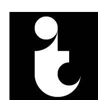
IT Licentiate thesis
2001-008

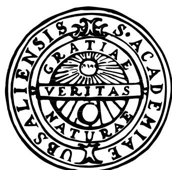
Personal Service Environments – Openness and User Control in User-Service Interaction

MARKUS BYLUND



UPPSALA UNIVERSITY
Department of Information Technology

SWEDISH INSTITUTE OF COMPUTER SCIENCE
SICS



UPPSALA UNIVERSITY

Personal Service Environments – Openness and User Control in User-Service Interaction

BY

MARKUS BYLUND

June 2001

COMPUTING SCIENCE DEPARTMENT
INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in Computing Science
at Uppsala University 2001

Personal Service Environments –
Openness and User Control in User-Service Interaction

Markus Bylund
markus.bylund@sics.se

Computing Science Department
Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

© Markus Bylund 2001
ISSN 1404-5117
Printed by Uppsala University, Tryck & Medier, Uppsala 2001

Abstract

This thesis describes my work with making the whole experience of using electronic services more pleasant and practical. More and more people use electronic services in their daily life – be it services for communicating with colleagues or family members, web-based bookstores, or network-based games for entertainment. However, electronic services in general are more difficult to use than they would have to be. They are limited in how and when users can access them. Services do not collaborate despite obvious advantages to their users, and they put the integrity and privacy of their users at risk.

In this thesis, I argue that there are structural reasons for these problems rather than problems with content or the technology per se. The focus when designing electronic services tends to be on the service providers or on the artifacts that are used for accessing the services. I present an approach that focus on the user instead, which is based on the concept of *personal service environments*. These provide a mobile locale for storing and running electronic services of individual users. This gives the user increased control over which services to use, from where they can be accessed, and what personal information that services gather. The concept allows, and encourages, service collaboration, but not without letting the user maintain the control over the process. Finally, personal service environments allow continuous usage of services while switching between interaction devices and moving between places.

The *s*View system, which is also described, implements personal service environments and serves as an example of how the concept can be realized. The system consists of two parts. The first part is a specification of how both services for *s*View and infrastructure for handling services should be developed. The second part is a reference implementation of the specification, which includes sample services that adds to and demonstrates the functionality of *s*View.

Keywords. Electronic services, personal service environments, user control, ubiquitous computing, user interfaces, mobility, personalization, service collaboration, component-based software engineering.

Acknowledgements

The work behind this thesis has been conducted during a period of more than three years and many people have played important roles in helping to shape the results. It would be impossible for me to mention all the people that have contributed and unfair to the ones that I would forget. I have therefore taken the liberty of selecting a few that I believe have played a particularly import role in the process.

Annika Waern has been my supervisor and mentor in both the work that led to the ideas behind this thesis, and the work with the thesis itself. Thank you for all help, patience, and belief in my work.

I would also like to thank my supervisor Arne Andersson at Uppsala University for valuable and encouraging comments on my work.

Fredrik Espinoza has played an important role throughout the work described herein, in particular during the early work with laying the groundwork of the design of *s*View – thank you for inspiration and many creative discussions.

Many thanks to the members of the HUMLE laboratory at SICS for providing a stimulating environment and numerous creative formal and informal meetings. Especially Kristina Höök who has commented upon and encouraged my work from an outside perspective ever since I started at SICS. I am also particularly grateful to Mikael Boman (a HUMLE alumni) and Anna Sandin for help with developing parts of the *s*View system, and Stina Nylander for proof reading and commenting upon my manuscripts.

My colleagues at the SICS Uppsala office have also helped to create an inspiring work environment, especially Per Mildner with his ever optimistic and encouraging comments.

Finally, I would like to thank my family: my father Torgny and my two brothers Andreas and Hannes, and in particular, my wonderful son Victor for making me realize what really matters in life and my beautiful wife Pia for sharing it with me.

Table of Contents

Abstract i

Acknowledgements iii

Table of Contents v

Preface 1

Overview 2
Background 3
The Papers 4
References 5

Paper A

M. Bylund and A. Waern, "Service Contracts: Coordination of User-Adaptation in Open Service Architectures," Personal Technologies, vol. 2, pp. 188-199, 1998. Reproduced with permission.

Paper B

M. Bylund and A. Waern, "Personal Service Environments – Openness and User Control in User-Service Interaction," SICS Technical Report T2001:07, Swedish Institute of Computer Science, Kista, Sweden, May, 2001.

Paper C

M. Bylund, "sView - Architecture Overview and System Description," SICS Technical Report T2001:06, Swedish Institute of Computer Science, Kista, Sweden, May, 2001.

Appendix I

Selected parts of the API documentation of the core sView specification.

Preface

Electronic services are gaining an increasingly broader acceptance, not only for professional use, but also for school and family activities as well as recreation and pleasure. Any functionality that can be mediated electronically can be seen as an electronic service. Examples include the watch on the desktop of your personal computer, but also the autonomous software agent [1] for automatically placing bids in electronic auctions, a family calendar on the screen fridge in your household kitchen, or your voice mail service in a cellular phone network.

However, service providers seldom focus on the complete situation of their users when designing and deploying electronic services. A calendar service on a PDA (Personal Digital Assistant) for example, is not the user's calendar in the first place, but rather the user's calendar on that particular device. The fact that the user probably needs to coordinate his or her activities with people without access to the calendar, possibly using other calendar services, is seldom catered for. For reasons like this, I see a need to work towards maximizing the *user's control* over the whole experience of using electronic services.

One of the most important factors in achieving user control is how users can access their services. Services must be available to their users whenever they wish to access them and on an interaction device at hand at that time. Sometimes that is when using a particular personal computer (which is stationary) or PDA (which is mobile). Some other time a service is best used from any computer, provided that it is equipped with a Web browser and an Internet connection. In any case, we better provide services that can be reached from many different types of devices and with varying network connectedness, or else the users' possibilities to be in control of their usage will be limited.

Ubiquitous access to services is however of little use if the services must be restarted each time the user switches to another device. For example, a user might issue a search for cheap airline tickets on a stationary personal computer. It takes a while for the results of the search to appear and the user needs to catch a bus. It would be useful if the user could access the search results via a cellular phone from the bus instead of waiting by the personal computer. This example illustrates a need to support continuous usage of services even when switching between interaction devices and moving between places.

Another requirement on user control is that the user can choose which services to use. The World Wide Web is exemplary in this respect since any user can connect to any Web page on the Web [2]. At the same time anyone can publish his or her own Web page for everyone else to browse. In the context of electronic services, this issue is really about openness – in order to provide the user with a free choice of which services to use, the infrastructure for electronic services (as well as the devices that mediate services) must be open to any service provider.

A fourth issue concerns personalization of services. Electronic services typically gather and store large amounts of information about their users. Sometimes this information is vital for the service in question. An online bookstore for example,

needs to know the addresses of its users in order to deliver the purchased goods. In some cases however, this information is not collected for the benefit of the functionality of the service, but only for the purpose of making money (e.g. by gathering and selling demographic information). Two user control issues are related to these kinds of information gathering. Firstly, to maintain user privacy and integrity the service should allow the user to decide and control whether or not it is allowed to collect personal information or share information with other services [3]. Secondly, when the set of services grows, it becomes unmanageable to inspect and modify information about oneself for each and every service. One way to approach a solution to both of these problems is to provide a central access point (i.e. central with respect to the user) for personal information. This would allow the user to control which services that have access to what information. It would also allow the user to inspect and modify personal information in one place for all services.

A central access point for personal information is one example of how services can collaborate and share information to provide richer user support. Many other examples of where two or more services can benefit from collaborating can be found. For example: a calendar service can collaborate with a meeting booking service, a map service can collaborate with a phone registry service, a payment service can collaborate with any service that requires payment, etc. But which service should be allowed to collaborate and about what? This is yet another issue that the user should be allowed to control.

Overview

This thesis describes a novel approach to give users of electronic services more control. The approach is based on the concept of *personal service environments*, which are storage and execution environments for electronic services that are private to individual users. The personal service environment is open in the sense that any service provider can develop service components, which any user can put in his or her environment. Within the service environment, service components can collaborate about content and functionality provision. By collecting all services of an individual user in one place, the approach opens for solutions to manage personal information (the personal service environment becomes a natural place for storing personal information) and to control how services collaborate.

The service environment is in itself mobile, and it can follow the user as he or she moves between computers and devices. As the environment migrates, the services stored in it follow. This reduces the dependency on network connectivity for usage. In many cases, the service environment can execute locally on the interaction device, completely removing the need for a network connection. Local execution also allows the use of more powerful user interfaces such as Graphical User Interfaces (GUI). Remote user interfaces, such as HTML and WML browsers, are typically less expressive and powerful. However, the personal service environment allows interaction via any type of user interface. This makes services available to the user even when he or she does not have direct access to an interaction device that is powerful enough to run the complete service environment. In such a case, the service

environment has to execute on a network-based server that users can access from e.g. public Web kiosks and thin clients such as cellular phones.

During migration of a personal service environment, the services that are stored in it are offered to save their state. This allows the user to start a session with a service on one computer, suspend the interaction and move to another computer or device while bringing the service environment, and finally resume and continue the interaction exactly where it was suspended. With this, the approach also provides continuity.

The concept of personal service environments has been implemented in the *s*View system, which is a Java based specification and implementation for electronic services. The system consists of two parts. The first part is a specification of how to develop service components for use in the system. It also specifies how to develop the infrastructure that is required in order to store and execute the service components. This specification allows anyone to develop both service components and infrastructure for handling personal service environments.

The second part is a reference implementation of the specification. The implementation provides a server that is capable of storing and executing personal service environments. It also allows service environments to migrate between servers. A number of sample services (a calendar, an e-mail client, a payment service, etc.) and utility services for handling different types of user interfaces (GUI, HTML, and WML), user preferences, etc. are also included.

The *s*View system is freely available for download from http://sview.sics.se/.

Background

The ideas behind the concept of personal service environments, and later the *s*View system, emerged from experiences of several previous research efforts.

The first of these projects, the KIMSAC project [4], concerned presentation coordination in the context of public multi-service information kiosks. This work focused on how to coordinate the presentations (to the user) of a set of independent software agents in one single user interface. This resulted in the development of the sicsDAIS system [5, 6], which is a system that presents a user interface, to which agents can add user interface components. The system includes functionality for coordination of the different agents' interface components. The development of the sicsDAIS system taught us the importance of coordinating the user interfaces to independent but related functionality.

In the KIMSAC project we also worked with content adaptation [7]. The task was to coordinate how independent software agents, which worked in a shared domain for the same user, adapted the content of their presentations to the skills and usage history of the user. In this project, presentation coordination was managed by a separate personal assistant agent. This solution proved overly complex, as what really was needed was a common locale for storing information about the user. Information that could be added, modified, and shared between several agents simultaneously, at the same time as the user had a reasonable chance of inspecting and modifying the information. This experience from KIMSAC resulted in the first of the research papers included in this thesis [8]. The focus in this paper is on basic requirements on service

collaboration, requirements that have influenced the design of the personal service environment concept as well as the sView specification.

The EdInfo project [9] explored how to utilize the knowledge and expertise of individual users when performing complex information filtering tasks, and how an information system could be designed to support different user roles in this process. The ideas were implemented in the ConCall system, a system for distribution and filtering of conference calls. The system supports two user roles: expert editors and consumers of conference calls. ConCall was implemented using mobile agent technology [10] and it shares many properties with the sicsDAIS system and the support for coordination of content adaptation in the KIMSAC project. The use of mobile software agents for these kinds of systems was evaluated and it was found to have its merits, but not to the extent that the overhead of using it is justified [11].

