SmartWare – Framework for Autonomic Application Services

David Linner[†], Heiko Pfeffer[†], Carsten Jacob[*], Anna Kress[*], Steffen Krüssel[*], Stephan Steglich[†]
[†]Technische Universität Berlin,
Sekr. FR 5-14, Franklinstrasse 28/29, 10587 Berlin, Germany
{ david.linner|heiko.pfeffer|stephan.steglich}@tu-berlin
[*]Fraunhofer Institute for Open Communication Systems (FOKUS),
Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany
{carsten.jacob| anna.kress|steffen.kruessel|}@fokus.fraunhofer.de

ABSTRACT

The combination of mobile and embedded computing devices holds potential for a novel landscape of application and services in the direct surrounding of the user. The realization of such a landscape requires solutions to cope with the volatile nature of the environment composition, the absence of central management infrastructure, and the heterogeneity of resources. To unburden developers of application services from searching these solutions themselves, we started developing a generic software framework called SmartWare. SmartWare is a collection of principles and tools that are intended to simplify and accelerate the development of application services at the edge of the Internet and beyond. In this paper we describe the features of SmartWare, a prototype implementation and a test application we realized with the framework prototype.

Categories and Subject Descriptors

D.2.11 [**SOFTWARE ENGINEERING**]: Software Architecture - *domain-specific architectures, Patterns (e.g., client/server, pipeline, blackboard), Data abstraction*

General Terms

Management, Design, Reliability, Experimentation

Keywords

Autonomic Communication, Application Services, Service-oriented Architecture, Distributed Systems

1 INTRODUCTION

Small, wireless sensing and terminal devices change the face of computing environments. Their ubiquity, their richness in resources and function, and their configurability in purpose and behavior opens new ways for the design of distributed systems and services. We investigate the challenges and opportunities of service platform architectures at the edge of the internet, being neither always connected to global networking infrastructures nor completely detached in physical and intentional terms. In this context, wanted and unwanted disconnections of single devices or groups of devices from major management infrastructures represent the central problem to be solved. Creating a platform for intelligent, goal-driven user services, which tolerates the transient absence of a central control, means to give all constituting host devices the capability to autonomically cope with changes in the computing environment and user needs. With SmartWare we designed a platform to experiment with principles of autonomic communication at application level.

The description of SmartWare comprises two major parts, a model that describes a set of generic components, their purposes, and interworking, as well as a runtime environment for services compliant with the component model. In this regard, the definition the concept 'service' follows the definition of the Service-oriented Architecture (SOA) community. The SOA service pattern and the loose coupling of service consumer and provider have proven as appropriate for dynamic environments.

The remainder of this paper is organized as follows. Section 2 defines a number of requirements that were particular important for the definition of our SmartWare approach. An overview and the basic concepts of SmartWare are introduced in Section 3. Section 4 details the realization of SmartWare. Section 5 covers the show case "Pong Reloaded" that serves as an illustration of the depicted concepts. Related middleware approaches are described in Section 5 and a conclusion and an outlook are given in Section 7.

2 REQUIREMENTS

Looking at the topology of today's networks, neither a pure infrastructure based model nor a pure ad hoc model seems to be an appropriate assumption. On the one hand, the rapid emergence of small and portable computing devices entails highly dynamic network structures, where mobile devices meet spontaneously and establish ephemeral groups. On the other hand, wireless internet connections are increasingly spanning crowded places while the respective costs for accessing the Web are descending. In the following, those fluently merging networks we face today are referred to as *hybrid networks*, characterized by an increasing device availability and heterogeneity. Moreover, dynamic as well as infrastructure based device interaction has to be covered, while mobile devices should nevertheless be able to profit from phases of connectivity to infrastructure-based networks.

