community. The challenge is to support all stages of such a control loop, and to integrate them in a consistent manner. In facilitating the development and maintenance of self-managing systems, the goal is to separate adaptation policies from the mechanisms that implement them. The platform we propose in this paper separates adaptation policies from specific management mechanisms, and provides a high-level architecture with clearly separated planes for communication, coordination and management.

5. CONCLUSION

This paper has proposed a decentralized cooperative architecture for building and managing self-adaptable network nodes. The architecture consists of three planes: (1) the communication plane, which provides an open protocol stack architecture, (2) the coordination plane, which allows for decentralized coordination among nodes, and (3) the management plane, which allows for monitoring and changing the protocol stack, and may encapsulate multiple specific management sub-modules.

The main contribution is that the solution proposed is based on a core software platform for building adaptable and manageable protocol stacks. This core is the foundation for the management and the coordination plane. This bottom-up approach contrasts the top-down approach that is usually followed, i.e. providing a management and coordination platform, and then investigating how this platform can influence the behavior of an individual network node.

Thanks to our bottom-up approach, we have a clear view of the advantages for the offered quality of service when being able to customize protocol stack behavior. We believe that our approach is very powerful for ad hoc networks, where individual nodes cannot rely on a solid network infrastructure to manage the network's behavior.

ACKNOWLEDGEMENTS

Research for this paper was sponsored by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). Special thanks to Lieven Desmet for his valuable comments on the paper and to Ann Heylighen for proofreading the text. This paper was drafted mainly during Sam Michiels' stay in the group of Prof. Eric Brewer at UC Berkeley in 2004-2005, made possible by a grant from the Fund of Scientific Research – Flanders (FWO), which is gratefully acknowledged.

REFERENCES

Akyildiz I. F. and Kasimoglu I. H., 2004. Wireless sensor and actor networks: research challenges. *Ad Hoc Networks*, 2(4):351–367.

Blair G.S.,Coulson G., Grace P., 2004, Research directions in reflective middleware: the lancaster experience. *Proceedings of the 3rd Workshop on Reflective and Adaptive Middleware (RM2004)*, Toronto, Ontario, Canada.

Garlan D. et al, 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54.

Garlan D., et al, 2002. *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*. ACM Press, New York, NY, USA.

Garlan D. et al, 2004. *Proceedings of the Second ACM SIGSOFT Workshop on Self-Managing Systems (WOSS'04)*. ACM Press, New York, NY.

Gribble S.D. et al, 2001. The ninja architecture for robust internet-scale systems and services. *Computer Networks, Special Issue on Pervasive Computing*, 35(4):473–497.

Janssens N. et al, 2003. Towards Transparent Hot-Swapping Support for Producer-Consumer Components. *Proceedings of Second International Workshop on Unanticipated Software Evolution (USE 2003)*, Warsaw, Poland.

Janssens N. et al, 2004a. A Modular Approach Enforcing Safe Reconfiguration of Producer-Consumer Applications. P*roceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, Chicago IL, USA.

Janssens N. Et al, 2004b. An Agent Design Method Promoting Separation Between Computation and Coordination. *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC 2004)*, pp. 456–461.

Michiels S. et al, 2002. Self-adapting concurrency: The DMonA architecture. *Proceedings of the First Workshop on Self-Healing Systems (WOSS'02)*, Charleston, SC, USA, pp. 43–48.

Michiels S., 2003. *Component Framework Technology for Adaptable and Manageable Protocol Stacks*. PhD thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium.

Michiels S. et al, 2004a. The DiPS+ software architecture for self-healing protocol stacks. *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA-4)*, Oslo, Norway.

Michiels S. et al, 2004b, A DiPS+ Case Study: A Self-healing RADIUS Server. Technical Report CW-378, Dept. of Computer Science, K.U.Leuven, Leuven, Belgium.

Martin-Flatin J.P. et al, 1999. A survey of distributed enterprise network and systems management paradigms. *Journal of Network and Systems Management*, 7(1):9–26.

PEPiTA project, 2001, ITEA (Information Technology for European Advancement) project home page. Available: http://pepita.objectweb.org/ (Accessed 2004, December 6).

SCAN project, 2003. IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders) project. Towards Service Centric Access Networks, IWT project # 010319.

Searle J., 1990. Collective intentions and actions. *Intentions in Communications*. MIT Press, Bradford Books, Cambridge, MA, USA.

Tuduce C. and Gross T, 2004. Resource Monitoring Issues in Ad Hoc Networks. *Proceedings of the International Workshop on Wireless Ad-Hoc Networks (IWWAN2004)*, Oulu, Finland.