Having worked with autonomous agents and mobile agent systems in several projects, we felt a need to broaden our view on our primary abstraction for software development. In some situations we found the autonomous agent concept limiting – especially when we designed software that could not easily be categorized as autonomous or as having beliefs, desires, and intentions [1]. Instead we adopted the concept of electronic services, which very well may include autonomous as well as mobile agents.

In isolation, the different activities in the KIMSAC and the EdInfo projects were successful. However, taken together the results of these activities suffered from the same problems as I claim that electronic services suffer from today. The systems that were developed were limited to one or only a few access systems. They did not collaborate despite the fact that they explored similar domains and very well could benefit from collaboration. From the users' point of view they were isolated units of functionality that did not take the user's complete situation into account. The identification of these problems is what triggered my work with this thesis.

The Papers

This thesis consists of three scientific papers, A through C, which are summarized below.

Paper A [8] represents early work on the ideas behind the concept of personal service environments and the *s*View system. The paper introduces a particular type of service contracts in an agent architecture. These are described as mutual agreements on collaboration between software agents. The paper includes an example of how the technique can be applied as a base for coordination of adaptation of content towards the user. The material presented in this paper has influenced the work on personal service environments. The paper describes a classification of service architectures, Open Service Architectures (OSA), which includes most of the properties that we later refer to as openness. Neither personal service environments nor the *s*View system implements the ideas of service contracts or content adaptation coordination. However, a personal service environment is a sound locale for such functionality for two reasons. Firstly, it provides functionality for performing the kind of communication that is needed in order to specify service contracts. Secondly, it

provides a locale in which information about the user can be collected and stored for the purpose of performing adaptation of content towards the user.

Paper B [12] introduces the concept of personal service environments. The concept is motivated in terms of openness and user control, and further by more detailed requirements on heterogeneity, extendibility, accessibility, adaptability, and continuity. The concept is compared to alternative approaches such as the World Wide Web (with extensions) [2, 13, 14], Mobile Agent Environments [9-11, 15-18] and a few other systems for electronic services [19-21]. The paper also contains a description of the *s*View system as an example of how the concept can be implemented, as well as examples of our experiences with using *s*View in several research activities [22-27].

Paper C [28] is a technical description of the *s*View system. The design of *s*View is motivated in terms of openness, in particular by requirements on heterogeneity and extendibility. The architecture of the system is described as being composed of two main parts: a core specification and a reference implementation. The main part of the paper is a detailed description of the core specification. The reference implementation is also described in brief.

References

[1] J. M. Bradshaw, *Software Agents*. Menlo Park, CA: AAAI Press/MIT Press, 1997.

[2] T. Berners-Lee, R. Caillau, J.-F. Groff, and B. Pollermann, "World-Wide Web: The Information Universe," *Electronic Networking: Research, Applications and Policy*, vol. 2, pp. 52-58, 1992.

[3] E. Volokh, "Personalization and Privacy," *Communications of the ACM*, vol. 43, pp. 84-88, 2000.

[4] P. Charlton, Y. Chen, F. Espinoza, A. Mamdani, O. Olsson, J. Pitt, F. Somers, and A. Waern, "An Open Agent Architecture Supporting Multimedia Services on Public Information Kiosks," presented at Practical Applications of Intelligent Agents and Multi-Agent Systems, PAAM'97, London, UK, 1997.

[5] F. Espinoza, "sicsDAIS: A Multi-Agent Interaction System for the Internet," presented at WebNet 99—World Conference on the WWW and Internet, Hawaii, 1999.

[6] F. Espinoza, "sicsDAIS: Managing User Interaction with Multiple Agents," Ph.Lic. thesis, The Royal Institute of Technology and Stockholm University, Stockholm, 1998.

[7] M. Bylund, "Coordinating Adaptations in Open Service Architectures," M.Sc. thesis, Uppsala University, Uppsala, 1999.

[8] M. Bylund and A. Waern, "Service Contracts: Coordination of User-Adaptation in Open Service Architectures," *Personal Technologies*, vol. 2, pp. 188-199, 1998.

[9] A. Waern, M. Tierney, Å. Rudström, and J. Laaksolahti, "ConCall: An information service for researchers based on EdInfo," Swedish Institute of Computer Science, Kista, T98-04, 1998.

[10] J. E. White, "Mobile Agents," in *Software Agents*, J. M. Bradshaw, Ed. Menlo Park, CA: AAAI Press/MIT Press, ISBN 0-262-52234-9, 1997, pp. 437-472.

[11] M. Tierney, "ConCall: An Exercise in Designing Open Service Architectures," Ph.Lic. thesis, The Royal Institute of Technology and Stockholm University, Stockholm, Sweden, 2000.

[12] M. Bylund and A. Waern, “Personal Service Environments – Openness and User Control in User-Service Interaction,” Swedish Institute of Computer Science, Kista, Sweden, SICS Technical Report T2001:07, May, 2001.

[13] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, “Simple Object Access Protocol (SOAP) 1.1,” World Wide Web Consortium, W3C Note 27 July 1999, May 8, 2000.

[14] A. Di Stefano and C. Santoro, “NetChaser: Agent Support for Personal Mobility,” *IEEE Internet Computing*, vol. 4, pp. 74-79, 2000.

[15] N. Minar, M. Gray, O. Roup, R. Krikorian, and P. Maes, “Hive: Distributed Agents for Networking Things,” presented at First International Symposium on Agent Systems and Applications, Third International Symposium on Mobile Agents featuring the Third Dartmouth Workshop on Transportable Agents, Rancho Las Palmas Marriott’s Resort and Spa, Palm Springs, CA, 1999.

[16] C. Pullela, L. Xu, D. Chakraborty, and A. Joshi, “A Component Based Architecture for Mobile Information Access,” Department of Computing Science and Electrical Engineering, University of Maryland Baltimore County, Technical Report, TR-CS-00-05, March 31, 2000.

[17] H. L. Chen, “Developing a Dynamic Distributed Intelligent Agent Framework Based on the Jini Architecture,” M.Sc. thesis, University of Maryland Baltimore County, Baltimore, 2000.

[18] T. Sandholm and Q. Huai, “Nomad: Mobile Agent System for an Internet-Based Auction House,” *IEEE Computer*, vol. 4, pp. 80-86, 2000.

[19] “OSGi Service Gateway Specification Release 1.0,” Open Services Gateway Initiative, May, 2000.

[20] “Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); Mobile Station Application Execution Environment (MExE); Functional description; Stage 2,” European Telecommunications Standards Institute, ETSI TS 123 057 v.3.0.0, January, 2000.

[21] “Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); Mobile Station Application Execution Environment (MExE); Service description; Stage 1,” European Telecommunications Standards Institute, ETSI TS 122 057 v.3.0.1, January, 2000.

[22] M. Boman, “Implementing services for a PSE,” M.Sc. thesis, Uppsala University, Uppsala, Sweden, 2000.

[23] F. Espinoza, P. Persson, A. Sandin, H. Nyström, E. Cacciatore, and M. Bylund, “GeoNotes: Social Filtering of Position-Based Information,” Swedish Institute of Computer Science, SICS Technical Report T2001:08, May, 2001.

[24] H. Nyström and A. Sandin, “Social Mobile Services in an Open Service Environment - an Overview, Analysis and Implementation,” M.Sc. thesis, Uppsala University, Uppsala, Sweden, 2001.

[25] P. Persson, F. Espinoza, and E. Cacciatore, “GeoNotes: Social Enhancement of Physical Space,” presented at CHI'2001, Seattle, WA, 2001.

[26] S. Nylander and M. Bylund. “Providing Universal Device Access to Mobile Services,” Unpublished manuscript, available at: http://sview.sics.se, 2001.

[27] F. Espinoza and O. Hamfors. “ServiceDesigner: Enabling End-Users Access to Web Services,” Unpublished manuscript, available at: http://sview.sics.se, 2001.

[28] M. Bylund, “*s*View - Architecture Overview and System Description,” Swedish Institute of Computer Science, Kista, Sweden, SICS Technical Report T2001:06, May, 2001.

Paper A

Service Contracts: Coordination of User-Adaptation in Open Service Architectures

Markus Bylund and Annika Waern

Swedish Institute of Computer Science, Kista, Sweden

Abstract: An Open Service Architecture (OSA) is a framework that supports an open set of users to subscribe to, and possibly pay for, an open set of services. Today, the World Wide Web (WWW) is the most successful example of an OSA. Nevertheless, the WWW provides poor support for personalised services, since services cannot collaborate unless handcrafted to do so. We present a framework that allows independent, personalised services to coordinate their adaptations to individual users. The framework is described in terms of service contracts in an agent architecture. We first describe the general notion of service contracts, and then the particulars of service contracts used for adaptation coordination. Adaptation coordination addresses a crucial issue for OSAs: that of providing users with homogeneous interaction with heterogeneous services. We suggest that this is done by introducing a separate adaptation coordination agent, which orchestrates how the individual services are personalised.

Keywords: Adaptation coordination; Agent-oriented programming; Open service architectures; Service contracts; User adaptive services

1. Introduction

The computer systems of today are no longer stand-alone programs developed for a closed group of users or professionals in a well-known organisation and environment. Instead, computer systems are turning into computer services, available to a large, distributed and heterogeneous user group. In the same way, the individual user is no longer faced with an individual computer system, but with a vast and perpetually changing array of computer services. Services move between platforms: they are no longer bound to the individual desktop computer, but run on information kiosks, portable and wearable devices, mobile phones and the home TV. This requires an *Open Service Architecture* (OSA): a framework in which an open set of users can access and interact with an open set of services.

A critical issue for OSAs is their integration. An OSA forms a particular kind of complex system, where components are developed independently of each other. Unless there is any kind of integration of components, the individual user will be faced with a multitude of interaction metaphors and interfaces, that must all be understood and learned. The situation is worsened rather than eased by personalisation, since every service as well as platform may provide different ranges of adaptations and different means for accessing them.

In this paper, we address this issue by proposing to view the components of an OSA as agents rather than as stand-alone systems. The key feature here is that agents are not integrated in the classical sense; rather, they are constructed so that they can adapt to and collaborate with other agents. We propose that this collaboration is done within the framework of negotiating and executing a service contract: explicit or implicit descriptions of how agents are to collaborate.

2. Open Service Architectures

An Open Service Architecture includes and supports three necessary components:

- an open set of users,
- an open set of services,
- a means for users to access the services.

There exist today many simple examples of OSAs. The de facto standard OSA is the World Wide Web (WWW), that in its purest form provides nothing but this: users can access a service by navigating to it or by typing its web address. Services can be anything from pure web pages, to search engines, or market places where vendors present goods that users can both buy and pay for through the system.

The simple structure of the WWW has several advantages. The biggest advantage is that each information provider, vendor, or web service provider can make himself or herself available without collaborating with anyone else. They may gain

© Springer-Verlag London Ltd
Reproduced from Personal Technologies (1998) 2:188-199, with permission.

advantages from collaboration, but collaboration is not necessary. Technically, there are very few standards to adhere to, essentially, basic knowledge of hypertext mark-up language (HTML) is all that is needed to go online.[1]

The disadvantages of the WWW are also mainly caused by the fact that services are independent of each other. If there is to be any kind of collaboration between services, it must be handcrafted by the people developing them, and it can only be maintained if humans agree to coordinate their further development and maintenance of the services. The simplest example of this problem is stale links, but search engines, market places, and other brokering services run into similar problems. One effect is that it is almost impossible to develop subservices made available to other services, such as translation components, databases of video clips, user modelling capabilities, etc. There are also many problems with the WWW that occur because the WWW protocols were originally developed for information presentation and not as a generic OSA platform. This has lead to problems with payment schemes, secure identification, and the user's control over their user profiles and usage statistics.