Thus, for developing the core of the SmartWare framework, four key requirements have been derived:

This work was supported by the EU within the framework of the BIONETS project EU-IST-FET-SAC-FP6-027748, www.bionets.eu

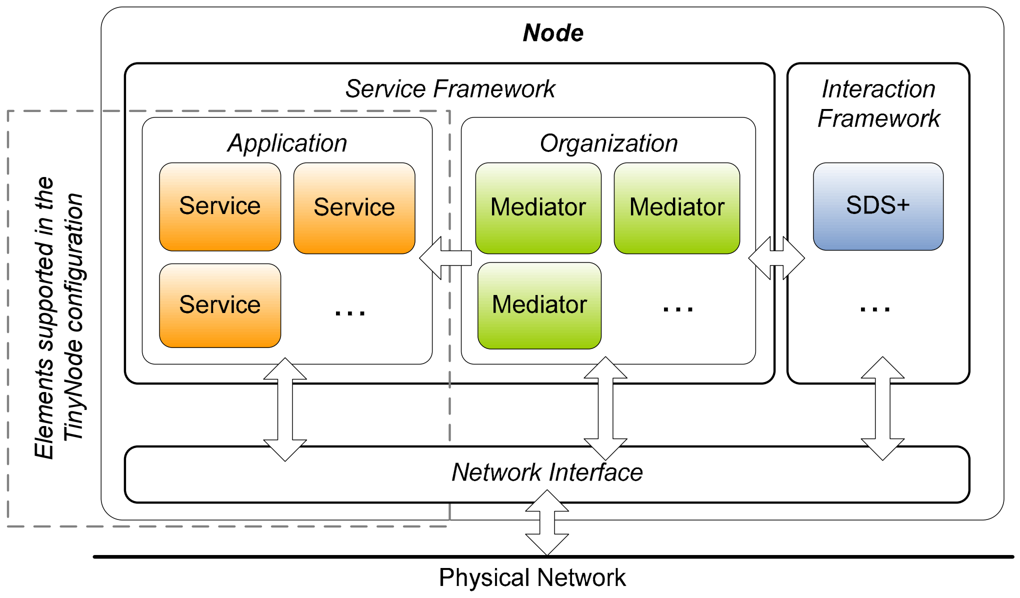

Figure 1: Overview of node components

First, the framework has to be extensible to make it adaptable to changing requirements. Some functionality is supposed to be provided by the middleware on-top, e.g., state management as mentioned above, but the need for additional features depends on the particular application domain. Furthermore, requirements will change due to new developments making new functionality available or existing functionality redundant.

Second, a lightweight solution and the regard for device resources to ensure support for multiple hardware platforms is an important requirement. Special attention needs to be put on resource-constrained devices.

Third, the support for the autonomic behavior of the devices is important in terms of accessing the own device and service state, bootstrapping mechanisms, or the full control of services and components.

Fourth, in the realization process standardized communication protocols need to be preferred to ease the connection to existing technologies and also benefit from future developments. Furthermore, addressing local and remote resources is to be supported in a uniform and unique manner also minimizing the communication overhead.

3 ARCHITECTURE

The architecture is described from two points of view, one is focused on basic components, the SmartWare Component Model, and the other explains the runtime system, the SmartWare Runtime Environment. The model introduces the basic framework elements application developers should use to design their applications and services. The Runtime Environment describes in detail which functions application developers can make use of when implementing their applications.

3.1 SmartWare Component Model

The SmartWare is built on the notion of abstracting physical and virtual devices by nodes. This abstraction is given in two configurations, the standard node configuration and the tiny node configuration. Latter is tailored to physically limited hardware equipment, i.e., it comprises less features. However, devices with standard node configuration may proxy advanced features for devices with tiny node configuration. The component model, which is depicted in Figure 1, describes the SmartWare framework from a structural point of view by introducing the kinds of active entities and their basic interrelation.

SmartWare distinguishes between three basic component types and respective containers: *Services*, management entities referred to as *Mediators*, and *Interaction Models*. While the Services realize application functionality, Mediators implement all environment-related organization tasks. According to the traditional service pattern, Services are utilized in an on-demand fashion. Thus, processes within Services are only triggered from outside and supposed to terminate when delivering a response. Basically, we distinguish between two kinds of Services, *device-dependent* and *device-independent* Services. Device-dependent Services are supposed to wrap functions particular to the underlying hardware platform. Device-independent Services are self-contained. Nonetheless, the same Service can be present on multiple nodes. The structure of the architecture benefits the on-the-fly deployment of Service to nodes and the migration of Services among several nodes. For this purpose, each Service is associated with a dynamic description, which outlines, e.g., its interface and functionality. This description can be regarded as a blackboard-like document, which can be flexibly modified or extended with updated information.