The problems with the WWW show that there is good reason to attempt richer approaches to open service architectures. These will not, of course, replace the WWW but they can live in parallel and within the WWW to provide better support for service interaction. The immense success of the WWW shows that it has some properties that should not be disregarded in constructing an agent-based OSA. In particular,

- developers must be able to develop services independently of each other, adhering to a minimum of very simple and clear standards,
- services should be identified with who provides them.

2.1. Agent-oriented programming for open service architectures

The notion of agency in computer science literature is not one uniform idea. In his introduction to the book *Software Agents* [1], Jeffrey Bradshaw identifies two motivations for the recent interest in agents in computer science: the search for novel interaction metaphors, and the need for new programming paradigms for large and open systems. The first has spurred work on visible interface agents that maintain a dialogue with the user, as well as software agents to which the user can delegate some of his or her more tedious tasks. The latter has concentrated on what Shoham [2,3] called Agent-Oriented Programming (AOP): techniques for constructing software architectures where the individual components are treated as if they had mental qualities such as beliefs, intentions, and desires, and collaborate with each other by communicating these.

This paper is concerned directly with the second issue: that of creating programming paradigms for large and open systems. We propose that the OSA be realised as a collaboration between *service* and *user* agents. It is simple to see that these can be viewed as entertaining beliefs, desires, and intentions.

- Service agents will maintain beliefs concerning the services it can provide. User agents will, in turn, maintain beliefs that have to do with the individual users: their characteristics, tasks and work environment.
- A user uses a service only if it provides some added value for him or her. Similarly, a company offers a service only if it can gain something from this. These objectives can be modelled as the desires of the service and user agents.
- The intentions of agents correspond to the collaborations that they are currently involved in. In this paper, we describe these in terms of service contracts. An agent can entertain two kinds of intentions: an intention to form a service contract, or the intention to complete a service contract.

In the KIMSAC project [4], we investigated the use of current agent technology for developing an OSA. Although this experiment was partly successful, it showed that building truly flexible and robust OSAs is a major challenge to agent technology. In particular, current agent technology gives little support for setting up or negotiating services to a user or another service, as opposed to just routing individual queries to a competent service.

A more recent example is the EdInfo system [5,6]. EdInfo is an information service system developed at the Swedish Institute of Computer Science. The current application of EdInfo ConCall deals with calls for papers and participation for forthcoming conferences. It contains two services: the ConCall service that routes information about calls to forthcoming conferences to individual researchers

[1] As with all new means of human communication, WWW has spurred novel social conventions and even fashion trends. Most developers will also aim to adhere to such conventions.

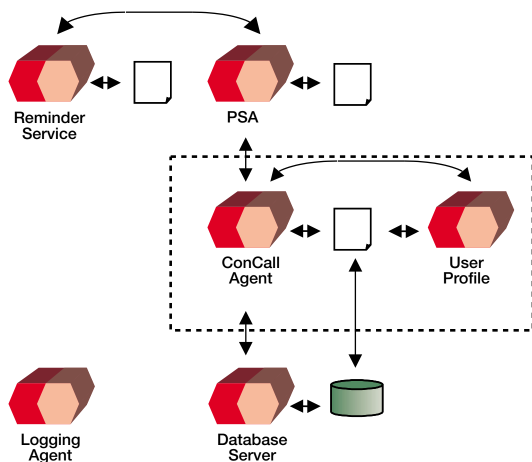


Fig. 1. The ConCall service architecture.

based on their interest profiles, and a reminder service that users can use to set up reminders about conference deadlines as well as other relevant dates.

The conceptual architecture of EdInfo is shown in Fig. 1. Most of the agents, the Personal Service Assistant (PSA), the Reminder Service, the User Profile and the ConCall agent, are personal to the user. They hold information about the state of the current session with the user, and may hold information about the user and store it between sessions. The Personal Assistant is the user's representative in the agent architecture, and is used to help the user to select services. The reminder agent and the ConCall agent are the user's personal "copies" of generic services, which other users also may subscribe to. The database agent is used by multiple users, and it does not store user-specific information. Agents communicate with each other by Knowledge Query and Manipulation Language (KQML) messages. The ontologies used are application specific. Users communicate with agents through special-purpose applets.

The EdInfo system is a very good example of the kind of service architectures that we want to accomplish. All agents in EdInfo have at least two interfaces: one towards users, and one towards other agents. Some of the agents maintain personalised information and adapt their interaction to it: the PSA agent maintains information about which services the user subscribes to, the User Profile agent maintains information about user preferences, and the Reminder Agent maintains information about the user's reminders. A particularly interesting functionality is the User Profile agent, which maintains a model of the user's preferences. This model could potentially be shared between different services. The only reason that it is not is that the EdInfo system holds no other service that needs to be adapted to user preferences. For example, it could be shared with an alert service, which would scan the WWW for new documents and reports in the user's area of interest. This would turn the profile agent used in ConCall into a special-purpose *adaptation coordination agent*. In Section 4, we will discuss the generic concept of adaptation coordination agents in detail.

EdInfo is still not an OSA. There are no means for users to seek for and subscribe to new services, or for the ConCall service to utilise any other reminder service than the one that is built-in. In future work on the OSA platform underlying EdInfo, we aim to achieve a way for services to be set up automatically or semi-automatically.

As an example, let us go through an imagined scenario for how a user could initiate subscription to the ConCall service.

- The user instructs the PSA to contact a facilitator service to seek a service that collects information about conference calls.
- The facilitator suggests the ConCall service.
- The ConCall service spawns a personal ConCall agent for the user. Its initial intention is to form a service contract with the user. It proposes a service contract to the PSA, indicating what the service costs, how the service works, and what information it needs from the user in order to perform its services. In addition, it explains that it cannot keep track of reminders – a reminder service is needed for that.
- The user agrees to the deal, under the provision that the user profile cannot be passed on to other agents. (If he or she had declined the contract, the personal ConCall agent would be deleted at this point.)
- The ConCall agent agrees to the deal, including the restriction on the user profile information, and provides a reference to a particular reminder service.
- The user may now register with the reminder service in the same manner, or ask the facilitator agent for an alternative service that can provide the same functionality.

3. Service Contracts

A *Service Contract* is a mutual agreement between a group of agents, which describes how they

collaborate. It can be partly implicit (hard-wired into the construction of agents and in the basic communication infrastructure), and partly explicitly negotiated and agreed upon. Agents may also negotiate service contracts at multiple levels: they may negotiate a service in itself, and they may negotiate how to negotiate a service. The important realisation is that each scheme for negotiation is by itself a service contract, and that this object/ meta level distinction need not stop at two levels.

At a very basic level, service contracts always exist. All languages for Agent-Oriented Programming are based on the assumption that agents adhere to certain basic principles. KQML agents are assumed to be benevolent and to provide answers to queries if they can [7]. Agent communication language (ACL) agents are committed to answering requests from facilitators that they register with [8], and Contract Net agents are committed to a particular protocol for requesting and accepting task allocations [9]. Within these basic collaboration principles, agents engage in collaboration and may negotiate details of their collaboration. Thus, agents are bound not only by the basic collaboration principles, but also by individual or mutual intentions or "deals" that they commit to based on their interactions with each other, and with external resources.

Note that agents may *break* service contracts, even implicit ones. Agents may be poorly constructed and unable to realise the conventions, or agents may deliberately flaunt collaboration, defecting from what was agreed upon to gain advantages for themselves or their owners. Furthermore, if agents engage in explicit negotiation, there is always the risk that they do not have a mutual understanding of what has been agreed. Agent architectures typically must contain some means of protection against misunderstandings, as well as incompetent or malevolent agents [10]. In this paper, we are not so much concerned with how agents are to protect themselves from such behaviour, but how to recognise it; what are the commitments that agents need to be able to make, and in which ways can agents break their commitments?

3.1. Elements of service contracts

Service Contracts can be used both to describe an actual service, to describe methods for service negotiation, and even to describe methods for negotiating ways of negotiation. But independently of what the service contract is used for, it must convey sufficient information to allow all participating agents to agree on two critical questions:

1. "Do we want to collaborate?"
2. "Can we collaborate?"

The first question involves comparison of the contract and the agent's own goals. The second question breaks down into several components, dealing with how the collaboration will be carried out. We can view service contracts as comprising at least five items of agreement, all of which could be subjects of negotiation:

- *Why:* Do we want to collaborate?
- *What:* Do we have anything to collaborate about?
- *How:* How should we use these competencies?
- *When:* What information do we need to exchange, and when?
- *Language:* What language should we use in the information exchange?

The list is not necessarily exhaustive; it stems from an analysis of the needs that arise in open service architectures. Below, we analyse each of the items and discuss why it is included. This discussion is only intended to give an informal understanding of what is involved in defining a service contract. In Waern [11] we go into some detail on how service contracts can be formulated in a semi-formal manner. A full formalisation of the notion of service contracts is lacking, but Verharen et al. [12] report a similar approach based on deontic logic.

Do we want to collaborate? Agent architectures can be built around the notion of self-interested agents. This is the natural construction for an OSA, where each agent is put online by some particular person or organisation, and can be seen as serving the interests of that person or organisation. In such architectures, agents will collaborate only if they get something out of it. This notion of self-interest has sometimes been seen as a defining characteristic of agents ("objects do it for pleasure, agents for money").

Do we have anything to collaborate about? Even in a collection of benevolent agents, collaboration will occur only if at least one agent finds a need to consult other agents. This is best analysed at task level: there are some tasks that one agent want to achieve, but that it either cannot do, or believes that some other agents can do better than itself. This agent can then take the initiative to negotiate a service contract with other agents.

How should we use our competencies to collaborate? In order to enable collaboration, the agents may tailor their behaviour based on information from each other. In the simplest case, this negotiation becomes part of the service itself. A travel agent, for example, needs information about my needs (travelling to Spain) and preferences (no stopover in Copenhagen) to set up a travel arrangement for me.

A more advanced example is the tailoring of general knowledge to a specific domain. Much work in artificial intelligence (AI) has gone into building expert system shells, truth maintenance systems, user modelling shells, etc., systems that provide a generic reasoning capability. These systems could potentially be encapsulated into agents that can provide reasoning capabilities for other agents, provided that they are equipped with the appropriate domain knowledge when the service is set up. In Section 5, we will discuss the usage of service contracts in tailoring the competencies of an adaptation coordination agent. In this domain, the agent's user modelling capabilities need to be tailored to the different domains that the subscribed services deal with.

What information do we need to exchange, and when? In some applications, the whole service contract negotiation leads up to a single transaction; this is the prevailing case in computational market architectures [13]. In service architectures, the product of negotiation is more complex and concerns a service that is to be provided over a period of time. This service will require some information exchange. In the simplest case, this exchange is unidirected, such as a Service Provider telling a user agent that a certain situation has arisen (the "tell me when I receive mail" case). A more complex example is the interaction schemes discussed in the adaptation coordination example (Section 4).

Which terminology should we use in the information exchange? A quite common assumption for agent communication languages is that agents need to share, not only the language in which to express ontologies, but also some domain-specific ontologies. This requirement can be lessened by meta-level negotiation of an object-level ontology. Agents can both negotiate to select an ontology they commonly understand, or exchange information so that some of them learn new ontologies that will be used in subsequent interactions. In particular, this occurs when a generic agent tailors its services to a specific domain. The information it accommodates will not only allow it to make inferences in a new domain, but also to communicate its inferences in a domain terminology that it did not know in advance.

4. Analysis of Adaptation Coordination

When moving from individual computer systems to OSAs, the task of adapting system functionality, dialogues and interfaces to the user becomes truly complex. There are several reasons for this:

- *Short life cycle:* individualising software takes time and requires effort, either from the user or from the system. The user might not want to spend the time required to individualise a service manually if it is only to be used once and for a short period. In addition, services might not get the amount of interaction with the user that is required for automatically performing adaptations.
- *Many entities to adapt:* with a multitude of services to choose from, and frequent collaboration between services, the total number of entities that the user encounters will be large. Due to the high workload, the user cannot possibly be expected to individualise new services manually if such are encountered frequently.
- *More of the same:* the information that adaptations are based on is expensive to gather. This is true whether it is done automatically by the system or if the user enters it manually. However, this information is not always unique to a specific service. A user's preferred language is an example of a preference that is likely to be stable across all services of the user. If adaptation is to be made individually for each service, a lot of work will be duplicated.
- *Protection of information:* the user might not want to share all information that is needed to perform an adaptation with the services. How to reduce that need for sharing of information while still performing adaptations is also a challenge.

In the following, we will argue for a solution to these problems by introducing an adaptation coordination agent. Such a component could be used to reuse user information details from services that the user previously has subscribed to. It could also constitute a uniform interface for controlling adaptations made by multiple services, in order to reduce the user's efforts. Lastly, it could hide information about the user from services, information that is needed when performing adaptations.

Following is an example of a very simple adaptive user interface in an OSA; namely, the adaptation of coloured links in WWW browsers. The example highlights many interesting aspects of adapting information in OSAs, and throughout the analysis it will be referenced repeatedly.

Example

Most WWW browsers automatically adapt the colour of links to whether the user has followed it or not. The colour might, for example, be blue if the link has never been followed (or if a certain period has lapsed since it was most recently followed), and red after the user has explored it. Several ways to modify the adaptation exist, one being that of letting the user select the colours representing followed and not followed links, respectively. A second is to let Service Providers define colours of their own, overriding the user's settings. A third way to modify the adaptation could simply be to let the user decide whether Service Providers should be allowed to override link colour settings or not. □

4.1. Actors

Categorisations of actors in adaptive systems frequently occur in literature, either explicitly or implicitly [14, 15]. However, they fail to address the issue that is central to OSAs, the support of an open set of both services and users. We therefore suggest three main categories of actors in OSAs: *Users*, *Service Providers* and *Adaptation Coordinators*.

Users: In this analysis, we are viewing the *User* as just another agent that participates in the adaptation process. For this task, the user could need assistance with representation to the other groups of agents in the OSA. Within the EdInfo project (see Section 2.1), such a user representative was implemented in the form of a PSA.

To enable service interaction, it is useful if the surface-level system-user interaction can also be integrated. A plain window system suffices to present the interfaces to different services, but it is desirable that services also are enabled to integrate their various presentations into a single one. Espinoza [16] describes an interaction system that has been developed for this purpose. It provides a front end to agent-based frameworks for open service architectures, in which services for example can share user interface components or relate interface components to each other through a common layout schema. This interaction system was used as the front-end for the Kimsac system.

Example

The prime example of an interaction system is of course a WWW browser, through which services of all kinds share a common interface. The browser knows how to render presentations specified in HTML directly, but can also be equipped with a set of components (shared between services) for rendering of presentations in other specifications. □

Service Providers: A *Service Provider* is responsible for the content and functionality that its service constitutes. It possesses most of the knowledge about the information and behaviour that can be adapted to the user, and so they are bound to play a central role in the task of performing adaptations.

Adaptation Coordinator: The *Adaptation Coordinator* task is to coordinate actions and knowledge between the User on the one hand, and Service Providers on the other. It works as a container for information in an OSA that otherwise would be duplicated. General information about the user is an example of an entity that an Adaptation Coordinator can maintain. In the discussion that follows, we will find other kinds of information that fall into this category. The Adaptation Coordinator can also perform adaptations, completely or in part, if they either are common between a set of services, or for other reasons delegated to the Adaptation Coordinator by a service.

Example

In the example mentioned earlier in this section, it is easy to identify the different actors. The User is of course the user of the WWW browser, using the browser as an interaction system that renders presentations of services and interprets user input for forwarding to Service Providers. The Adaptation Coordinator is a component in the browser, and Service Providers are WWW recourses (e.g. web pages) on the Internet. □

4.2. Models

To allow agents to take on these roles in collaboration, they must share common views on some classes of information. This common understanding may be built-in, or such views must be agreed upon prior to executing adaptations.

What follows is a division of the models that together describe an adaptation: *Domain Models*, *User Model* and *Adaptation Model*. While the division is influenced by Benyon [15], we have switched focus from design aspects of adaptation to the matters of how the adaptation process can be distributed.

Domain Models: In order to perform an adaptation, a great deal needs to be known about the domain in which changes are to be made. *Domain Models* describe adaptable aspects of services, including physical design, logical functions and tasks related to services.

One important aspect of adapting services is to recognise that in order to perform an adaptation at a certain level, this level must be modelled in a Domain Model. For example, if a menu is to be adapted to usage frequency, a model describing the menu hierarchy must exist, or if the set of tasks that may be performed is to be adapted to the user's cognitive skills, a task model is needed.

Example

In the example from the introduction of Section 4, the domain model includes a common understanding of the object link. Also described are actions that can be performed on it (*follow*), as well as attributes of it (*current_state*, *time_since_last_follow*, *not_followed_colour*, and *followed_colour*). □

Models of service domains are service specific and to the greater part owned by Service Providers, they are not likely to be duplicated in an OSA. However, it is often useful to store at least parts of Domain Models with the Adaptation Coordinator. This is true if multiple services share a domain, whether in part or completely, and wish to delegate the task of performing adaptations to the Adaptation Coordinator. This would let the user control an adaptation that applies to multiple services from one entry point only.

Another reason for storing parts of Domain Models with the Adaptation Coordinator is that the user-modelling capabilities of the Adaptation Coordinator may require domain-specific knowledge to allow for making inferences.

User Model: The *User Model* describes attributes that the system can adapt to; the very source of input for the adaptive system, namely the user of the system. Models that describe aspects of the user include profiles, cognitive models, and student models.

User models can be implemented with generic components. An example of such a tool is BGP-MS [17], a shell for modelling user's (presumed) knowledge, beliefs and goals. General parts of the user model naturally reside within the Adaptation Coordinator. However, User Models sometimes are not easily expressed in a generic form. For example, Malinowski [18] describes a system that models the user's use of an interface as a base for adaptation of shading and colouring of user interface details. This adaptive system shades items that have not been used for a long time, and colours red fields with values not commonly used. Malinowski's User Model constitutes a very shallow description of the user, a model that is quite implementation dependent.

Example

The model for describing what to adapt to includes all attributes that are needed to decide the colour of the link (i.e. the adaptation). These attributes include one set of values each for representing default values, the user's values and the Service Provider's values. Also defined is an attribute that states whether the user accepts Service Provider overrides or not. Note that this model need not be completely distributed between the actors. Only the Adaptation Coordinator needs to access all parts of it. □

Even in the case when the User Model is service or domain specific, there may be reasons to store it with the Adaptation Coordinator. It is always desirable to let users have control over their own models. A user should for example be able to specify what information about itself that a service should be allowed to share with other services. In some cases, the user might even want to hide the User Model from the very service that the adaptation stems from. If the Adaptation Coordinator is implemented as a component that the user can trust, the user can store information with it instead of with individual services. Given that the Service Providers agree to delegate the necessary parts of their domain models, the Adaptation Coordinator can perform the adaptation in place of the service.

Based on this discussion, it seems like a good idea to place large parts of the User Model with the Adaptation Coordinator. This makes for a situation where adaptations that are directed towards the user can be based on a uniform model, no matter which Service Provider is responsible.

Adaptation Model: The *Adaptation Model* describes mappings from attributes of the User Model, to attributes of the set of Domain Models. These mappings constitute descriptions of the very adaptations that the adaptive system performs.

The Adaptation Model also includes mechanisms for detecting when a certain adaptation should be triggered (i.e. a mapping from the User Model to the Adaptation Model itself). Adaptation Models of self-regulating, self-mediating and self-modifying adaptive systems [19] also include functionality for evaluating the effect of an adaptation, together with functionality for adapting the adaptations themselves (i.e. mapping both from and to the Adaptation Model).

Example

The Adaptation Model includes an adaptation function as follows:

```
if(isDefined(sp_not_followed_colour) &
    isDefined(sp_followed_colour) &
        allow_override==true)
      if(current_state==followed)
```

```
setLinkColour(sp_followed_colour)
else
setLinkColour(sp_not_followed_colour)
else if
(isDefined(user_not_followed_colour) &
isDefined(user_followed_colour))
if(current_state==followed)
setLinkColour(user_followed_colour)
else
setLinkColour(user_not_followed_colour)
else
if(current_state==followed)
setLinkColour(default_followed_colour)
else
setLinkColour(default_not_followed_colour)
```

The ontology must also properly describe the attribute current_state, which holds information about whether the link has been followed or not. □

The Adaptation Model can exist in both domain specific and generic forms. Service providers generally hold knowledge about adaptations that are specific to the service, e.g. "if the user is unidentified, disable secure transmission". The Adaptation Coordinator may complement such information with generic knowledge about adaptations, e.g. "activate help if the user makes three consecutive errors when interacting with a new service".

4.3. Adaptation phases

The adaptation process can be split into a set of tasks. Kühme et al. [14] suggest the following four phases of adaptation:

- *Initiative*. One of the actors suggests a need for an adaptation.
- *Proposal*. Adaptation alternatives are proposed.
- *Decision*. One of the alternatives is chosen.
- *Execution*. The adaptation is executed.

Kühme et al. [14] mapped these tasks to the set of actors that were most relevant for their analysis, *namely* the *system* and the *user*, thus developing a way to describe configurations of adaptations. In order to describe adaptation configurations in OSAs, we replace a system with Service Providers and Adaptation Coordinator, as defined in Section 4.

This model of adaptation configurations in OSAs partially determines how the models described in Section 4.2 are distributed between actors. For example, the configuration that is pictured in Fig. 2 indicates that the Service Provider must possess large parts of the Domain Model. Otherwise, it would be difficult to actually execute the adaptation. The Adaptation Model that describes what is triggering the adaptation and the direct mapping from the User Model to the Domain Model must be held by the Adaptation Coordinator. For this purpose, the Adaptation Coordinator, as well as the User, need to share an ontology of terms with the Service Provider for describing the adaptation.

***Interaction schemes*:** The actors that participate in the adaptation process must agree upon which interactions are necessary in order to complete the adaptation. We will refer to such an agreement as an *Interaction Scheme*. This construct constitutes a detailed description of the interaction that takes place between the User, Service Providers and the Adaptation Coordinator while performing an adaptation.