Furthermore, an important objective for the development of SmartWare was to provide maximum freedom in reconfiguration of Services. Therefore, the code base of device-independent

Services is accessible for reading and writing to all Mediators. These aspects are explained more in detail in the next section.

Service Mediators are autonomic management components. They take care for the fluent functioning of the entire environment while controlling the Service life-cycle including provision, discovery, selection, and execution. Additionally, Mediators may monitor the environment and make situation-based decisions with regard to Service reconfiguration or migration. Since most self-organization tasks need to be handled cooperatively in a distributed setting, an instance of a Mediator needs to be present on each node which is required to support a distributed management task. For example, to provide the environment-wide capability to discover or migrate Services, respective Mediators are required on each node. Mediators do not implement service-related application logic; they rather summarize basic management functions that are required by all Services.

The Interaction Framework is as container for communication middleware systems, which implement predefined patterns, the so-called interaction models such as Publish/Subscribe, DHT, or Semantic Data Space (SDS) [1]. The Interaction Framework is only available to Service Mediators and consequently not included in the service architecture configuration for tiny nodes. An instance of the Interaction Model is required on each node, in order to enable the usage of this Interaction Model by Mediators. The Interaction Framework allows abstracting the implementation of Interaction Models form the upper components like the Mediators, so that for each situation the most appropriate implementation can be chosen. The Network Interface enables the access to other nodes over the network.

3.2 SmartWare Runtime Environment

The SmartWare Runtime Environment realizes an extendable runtime framework for Services and Mediators. The Runtime Environment is build around an interpreter for only one programming language. For the concept explanation the actual syntax of this exclusively supported programming language does not matter. However, Mediators can be utilized to extend the Execution Environment with interpreters for other programming languages if required. The interpretable language supported by the Runtime Environment is supposed to be used for the implementation of Services and Mediators. Utilizing an interpreter for the execution of Service and Mediators simplifies the realization of platform-independence, benefits the integration of security models, and enables a flexible runtime management of Services (start, terminate and ship to other Nodes).

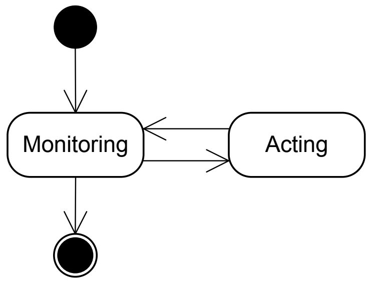

Figure 2: State model of Mediator

Mediators are initialized during the startup of the SmartWare node and usually remain active until the node is terminated. In contrast to Services they self-reliantly monitor their environment, i.e., the hosting node, co-located Mediators and Services, as well as Mediators and Services at neighboring nodes. In situations that require intervention, the Mediator autonomically interacts with its environment. For instance, if a node is going to run out of battery power while the user consumes one of its Services, a Mediator co-located on the same node may agree with a Mediator from a neighboring node to migrate this Service.

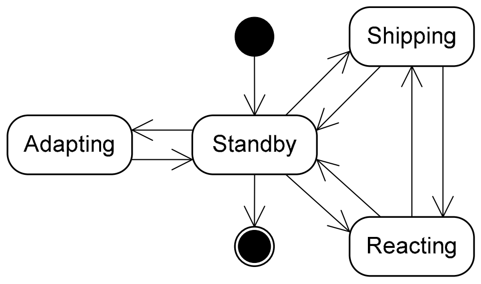

Figure 3: State model of Service

Services are also initialized during the startup of the SmartWare node and terminated on node shutdown. However, to enable dynamic migration of Services between nodes, the Runtime Environment also supports their ad hoc initialization and termination at runtime. After the initialization Services wait for requests. If a request is completed, the Service returns to standby state again. Services can migrate from one node to another, but the decision for migration is made by one or multiple Mediators. Hence, in contrast to mobile agents Services are not able to migrate themselves. SmartWare supports four kinds of Service migration:

1. Copy code base, description, and state; move copy
2. Copy code base and description; move copy
3. Move code base, description and state
4. Move code base and description; discard state

Device-independent Services expose their code base to Mediators for read and limited write access. Thus, Mediators can transfer a Service into a certain state for changing the Service's code base. By supporting code manipulation, SmartWare benefits advanced methods for adaption and improvement of Services at runtime. Moreover, the open code base is mandatory for the flexible extension of the Runtime Environment. A Mediator can perceive if the code base of a Service is not interpretable by the standard interpreter included with the Runtime Environment and care for the Service execution itself. For example, such a Mediator could implement another interpreter or ask for the required capabilities at neighboring nodes.

This aspect of SmartWare is, for example, utilized for the distributed execution of Service compositions. The code base of the composite Service is given as workflow-like description that refers to several types of Services. Mediators from different nodes coordinately analyze the composition description and broker co-located Services for the corporate execution.

The Execution Environment offers two sets of functions. The first function set grants access to most capabilities of the SmartWare node framework and is intended for usage by Mediators. The second function set is limited to the minimal set of capabilities required to realize a distributed user Service. Table 1 summarizes the functions accessible by Mediator and Services.

Table 1: Matrix of functions available for Services (column S) and Mediators (column M)

Function	S	M
User Interface Access enables the presentation of information to the user, while the means of presentation (e.g. text, graphical, audio) depend on the output capabilities of the node. Furthermore, this function allows obtaining data from input devices (e.g. keyboard, mouse).	x	x
Network Access supports interactions among multiple SmartWare nodes. The function realizes four basic operations on resources, namely create resource, read resource, update resource, and delete resource. This model is also well-known as CRUD.	x	x
Persistent Storage Access enables uniform, exclusive access to persistent storage of finite capacity in the SmartWare node. Services and Mediators may use this storage to file their state.	x	x
Service Life-Cycle Monitoring allows Services and Mediators to obtain information about activities affecting their life-cycle. If the SmartWare node is shutting down or a Mediator is going to migrate a Service, the affected entity gets this information in advance and may, e.g., persist its state, inform the user, or role back transactions.	x	x
Interaction Framework Access supports Mediators in the coordination of task among multiple SmartWare nodes. For this purpose the function offers access to various interaction models, which can be utilized according to the needs of the Mediators.	-	x
Service Life-Cycle Control allows Mediators to actively affect the life-cycle of Services through transferring them from one state to another.	-	x
Service Description Access grants read and write access to the description of Services. Mediators may append, rewrite, or delete parts of the Service description.	-	x
Service Code Base Access enables Mediators to read and manipulate the implementation of Services as needed.	-	x

4 REALIZATION

SmartWare was implemented for Java Standard Edition 6 and and a version for mobile, resource restricted devices (equivalent to Java SE 1.4). The network interface for this implementation bases on the Hypertext Transfer Protocol (HTTP). The system design follows the principles of the architectural style Representational State Transfer (REST) [12]. Thus, the SmartWare implementation is modeled with uniquely addressable resources that can be created, read, updated, or deleted through a standardized HTTP interface.

The implementation details of the underlying basic entities Nodes, Services, Mediators and Interaction Models, are given in the next subsections.

4.1 Nodes

As we already described in section 3.1, in the conceptual model of SmartWare nodes represent the basic abstraction from the underlying devices and are given in two configurations: Nodes, which represent the standard, fully-featured node configuration, and Tiny Nodes, which represent limited hardware equipment and therefore support only a subset of the SmartWare functionality.

Independently of the configuration type, each given individual node can be queried for information describing its current configuration. The query is executed by sending an HTTP GET request to the URI of that node. The available information includes the configuration type of the Node and the components currently deployed, that is, available Services (including their endpoints), Mediators and Interaction Models. Services can further be queried for their service descriptions; this procedure is described in more detail in the next subsection.

Tiny Nodes, which implement only a subset of the SmartWare functionality, are only capable of hosting Services. They do not support a Mediators or an Interaction Models directly. Instead, a proxy mechanism is applied, which allows fully-featured Nodes to integrate Tiny Nodes in their surroundings into the SmartWare infrastructure. Here, integration means that the Services hosted on the Tiny Node are made manageable by the Mediators available on the managing Node. For example, they can be made accessible to other nodes through a Mediator acting as a naming directory for Service lookups, or can be bound to a distributed execution of Services through a state management Mediator.