An example of an Interaction Scheme, taken from the KIMSAC project [4], is illustrated in Fig. 3. In this example, the User interacts via an Interaction System (see Section 4.1, Users) and the task is to adapt the selection of terms in a

	Adaptation Coordinator	Service Provider	User	
Initiative	(shaded)			The AC initiates the adaptation
Proposal	(shaded)			The AC proposes a set of alternatives
Decision			(shaded)	The user selects an action from the set of alternatives
Execution		(shaded)		The SP in question executes the choice of the user

Fig. 2. An example of a task/actor configuration of an adaptation in an OSA. Note that this configuration allows only the user to decide which adaptation to execute.

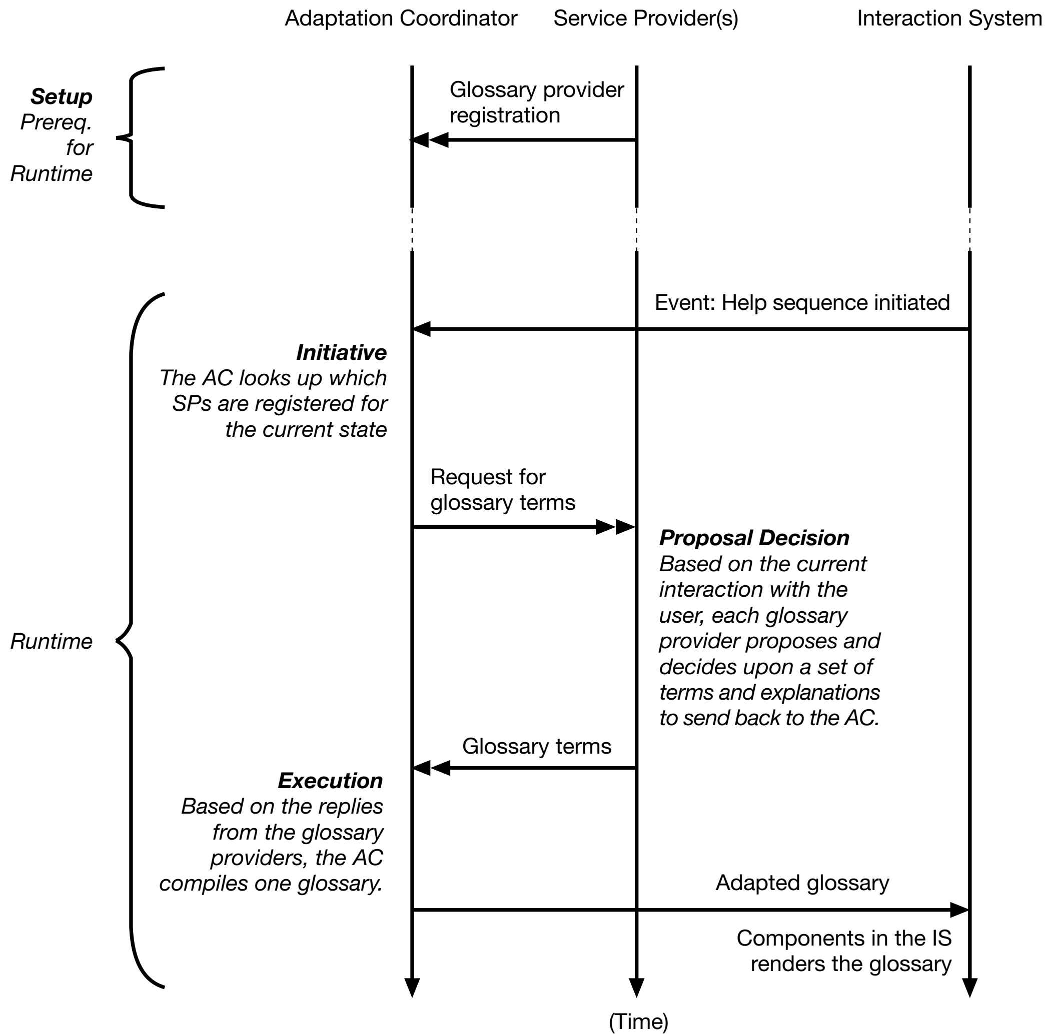


Fig. 3. An example of an Interaction Scheme from the Kimsac system [4]. Horizontal arrows represent messages being sent between actors. Arrows with a single head represent a single message, and arrows with two heads represent multiple messages.

glossary to the content of the workspace of the Interaction System. The adaptation is triggered when the User requests extended help, an action that will reveal an option for the user to request the context-sensitive glossary. Multiple Service Providers can sign up to contribute with terms for the glossary, and the Adaptation Coordinator coordinates the compilation of the selection of terms.

5. Adaptation Coordination as a Service Contract

In the previous section, we analysed in detail what information services need to share in order to allow adaptation coordination. Here, we will discuss how this information can be negotiated as part of a service contract.

As should be clear from the previous discussion, adaptation coordination requires two levels of service contract: one that governs the actual adaptation coordination, and one meta level contract that is used when a user subscribes to a new service. We expect that the latter would be hard-wired in the agent architecture: agents that take on the roles of Service Providers and Adaptation Coordinators must share a common communication language, in which they can negotiate how to perform adaptation coordination. The complexity of this language depends on when the adaptation coordination models are decided upon, whether at design time, at startup, or during execution.

5.1. Specifying the domains during design

The simplest solution is to define all models, including the interaction schemes, at design time. If this approach is used, a new service can introduce itself to an adaptation coordinator simply by stating which domain models it is able to handle. This introduction is in itself a very simple negotiation, similar to the KQML approach, where agents publish themselves to a facilitator by stating their competencies [7]. We could envision that the Adaptation Coordinator is allowed to reply "no" to this information, implying that the service has mentioned a domain model that the Adaptation Coordinator is unaware of.

An obvious drawback with this option is that all methods for adaptation need to be known when designing the system. Adding new adaptations to the system would require that the adaptation coordinator and many of the services are redesigned and reimplemented. In practice, this is a very disturbing limitation, as it severely restricts the OSA in terms of what kinds of services it can cover. The very purpose of an OSA is to allow new services and new users to come online. At least sometimes, this should lead to the introduction of new service domains, new user aspects, or new methods for adaptation.

Example

The coloured links adaptation is an example where ontologies and the interaction scheme are fixed during design. This makes the implementation very straightforward. Service Providers do not even have to participate in the adaptation process at all. If they do want to affect the process though, all they have to do is to add a few statements to the HTML code that specifies how the service should be rendered. However, if a Service Provider wishes to extend the adaptation, for example by introducing a third colour for links that the user followed more than a week ago, it would not be possible. There is simply no means for the Service Provider to express this to the Adaptation Coordinator and the User. □

Service Contracts defined at startup: A more flexible solution is obtained if agents are implemented in a generic manner, so that they can be specialised by *loading* appropriate parts of the domain, user and adaptation models. The models are then stored in a separate database that is accessed by all agents upon startup of the system.

This solution allows the designers to make changes to the models that the system supports, without having to redesign and reimplement an arbitrarily large subset of the agents. There are two main drawbacks of the method, though. First, the option is obviously only open to systems that can be shut down and restarted each time a novel type of service is added. Second, all agents must share a common language for the descriptions of these models, so that they are able to load appropriate parts of the domains at startup. As discussed in Section 2, a useful requirement on an OSA is that developers can develop services independently of each other, adhering to as small a set of standards as possible. It may be difficult to incorporate this method for domain specialisation, and still adhere to this principle.

The KIMSAC system [4] was made open for maintenance using this technique.

5.2. Service contract negotiation at runtime

The most flexible solution is found if agents are allowed to negotiate service contracts at runtime. In this case, agents must share a communication language in which they can negotiate parts of the runtime service contract prior to executing it. But even this alternative has many variants: we need not allow all five levels of the service contract (the why, what, how, when and language levels discussed in Section 3.1) to be negotiated. In addition, at each of the levels of the service contract some information can be hard-wired into the OSA structure, depending on the fact that service and adaptation coordinator agents take on predefined roles in the architecture. Below, we show why the domain, user and adaptation models cover most aspects of what remains to be negotiated. In practice, the negotiation will be even more limited in most systems, depending on restrictions in what Service and Adaptation Coordinators actually can do.

Why: We have so far silently assumed that agents need not negotiate at the "why" level for implementing adaptation coordination. The adaptation agent exists solely for the purpose of coordinating adaptations. Services that wish to participate in coordinated adaptation will simply initiate negotiation with the adaptation agent, which is always willing to negotiate a service contract.

This assumption is rather too strong, though. The privacy issue makes it unlikely that the user will always be willing to volunteer sufficient personal information to an arbitrary service to allow adaptation coordination. Instead, the user may very well choose, for the sole purpose of remaining anonymous, to use the service in a less personal way than is optimal. In reality, there should be a negotiation at the "why" level, in which the user is presented

with some information about what the service can do, and can decide whether he or she really wants to use this service in a personalised manner.

What: The "what" issue is encoded in the agent roles: service agents provide the adapted services, and adaptation agents coordinate these adaptations. The "what" issue is closely related to the competence profiles for agents. Negation at the "what" level may occur if there are adaptation agents with different competence profiles, for example if they have different capabilities to perform user modelling or store service-specific information about the user.

How: The adaptation model expresses the "how" level issues for adaptation coordination. As discussed in Section 2, the adaptation model is seldom shared, but resides partly with service and partly with Adaptation Coordinator agents. In practice, this means that much of the adaptation model information will be hard-wired into agents and never explicitly negotiated.

In addition to the adaptation model, parts of the domain and user model structures may very well need to be negotiated in order to tailor the competencies of agents prior to execution. An example is when a service agent provides information about the domain structure to the adaptation coordinator, in order to allow it to make domain-dependent inferences about the user during execution.

When: The interaction schemes discussed in Section 3 describe "when" agents collaborate in adaptation coordination. Ideally, the actors should negotiate these by the actors prior to execution. In practice, we can expect that an OSA will only supply one or a few types of adaptation coordination agents, each capable of only a small set of interaction schemes. One reason is that many of the possible interaction schemes are meaningless in practice, since they assume impossible distributions of competencies between the actors involved (Bylund [20] discusses this issue further). If the agents only support a limited set of interaction schemes, the negotiation of interaction scheme is reduced to agreeing on the names of the ones that will be used. At runtime, the agent that initiates adaptation can select the interaction scheme, as is done in KaOS [21].

Language: As should be clear from the EdInfo example discussed in Section 2, it is often meaningless or impractical to require that agents interact in a generic language during execution of a service contract. Much of the language of interaction can be defined by a combination of explicit transfer of domain and User Model knowledge (negotiation at the "how" level), and explicit definition of the interaction schemes. This could, however, be very costly, and we expect that designers of adaptive systems must compromise here between openness and the cost of implementation. A useful approach is that taken by current standard agent communication languages such as KQML [22]. In KQML, the syntax and certain high-level aspects of the semantics are fixed, whereas the detailed semantics are up to designers as well as to agent negotiation to tailor to the specific needs of the application.

6. Conclusions

Agent-oriented programming provides a useful means of integration and coordination of services inOSAs. Service contracts form a useful concept in describing how agents negotiate and execute collaboration.

One particularly challenging example of a service contract is the task of coordinating multiple services that adapt to a singular user. The example is challenging, since adaptation coordination is a very complex process, both in terms of what information needs to be shared between agents, and how agents need to communicate. To deal with the task, we suggest that a separate agent be introduced that coordinates the adaptations between services. Based on this architecture, we analysed the knowledge needed to perform adaptation coordination, and how this knowledge needs to be distributed between the service and adaptation coordination agents. Finally, we discussed how this knowledge forms a service contract, taken together with such knowledge that is hard-wired into the roles of the service and adaptation coordinatior agents in the architecture.