For this purpose, each Node periodically sends announcement messages including its identifier in its communication range. A Tiny Node receiving such an announcement message responds to its sender with a list of its hosted Services and stores the sender's identifier locally. The stored identifier is used to determine whether the Tiny Node is already managed by another Node, in which case the announcement message is ignored. Additional heartbeat and time-out mechanisms are applied to make sure that the information on managed Tiny Nodes and their respective managers is kept up to date in the network.

For interworking with other entities in a network, Nodes offer an HTTP interface that supports the operations summarized in the following table.

Table 2: HTTP interface of nodes

HTTP Request	Address	Returns
HTTP GET	Node URI in the form: *http://ip:port*	An HTML representation of the node's configuration type and components currently deployed, i.e., available *Services*, *Mediators* and *Interaction Models*

4.2 Services and Mediators

While in the conceptual model of SmartWare Services realize application functionality and are executed only on demand, Mediators are active components which implement and execute all environment-related organization tasks.

For both, Services and Mediators, a hot deployment/ undeployment mechanism is provided by the SmartWare Runtime Environment. The mechanism is invoked during the start up or shutdown of a node or during uptime of a node when a Java jar archive containing a Service or Mediator implementation is added to a reserved directory.

To support that mechanism, Services and Mediators have to provide the methods *initService(), shutdownService()* and *initMediator(), shutdownMediator()* respectively, where a proper initialization and finalization of deployed or undeployed components should be implemented. These methods are then automatically invoked by the SmartWare Runtime Environment.

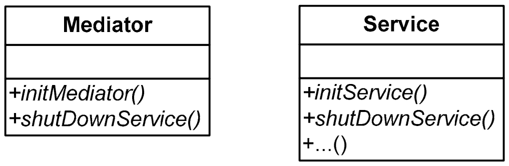

Figure 4: Mediator and Service interfaces

The SmartWare Runtime Environment does not only support queries for additional information about individual nodes, but also about individual services deployed on a particular node. For this purpose, service descriptions can be accessed through an HTTP GET request to the appropriate URI containing the identifier of the Service and the identifier of the description (as several service descriptions may be attached to a particular Service). So far, XML and text representations for service descriptions are supported. In the same manner descriptions can be changed or deleted or new descriptions can be added by issuing HTTP PUT, POST or DELETE requests accordingly.

Furthermore it is possible to access and manipulate the source code of a Service. The full HTTP interface supported by Services is summarized in the following table.

Table 3: HTTP interface of Services

HTTP Request	Address	Returns
HTTP GET	[Node URI]/Services/ [Service ID]/Descriptions? [xml | txt]	Xml or txt representation of all available descriptions of this Service
HTTP GET, DELETE, PUT, POST	[Node URI]/Services/ [Service ID]/Descriptions/[Description ID]	Read, delete, create or modify this description
HTTP GET, POST	[Node URI]/Services/ [Service ID]/Descriptions? Code	Get or modify the Service byte code
HTTP GET	[Node URI]/Services/ [Service ID]/Descriptions? Codebase	Get Service code base (only Java is supported so far)

SmartWare imposes no restrictions on the communication paradigm of implemented Services, though a REST framework developed by Fraunhofer FOKUS is integrated with SmartWare. The REST framework named RESTAC [14] is implemented in Java and aims at the rapid implementation of peer-to-peer applications by following a layered approach.

The basic layer of RESTAC is a communication layer based on HTTP which provides TCP-based HTTP unicast messaging and UDP-based HTTP unicast and multicast messaging. On top of this layer, RESTAC includes a layer for management of resources. In the cases of SmartWare these resources are all entities that are required for interworking across several nodes, for example Services and Service Mediators, but also service description and Service code base. An object wrapping model helps to make any Java object (entity implementations) quickly available as a REST-conform resource. RESTAC is released by Fraunhofer FOKUS to the public under the GNU Lesser Public License.

4.3 Interaction Models

Interaction models represent predefined interaction patterns provided by communication middleware systems such as for example Publish/Subscribe, DHT or Semantic Data Space (SDS). Interaction models are only available to SmartWare Mediators and consequently not included in the service architecture subset for tiny nodes.

The number of Interaction Models supported by a SmartWare node is not restricted. To manage the available models, an Interaction Model Framework is provided. A particular model is accessed by getting an instance of the Interaction Model Framework and calling the method *lookupInteractionModel*() which takes the name of the Interaction Model class as a parameter.

Each supported interaction model has to implement the methods *initModel*() and *getModel*(), where the first method is called automatically during the bootstrapping process of the node and instantiates the model, and the latter method should return a reference to the communication controller of the model. Figure 5 illustrates the basic interaction model interface.

A new model can be added to the SmartWare framework straightforward and with little effort. So far a distributed variant of the Semantic Data Space [1] was integrated into SmartWare.

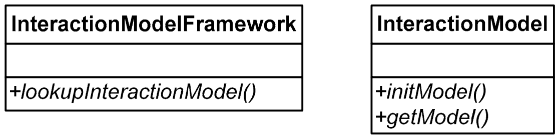

Figure 5: Interaction Model Interface

5 SHOW CASE: COLLABORATIVE SERVICE EXECUTION

The following scenario illustrates how the conceptual components of the framework can be utilized to control a cooperative Service execution across several devices:

Pong Reloaded is a distributed version of the popular game Pong, whereby parts of the game are provided by different Services running on different devices (Figure 6). In our implementation, 16 tablet PCs fixed into a rack collaboratively provide the gaming field, whereas two mobile phones act as remote controls to move the paddles for playing the ball. If one of the tablets is removed from the rack or switched off during the game, the gaming field adapts to the new conditions, i.e., the ball bounces off a cleared rack position or – in case that the ball is located on the removed device – the ball is “snatched away” from the game. By reinserting the tablet into an empty rack position the ball is “released” back into the gaming field. In terms of collaborative Service execution “snatching away” the ball means here that the focus of the Service execution has moved, and therefore an adaptation to the changed environment is necessary. Accordingly, a “released ball” represents the additional incorporation of devices that have become available.

To play *Pong Reloaded*, appropriate Services spread over different devices have to be found (like “displaying part of the gaming field” or “acting as remote control”), executed and supervised for runtime failures as for example caused by a removed device. Therefore two Mediators were realized which are reusable for other scenarios as well: a *Discovery Mediator* listing available Services and acting as a naming directory for Service lookups and a *State Management Mediator* controlling the execution flow of application related Services and signaling runtime failures or new detected devices to the application level by utilizing information provided by the *Discovery Mediator*.

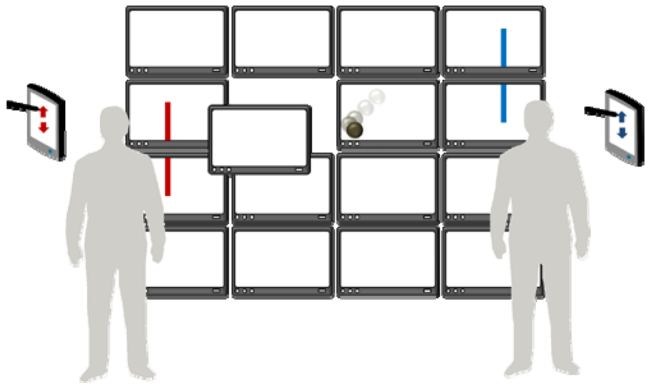

Figure 6: Pong Reloaded

For the implementation of Pong Reloaded we assumed that the description of the execution flow of the application is predefined as Service by the application programmer and thus available for the *State Management Mediator*. Envisioned extensions of the implementation are, for example, a *Composition Mediator* able to build Service compositions dynamically by mapping abstract user requests, such as “Playing Pong”, to available Services, or an extension of the *Discovery Mediator* using semantic annotations of Services to enhance the discovery mechanism.

Furthermore, a recovery mechanism for replacing lost Services by equivalent ones, such as replacing a lost remote control by another one, is part of future work.

6 RELATED WORK

The area of autonomic communication attracted the attention of many researchers; accordingly the efforts for SmartWare are not the sole in this area.

A method for hardware abstraction has been presented by Seshasayee et al.[9], who introduced a middleware for self-management and reconfiguration of nodes in an ad-hoc environment. They approach aims on the provisioning of a network that reliably reacts on physical changes as well as distributes load to accessible nodes.