In addition to the task of coordinating adaptation, service contract negotiation can also provide means for users to find services and pay for services. We are currently involved in the development of different open and adaptive information service systems. In this work, we have come to focus on a common agent-based platform for such services, which implements a limited range of service contract negotiation for information services. The work is carried out in collaboration with work at SICS on open platforms for electronic commerce [23].

References

1. Bradshaw JM. Software agents. AAAI Press,Menlo Park, CA and MIT Press, Cambridge, MA, 1997
2. Shoham Y. An overview of agent-oriented programming. In:Bradshaw JM (ed) Software agents. AAAI Press, Menlo Park, CAand MIT Press, Cambridge, MA, 1997; 271–290
3. Shoham Y. Agent-oriented programming. Artif Intelligence, 1993; 60:51–92
4. Charlton P, Chen Y, Espinoza F et al. An open agent architecture supporting multimedia services on public information kiosks. In: Proceedings from Practical Applications of Intelligent Agents and Multi-Agent Systems, PAAM'97, London, 1997
5. Höök K, Rudström Å, Waern A. Edited adaptive hypermedia: combining human and machine intelligence to achieve filtered information. In: Proceedings from Flexible Hypertext Workshop held in conjunction with the 8th ACM International Hypertext Conference, Hypertext'97, 1997
6. Waern A, Tierney M, Rudström Å , Laaksolahti J. ConCall: edited and adaptive information filtering. Proceedings from Intelligent User Interfaces (IUI99), Los Angeles, 1999 (forthcoming)
7. Finin T, Labrou Y, Mayfield J. KQML as an agent communication language. In: Bradshaw J (ed) Software agents. AAAI Press,Menlo Park, CAand MIT Press, Cambridge, MA, 1997
8. Geneserth MR. An agent-based framework for interoperability. In: Bradshaw J (ed) Software agents. AAAI Press,Menlo Park, CAand MIT Press, Cambridge, MA, 1997; 317–345
9. Smith RG. The Contract Net Protocol: high-level communication and control in a distributed problem solver. IEEE Trans Comput 1980; C-29
10. Rasmusson L, Rasmusson A, Janson S. Using agents to secure the Internet marketplace: reactive security and social control. In: Proceedings from Practical Applications of Intelligent Agents and Multi-Agent Systems, PAAM'97, London, 1997
11. Waern A. Service contract negotiation: agent-based support for open service environments. In: Proceedings from 1998 Workshop on Distributed Artificial Intelligence, at the 4th Australian Conference on Artificial Intelligence, Brisbane, Australia, 1998
12. Verharen E, Dignum F, Bos S. Implementation of a cooperative agent architecture based on the language-action perspective. In: Proceedings from The Fourth International Workshop on Agent Theories, Architectures, and Languages, ATAL'97, Providence, Rhode Island, USA, 1997
13. Tsvetovatyy M, MobasherB, Gini M, Wieckowski Z. An agent-based virtual market for electronic commerce. Int J Appl AI (in press)
14. Kühme T, Dietrich H, Malinowski U, Schneider-Hufschmidt M. approaches to adaptivity in user interface technology: survey and taxonomy. In: Engineering for human-computer interaction. Elsevier, Amsterdam, 1992
15. Benyon D. Adaptive systems: a solution to usability problems. User Model User-Adapted Interaction 1993; 65–87
16. Espinoza F. sicsDAIS: Managing user interaction with multiple agents. Ph.Lic. thesis, Department of Computer and System Sciences, The Royal Institute of Technology and Stockholm University, Stockholm, 1998
17. Kobsa A, Pohl W. The user modeling shell system BGP-MS. User Model User-Adapted Interaction 1995; 4:59–106
18. Malinowski U. Adjusting the presentation of forms to users' behavior. In: Proceedings from 1993 International Workshop on Intelligent User Interfaces, Orlando, Florida, 1993
19. Browne DP, Totterdell PA, Normann MA. Adaptive user interfaces. Academic Press, London, 1990
20. Bylund M. Coordinating adaptations in open service architectures. M.Sc. thesis, Computing Science Department, Uppsala University, Uppsala (forthcoming)
21. Bradshaw JM, Dutfield S, Benoit P, WoolleyJD. KAoS: toward an industrial strength open agent architecture. In: Bradshaw JM (ed) Software agents. AAAI Press, Menlo Park, CAand MIT Press, Cambridge, MA, 1997, 375–418
22. Labrou Y. Semantics for an agent communication language. Ph.D. thesis, Computer Science and Electrical Engineering Department, University of Maryland Graduate School, Baltimore, Maryland, 1997
23. Eriksson J, Finne N, Janson S. Information and interaction in MarketSpace – towards an open agent-based market infrastructure. In: Proceedings from Second USENIX workshop on Electronic Commerce, 1996

Correspondence to: A. Waern, Swedish Institute of Computer Science, Box 1263, SE-164 29 Kista, Sweden. Email: annika@sics.se

Paper B

Personal Service Environments – Openness and User Control in User-Service Interaction

Markus Bylund and Annika Waern

Swedish Institute of Computer Science, Box 1263, SE-164 29 Kista, Sweden
{bylund, annika}@sics.se

Abstract. We describe an approach for mobile and personalized use of electronic services that meet very high requirements on openness, user control, and mobility. The design is centered on the concept of personal service environments. These offer users mobile and network independent access to services from many different types of devices. The concept also allows services to interact locally. This can be used in several ways: services can for example share information but also make use of each other's services. In particular, we want to use this to support service personalization. The design has been successfully implemented and tested with a number of sample services and in several related research projects. This implementation, the *s*View system, is described.

Keywords. Electronic services, personal service environments, user control, ubiquitous computing, user interfaces, mobility, personalization, service collaboration.

May 2001
SICS Technical Report T2001/07
ISSN 1100-3154
ISRN: SICS-T--2001/07-SE

1. Introduction

Today, most people in the Western world are used to using Internet-based services. We use services – almost exclusively placed in the World Wide Web (WWW) framework – to search for information, to buy books, and to book airline tickets. We use them to communicate with our friends, and to make new friends. A home computer is no longer made useful only through the stack of CDs on our bookshelf – it is a point of access to a vast and perpetually changing world of entertainment, shopping, business and general information resources. And as if this was not enough, we are rapidly acquiring other devices for access to the Internet world of services: mobile phones, communicators, home gateways and Internet-equipped game consoles.

This rapid development has created a mismatch between the development of Internet content and the technology for content provision. Content is developing into a set of independent (but potentially combinable), device-independent, and personalizable services. But the predominant technology still provides a strongly server-based functionality, where services run entirely at the server side, accessed through media-specific, content-insensitive, interaction devices (HTML/WML browsers, media players, etc.) with little support for user personalization.

We see two main problems with this development. Firstly, current technology lacks true support for nomadism and continuous interaction channels through multiple devices [1, 2]. Secondly, technology put services almost entirely in the control of service providers. This gives rise to privacy problems, such as the fact that service providers have complete control over the information users provide to personal services. Even more importantly, it makes it difficult to combine services to support specific users' need. For example, there are still very few Web sites that offer automatic comparisons between sales offers from other Web sites, despite the clear benefit of such services.

These issues have not been left unaddressed in research. In particular, work on mobile agent technology and multi-agent systems (see Section 4.2) has aimed to address the issues of service nomadism and interaction. This paper takes a similar but slightly different perspective: we explore how Internet-based services, rather than generic software agents, can be developed to be nomadic, personalizable, and provide possibilities for service interaction.

The solution proposed is that of a *personal service environment*. A personal service environment is an individually collected and tailored set of services, available to the user at all times, and at least partially independent of Internet access. The services are retrieved from service providers around the Internet, and the personal service environment itself is mobile, following its user around in the network.

We have designed and developed a Java-based system for electronic services that is based on the notion of personal service environments – *sView*. In this system, personal service environments are composed both of services that are mobile and follow the user, and of services that are platform or location specific. In this way, *s*View provides personal service environments that are tailored both to the user and to

the usage context. The design of *s*View is highly modular. In fact, it could replace the Web browser as such, as some of the services may well be provided to support presentation and interaction.

The paper is structured as follows. Section 2 is a requirements analysis; in this Section we analyze the requirements on openness and user control in more depth. Section 3 presents personal service environments and shows how this concept has been designed to meet the requirements from Section 2. Section 4 covers existing Web technology and some alternative approaches in terms of how well they are able to fulfill our requirements. Finally, Section 5 presents the *s*View system, and Section 6 describes a few experiences that build on the work presented in this paper.

2. Requirements Analysis

We see two requirements as central for an infrastructure for electronic services. The first is that it must be *open*. It should be possible to add and remove services and users without affecting other services or users. The second requirement is that it must be *controlled by the user*. An infrastructure for electronic services should give the user control over which services to use, what information about the user that services handle, how services collaborate, etc. Some users may not ever do so, but the possibility for user control should always exist. Furthermore, the user should be in control of the usage situation. In practice this means that services should be reachable from everywhere using many different types of devices, both the user's own devices and publicly available devices.

These requirements on openness and user control imply a number of more specific requirements, which we now will discuss in more detail.

2.1. Heterogeneity

Many electronic services already exist, both in the form of commercial and research products. A sound requirement on an open infrastructure for user-service interaction is to allow a heterogeneous mix of service components to utilize features of each other.

2.2. Extendibility

Openness also implies a demand for extendibility. As new services are added to the system it should be possible to add support for new protocols for user-service interaction, protocols for communication between service components, support for collaboration between services of different kinds, protocols and algorithms for implementing security functionality, and more.

2.3. Accessibility

User control requires that all users always can access the service infrastructure. Users should be able to access services while on the move, not just from the office or from home, but also while riding the bus, in an airplane, on the street while shopping, etc. Disabled and elderly users must be able to access the infrastructure, as well as children.

2.4. Adaptability

Accessibility is only the minimal requirement for user control, but it poses already very high demands on adaptability. The service infrastructure must be extendable and adaptable to a wide range of input devices [3]. Another requirement is network adaptability: the infrastructure, and services, must be able to adapt to variations in network connectivity and bandwidth [4, 5].

Services should also be able to adapt to their users, or rather the preferences, experiences, or usage history of their users. This is a delicate issue since adapting to qualities of individual users requires services to handle personal information, which in turn may jeopardize the privacy of users [6]. The requirement on user control means that personalization must be done in a way that ensures user privacy. Furthermore, the infrastructure should ensure that the task of managing personalization does not overburden the user.

2.5. Continuity

Finally, user control requires that services not only are accessible from multiple devices, but also maintain their state when the user switches between devices. Users should not need to restart a service just because they move over to another device. When switching between devices, a user should be able to resume his or her interaction with a service exactly where it was suspended.

3. Proposed Approach

The personal service environment concept describes a service infrastructure that is targeted to fulfill the requirements from Section 2. It is a runtime environment that is private to an individual user, and functions as a briefcase for his or her electronic services. As with the Web, we assume that services themselves use a client-server model. In contrast to the Web however, services can store both logic and data locally within the personal service environment, and there is no predefined split between what should be performed on the client and on the server sides.

The service environment infrastructure fulfills our requirements on openness. Any individual or organization with an Internet connection can own an environment in which services can be stored and executed. In the same way, anyone can publish

services for use in an arbitrary service environment. Adding a new user or service to the system does not affect already present users or services.

The most important requirement we pose on service environments is that the environment itself must be mobile. It should be able to follow its user in the network between e.g. a workstation when the user is in the office, to a notebook computer when the user is on the road, or to a shared server for service environments when the user lacks immediate access to the network. As the environment migrates, the services it stores should follow, and the state of the services and their ways of interacting with each other should be preserved. We also require that the service environment can move between client devices without loss of interaction state.

There are several reasons for making the service environment mobile and execute services locally. Firstly, a service that executes locally is not necessarily dependent on a network connection. Secondly, a local service is likely to have access to richer user interface types than remote interfaces. Thin devices with Internet access, and possibly with a less powerful interface (e.g. WAP/WML capable phones), can be used to access the environment on a networked host. Finally, by having parts of the functionality of services executing on users clients, the total CPU processing of a service is distributed between the users, which alleviates base services.

A key feature with the personal service environment is that it provides a natural boundary for service-service interaction. Services within an environment could be allowed to publish their APIs (Application Programming Interfaces) to each other.

To meet requirements on user accessibility, personal service environments must also support numerous channels for user interaction, e.g. HTML over HTTP, WML over WAP, Graphical User Interfaces (GUI), etc. The architecture must be open to enable integration of novel interaction models over time.

Since service environments are personal and follow their users around, they provide an ideal place for storing personal information for use by services (e.g. preferences and contact information). Whenever a user wants to add or change his or her personal information, or just inspect the information, it can be done in one place for all services. Service providers get a central access point to personal information of each user, information that can be shared with other services (with the user's permission).

3.1. A Usage Scenario

Below we present a usage scenario that illustrates the use of personal service environments and a few services.

> *A man is about to make a business trip to Cairo. Using his personal service environment search tool on his desktop computer he locates a travel agency service and initiates a dialog with it.*
>
> *The travel agency uploads a travel service component to the user's service environment.*
>
> *Once in the service environment, the travel service receives the man's instructions, via a standard graphical user interface (GUI), to make a flight and hotel reservation for his planned trip.*

Then the man turns his attention to something else and leaves the office. But before doing so, he lets his service environment know that he is no longer available via his desktop computer but rather via his cellular phone.

The travel service now makes use of a number of information sources in order to accomplish its task. It searches the service environment for a preference manager and asks it about its client's complete name and address, as well as his seating and smoking preferences. It also locates a calendar within the service environment and checks when the man must be back and if the trip conflicts with any of his other appointments.

Having collected all background information, the travel service turns to its base service trying to find an appropriate flight and hotel. The service finds three alternatives that all match the man's request, preferences, and schedule.

The travel agency is now ready to get back to the client with the result of the search. However, since the man is no longer available via the desktop computer, the service contacts him via his cellular phone. The man, now on the train on his way home, selects one of the alternatives and instructs the travel agency to go ahead with the reservation.

The travel service accepts the request and starts searching the client's service environment again, this time for a service that provides payment. One of the man's services, a bank service, is willing to provide payment, but only after a confirmation by the user (this is also done through the interface of the cellular telephone).

Having everything that is needed, including payment, the travel service now executes the man's request by instructing its base service to buy the flight tickets and make the hotel reservation.

4. Related Technologies

There exist many techniques and systems that fulfill some of the requirements of Section 2. The concept of personal service environments has in many ways been inspired by existing work on WWW enhancements, as well as of experiences with mobile agent technologies. There are also examples of more recent technology that fulfils some, but not all, of our requirements.

4.1. The World Wide Web

The WWW was originally intended to combine hypertext and text retrieval to get a "global information universe into existence using available technology" [7]. Considering these design goals, it is truly remarkable that the Web has been able to take on the role as an infrastructure for general user-service interaction as it plays today, with demands on highly interactive interfaces, mobility, and personalization. The ambitions of the inventors of the WWW to make the Web extendible, platform independent, and transparent has clearly played a key-role in this development. However, with the requirements in Section 2 in mind, the WWW is facing some challenges.

Services mediated through the WWW require a Web browser for user interaction. Web browsers in turn, most often require a quite powerful computer as host, as well as a keyboard and a mouse, in order to function. This limits the types of devices that users can reach services from. Telephones (both traditional and cellular), palmtops, and other special purpose devices can be used in some cases, but only by extending the WWW with separate interaction systems such as WAP/WML technology.

On the WWW, service logic and data are hosted by the set of backend services used by an individual user. This results in a dependency on the network connection between users and services; if the connection fails, not even the simplest functionality of a service can be utilized. In many cases (e.g. bank services), the scheme used to ensure privacy makes it impossible to even view information that was viewed only moments ago, just before the connection broke.

The support for saving the state of the user-service interaction is also limited. If a user is in the middle of a session with a service the user cannot suspend the interaction in order to resume it from someplace else. This is because WWW clients, through HTTP, are stateless [7]. The state of the user's services is instead distributed across all of the user's service providers, which makes it difficult to find a general solution to the problem.

There is little support built into Web browsers for personalizing services. Essentially, the only way to handle it is by having the service provider identify the user in order to tailor the interaction at the back-end of the service. The problem is worsened by the fact that personal information of individual users is distributed across all of their service providers. As the number of services in use increases, the user soon loses control over the personalization process. Also, all information that is needed for personalization, no matter how sensitive to the user, needs to be passed to the service via a network. This opens for privacy issues.

While it is a strength of the WWW that no information about other services is needed in order to add a new one, the lack of a general way to obtain information about other services makes service collaboration difficult. This is a two-faced problem. Firstly, services have difficulties finding peers to collaborate with since there is no uniform way for services to publish their capabilities to other services. Secondly, how do they actually collaborate once a peer is found? The APIs of Web-based services are typically made for humans using protocols that are very awkward for machine-based services to utilize. While it is relatively easy for users to find and use such services, these problems make it difficult for end-users to combine the services' functionality. The Simple Object Access Protocol is an example of a recent initiative to relieve the latter problem with the WWW [8].

General Web Extensions. Web organizers (e.g. www.eorganizer.com) and virtual desktops (e.g. www.magicaldesk.com) provide their users with integrated suites of Web-based e-mail handling, calendar, on-line storage of data, and sometimes news and games as well. In some cases, the services go as far as to simulate a desktop environment of an ordinary personal computer, complete with folders, desktop icons and even drag-and-drop functionality. In a way, this approach is similar to personal service environments. Even closer comes NetChaser [9], which is a system that supports personal mobility of Internet services such as the WWW, FTP, and e-mail. The system offers its user a personal view of his or her services via WWW browsers.

The system keeps track of the state of its user's services, which makes it possible for a user to start a session on one WWW client, suspend the session, and resume it again from a different client.

The major difference between these approaches and the personal service environment concept is that the former do not meet our requirements on openness. Web organizers are not in any way open for everyone to add new services. Furthermore, since these services rely on Web technology, they are not able to adapt to changing bandwidth availability or intermittent Internet access.

4.2. Mobile Agent Environments

Personal service environments bears many similarities to general Mobile Agent Environments (MAE) [10] and the concept was partly inspired by experiences from projects in which MAE were applied [11, 12]. They both provide environments that support dynamic loading of lightweight software components, as well as migration between such environments.

Many application examples apply mobile agent technology to meet requirements that are similar to those analyzed in Section 2. For example, Minar et al. [13] use mobile agents as a primary abstraction for creating dynamic and distributed systems with a focus on embedded computers. Hive agents are self-describing, mobile and capable of dynamic collaboration. Users are given a high degree of control in that they can create and manipulate (kill and move) agents using a GUI. The user can also create new applications by connecting agents with different capabilities, just by drawing lines between them. The system described by Minar et al. can be seen as a potential high-end interface for personal service environments.

Pullela et al. [14] describe a middleware that dynamically distributes computations in a mobile environment. The distribution is based on what resources are currently available at the mobile client. In this, Pullela et al. fulfill our requirement on access from multiple platforms, including very thin clients. They make use of the Ronin Agent Framework [15] that mixes agent-oriented and service-oriented paradigms for creating dynamic distributed systems. The Ronin Agent Framework shares the requirement on heterogeneity with the PSE concept, and meets it by including a meta agent communication language and a network independent agent communication message mechanism.

The Nomad system [16] is an advanced example of a service built on mobile agent technology. It allows mobile agents to travel to an auction service provider and participate in auctions on their user's behalf. Agents can place bids, learn and collect information, and set up new auctions. The Nomad system is an example of a type of service that goes beyond the requirements posed in Section 2, since the agents are initiated by users and travel to servers.

While MAEs have been a great source of inspiration when designing the personal service environment, the two are not altogether the same. A personal service environment is, in contrast to MAEs in general, specialized towards a particular task: to enable user interaction with networked services. This means that much of the functionality that is traditionally associated with MAEs can be simplified or removed [12]. For example, client side service components need not be able to migrate freely

between any two service environments; it is enough if a service component can be created at its base service and then moved to its owner's environment. Assumptions can also be made about the scope of a service component. It only needs to know about its base service and the other service components within the same environment.

Service environments on the other hand pose higher, or at least more complex, requirements on mobility and persistence than pure MAEs. A personal service environment must support persistence of itself as well as the service components that it houses. It also controls how services are allowed to move. Once services have been initialized in a service environment, they do not move individually over the network. Their mobility is rather controlled by the service environment they belong to.

In summary, although personal service environments could be implemented using MAE technology, they are not subsumed by it. As an implementation option for personal service environments, mobile agents introduce unnecessary overhead, as many central functionalities would be used only to a limited extent.

4.3. The OSGi Service Gateway

OSGi (the Open Services Gateway Initiative) provides a specification of an open framework for a service gateway [17]. The gateway can be loaded with multiple software services and it can execute on a number of platforms. The goal with OSGi service gateway is to create a common programming model for consumer services in which implementations are separated from their functional descriptions, and to create a simple and self-contained format for distribution of services. The former goal allows consumers to make use of implementations of a service from several manufacturers interchangeably. The latter goal makes it possible to partition applications into smaller pieces, possibly implemented and provided by different manufacturers.

The OSGi service gateway shares many of the design goals with personal service environments, and its realization share many properties with the *s*View platform described below. For example, OSGi also provides an open environment in which service components from different manufacturers can be loaded and collaborate to form an application. However, OSGi does not provide *personal* environments. The OSGi service gateway is intended as a service gateway for small groups of users (e.g. a family). The service gateway environment of OSGi is also stationary, which is the natural choice for gateway software. A personal service environment needs to be mobile, since this enables services to follow the user in the network and still execute close to the user and independently of a network connection.