However, in service-oriented environments, higher abstraction levels have been explored in order to support application developers. Within [2] an agent-based middleware has been developed that provides a transparent view on distributed sensor nodes. The middleware allows for cooperative data mining, self-organization and administration of sensor nodes providing an abstraction layer of the physical network for high-level application development. There have been many efforts in the field of sensor network abstraction, with some approaches aiming for high-level context provisioning [3][4] and other attempts tending to allow easy access to sensor data by covering the underlying physical network topology [5].

The intend to abstract the physical network layer and provide a simple application layer is also followed by [6], but with a slightly different view on composing network topologies to transparently use services beyond network borders. Therefore, so called composition agreements are formulated with the help of policies describing the requirements for the composed networks and the actual services running on top of them (e.g. QoS or security issues). A similar approach that includes context information into the self-organization process within the network layer has been developed by Malatras and Pavlou [7][8].

7 CONCLUSION AND OUTLOOK

In this paper we introduced a software framework called SmartWare for experimenting with autonomic communication principles. Here, the architectural entity Mediators encapsulates management functionality to control the life cycle of a Node’s Services. Additionally, an Interaction Framework enables the utilization of multiple interaction models to facilitate realization of a collaborative and autonomic behavior for Nodes. The explained concepts are illustrated by a show case called “Pong Reloaded”, a distributed version of “Pong” using 16 tablet PCs and two mobile phones as controlling devices.

Within the UST+ [13] project, the core framework of SmartWare is further advanced to support, for example, sophisticated service recovery mechanisms or the collaborative processing of user requests. Here, a middleware based on SmartWare is proposed for the distributed and autonomic service composition and provision with the help of service communities. Such communities are groups of services on different mobile or fixed devices that are characterized by:

- The sharing of complementary information or functionality,
- The common work towards a user-defined goal,
- The active information exchange,
- The interaction in a loosely coupled manner.

Here, multiple modules fulfilling a particular purpose such as state management or event processing where defined as a set of Mediators.

REFERENCES

[1] D. Linner, I. Radusch, S. Steglich, and C. Jacob, "The Semantic Data Space for Loosely Coupled Service Provisioning," in *Eighth International Symposium on Autonomous Decentralized Systems*, pp.97-104, 21-23 March 2007

[2] P. K. Biswas, and S. Phoha. A Middleware-Driven Architecture for Information Dissemination in Distributed Sensor Networks. 2004.

[3] H. Q. Ngo, A. Shehzad, S. Liaquat, M. Riaz, S. Lee. Developing Context-Aware Ubiquitous Computing Systems with a Unified Middleware Framework. 2004.

[4] C.-F. Sørensen, M. Wu, T. Sivaharan, G. S. Blair, P. Okanda, A. Friday, H. Duran-Limon. A Context-Aware Middleware for Applications in Mobile Ad-Hoc Environments. 2004.

[5] Y. Yu, B. Krishnamachari, and V. K. Prasanna. Issues in Designing Middleware for Wireless Sensor Networks.

[6] C. Kappler, P. Mendes, C. Prehofer, P. Pöyhönen, and D. Zhou. A Framework for Self-Organized Network Composition. 2005.

[7] A. Malatras, and G. Pavlou. A Practical Framework to Enable the Self-Management of Mobile Ad-Hoc Networks. 2007.

[8] A. Malatras, and G. Pavlou. Context-Driven Self-Configuration of Mobile Ad-Hoc Networks. 2006.

[9] B. Seshasayee, and K. Schwan. Mobile Service Overlays: Reconfigurable Middleware for MANETs. 2006.

[10] Z. Li, and M. Parashar. A Decentralized Agent Framework for Dynamic Composition and Coordination for Autonomic Applications. 2005

[11] S. Kern, P. Braun, and W. Rossak. MobiSoft: An Agent-Based Middleware for Social-Mobile Applications. 2006.

[12] R. T. Fielding, R. N. Taylor, "Principled design of the modern Web architecture," in *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, pp. 407-416, 2000.

[13] C. Jacob, H. Pfeffer, L. Zhang, and S. Steglich: Establishing Service Communities in Peer-to-Peer Networks. 1st IEEE International Peer-to-Peer for Handheld Devices Workshop CCNC 2008, Las Vegas, NV, USA, January 10-12, 2008.

[14] RESTAC. [Online] Available: http://developer.berlios.de/projects/restac. [Accessed: July 27, 2008].