4.4. MExE

The Mobile Station Application Execution Environment (MExE) initiative is a budding standard for platform independent development of services, targeting mobile devices [18, 19]. The initiative includes a classification of mobile devices, which describe minimum levels of capabilities for certain categories of devices. The initiative also adopts W3C's CC/PP protocol [20] for runtime capability and content negotiation of mobile station capabilities and user preferences. This provides a

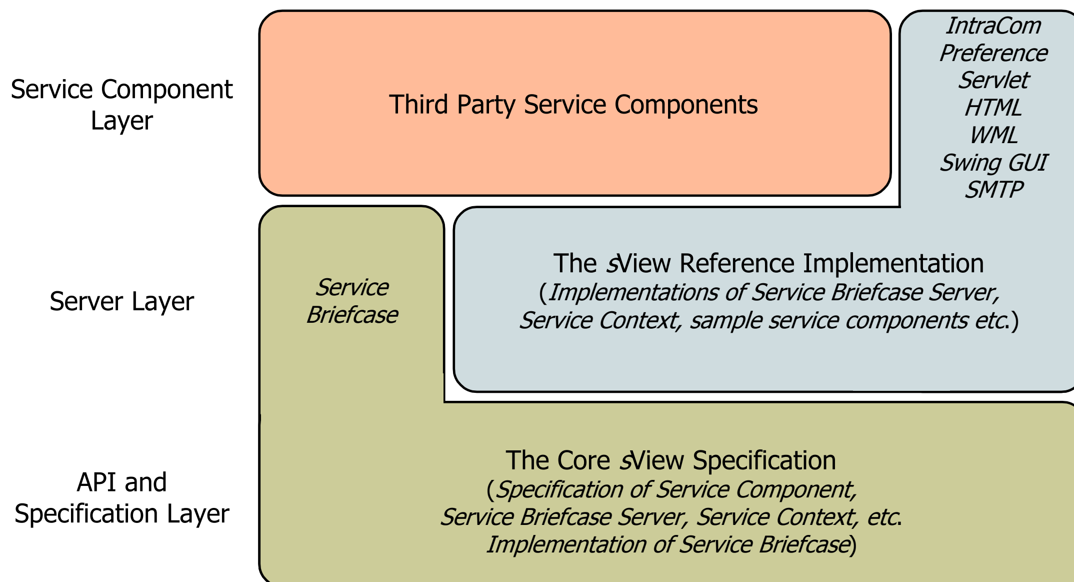


Fig. 1. A schematic overview of the *s*View system and its different parts.

foundation for performing personalization of services and adaptation of content towards mobile devices with different capabilities.

The MExE initiative shares the requirements on device independence and personalization with the personal service environment concept. The crucial distinction between the two approaches lies in how mobility is defined. In the MExE case, mobility means access to electronic services via mobile devices. In the perspective put forth in this paper however, mobility means that the service environments themselves are mobile, and free to migrate (in complete) between hosts.

5. The *s*View System

We have designed and developed *s*View to meet the requirements put forward in Section 2. In *s*View, personal service environments are composed both of services that are mobile and follow the user, and of services that are platform or location specific. In this way, *s*View provides personal service environments that are tailored both to the user and to the usage context.

At a high level, the *s*View system is separated into two parts. The *core sView specification* provides an API for developers of service components and service infrastructure that builds on sView technology. The *sView reference implementation* provides developers with a development and runtime environment for service components as well as a sample implementation of an *s*View server. An illustration of how the core specification and the reference implementation are organized in three layers can be seen in **Fig. 1**.

The architecture and design of the sView system is further described in [21]. Both the core specification and the reference implementation is available for download from http://sview.sics.se.

Mobile / Persistent	Yes	No
Yes	Follows the user and preserves its state (e.g. a calendar).	Does not follow the user but preserves its state (e.g. a printer queue).
No	Follows the user but does not preserve its state (e.g. a proxy to a Web based service).	Does not follow the user nor preserve its state (e.g. a driver for a loudspeaker).

Table 1. Examples of four types of service components.

5.1. The Core *s*View Specification

The core *s*View specification constitutes an API that defines the basics of service components and personal service environment handling. It specifies service components, a runtime environment for service components, a data structure for persistent and mobile images of service environments (service briefcases), and a server for handling service briefcases. The specification has been implemented as a set of Java packages (which contains mostly interfaces and a few classes).

The Service Component. A service provider needs to implement a service component that can be loaded into the users' service environments. The core *s*View specification includes a Java interface that service components must implement. The service component interface includes methods for initializing, starting, suspending, resuming, and stopping the service component. Service components can implement an arbitrarily large part of the functionality of the service, and range from mere proxies to Web based services, to standalone services.

Service components can be declared to be persistent and/or mobile. A persistent service component can save its state together with the service environment when the environment is saved locally on a host or migrates to another host in the network. If a service component is mobile it will follow the service environment as it migrates to a different host in the network. Note that the persistent and mobile properties are orthogonal. A persistent service component that is not mobile can save its state locally, but not migrate to a different host. A mobile service component cannot save its state, neither locally nor while moving; every time such a service component is restarted it will start from scratch (which is fine for many services). The most powerful service component however combines the two properties. Such a service component can both save its state and migrate. **Table 1** lists and exemplifies the four possible combinations of the two properties.

The Service Context. The service context is the entity that maintains and provides runtime support to service components. It manifests the personal service environment in *s*View and is responsible for creating, loading, and removing service components. Once the service components are loaded, the service context controls and sets the

states of the service components. For example, newly created service components should be taken through an initialization state, and when the service environment is about to migrate all service components should be suspended.

When a service environment resumes after having been suspended (e.g. after migrating to a new host), the service context loads service components and other data from a service briefcase. Likewise, when an environment is about to migrate to another host or save its state to disk, service components are stored in a service briefcase.

Service components within a service environment can communicate via service interfaces. A service component that wishes to offer its services to other service components should come with a class that implements an interface to the service. The service context is responsible for mediating service interfaces between service components. It is straightforward to implement a message passing service component on top of the core *s*View service-service communication scheme, and the *s*View reference implementation includes such a service.

Via the service context, service components can control the behavior of the service environment (e.g. migration and shutdown) as well as the behavior of other service components (e.g. creation, suspension, resumption). However, since these activities are sensitive matters, the user must grant privileges to each service component in order for them to perform these actions. A service component may for example be granted the privileges to create and load, but not suspend or kill, service components within the environment. Another service component may be allowed to initiate a migration of the whole service environment to another host, but not to control any aspect of individual service components.

The Service Briefcase. Service briefcases are persistent and mobile images of personal service environments of individual users. The service briefcase is what is actually saved when the user suspends the execution of a service environment to move or shut it down. This is also what is sent between hosts in the network when the user switches interaction devices.

A service briefcase consists of three parts: serialized service components, service component specifications, and preferences.

The Service Briefcase Server. A service briefcase server specifies an API for service briefcase handling. It includes basic functionality such as create new, get, put, and delete briefcase. It also includes functionality for synchronization of content in different instances of a service briefcase on different servers.

Synchronization is used when a service briefcase is to be moved to a server on which it has already been stored. In this case it is possible that parts of the briefcase (e.g. service component specifications) need not be sent with the briefcase. Synchronization is performed in two steps: the first step is to find out the difference between the two service briefcases followed by the second step to update the parts that have been changed. The two-phase commit protocol is used to ensure data integrity during synchronization.

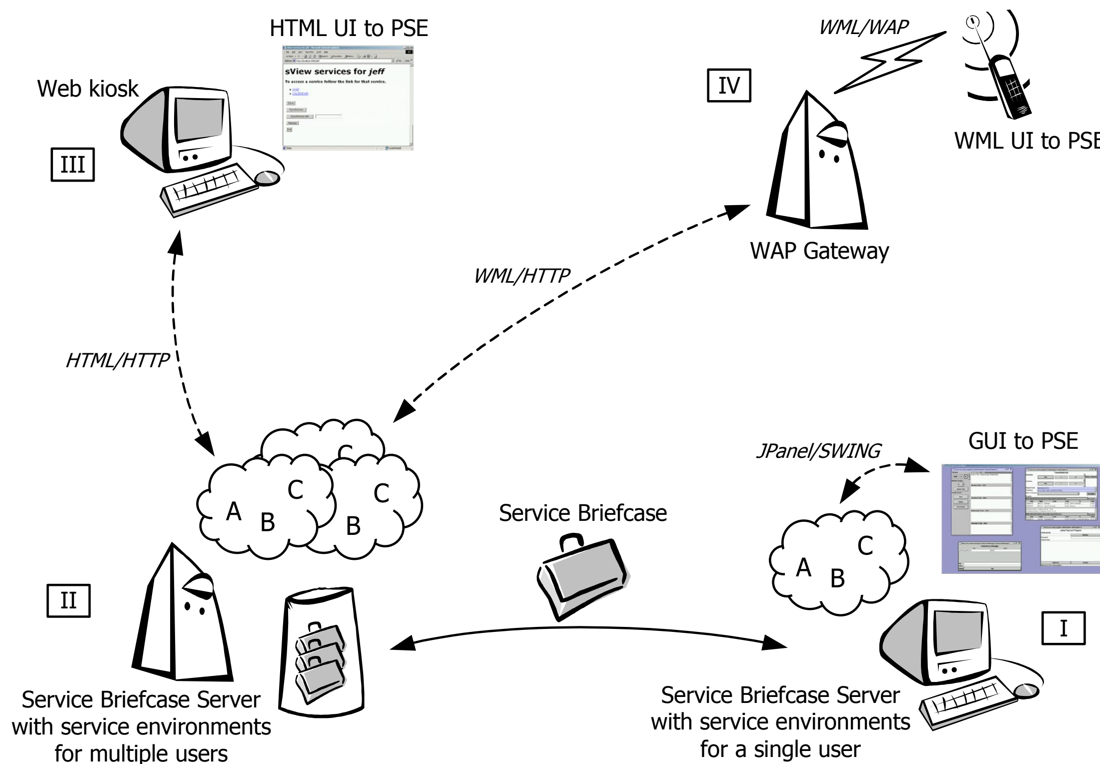


Fig. 2. The main parts of the core sView specification and their relations.

5.2. Dynamic Behavior

This Section describes how services are handled by the *s*View platform. An illustration of the different parts of the core *s*View specification and their relations is given in **Fig. 2**. On the computer marked I a briefcase server and a service environment is executing. In this case the user is sitting next to the same computer as the service environment (represented by the cloud together with service components A, B, and C) is executing on. This makes it possible to use a standard GUI for user-service interaction. The computer marked II hosts service briefcases and environments for several users, which use remote interfaces. One user is using a Web-kiosk with a Web browser for user-service interaction (III) and another user uses a WAP phone (IV). Stored service environments, in the form of service briefcases (illustrated between I and II), can migrate between any computers that run a briefcase server.

Searching for and Adding Service Components. The core *s*View specification does not include any predefined schema for how a user should search for and add service components. Instead, these tasks are left to service components. Every service component within an environment, if created with the correct privileges, is allowed to create and load new service components to the system. Based on this, numerous types of search engines can be implemented.

Saving and Moving the Personal Service Environment. A user that needs to temporarily suspend the execution of a service environment, or move to a different host, has two options. Firstly, if the service environment is executing locally and the user does not intend to move to a different host, the user can save the environment to persistent media on the host. In this case all persistent service components (see Section 5.1) are offered to save their state together with the environment. Secondly, if a network connection is available, the user can move the service environment to a server for remote storage or execution. In this case all mobile service components will migrate with the environment. Service components that in addition are declared as persistent are also offered to save their state with the service environment during the migration. The environment can also migrate directly between two clients, in which case the receiver client will act as a server for remote execution during migration.

The service context includes primitives that allow service components to initiate a save or a migration of a service environment. A service component can be implemented that e.g. automatically saves the environment whenever the user has been idle for more than a few minutes, or actively moves the environment if the user is sighted on a different host.

5.3. The *s*View Reference Implementation

The *s*View reference implementation implements most of the core *s*View specification. It includes a server that implements the service context API (to multiple simultaneous users) and the service briefcase server interface. The server can execute on any computer that hosts a Java 1.3 VM.

The core *s*View specification does not include UI handling. This is instead left as a task for service components to handle. The reference implementation includes service components that handle user interfaces of three types: GUIs specified in Java Swing as well as HTML and WML based user interfaces.

The reference implementation also includes a set of service components for handling other miscellaneous functionality. The *IntraCom (Intra Communication) manager* let service components register a mailbox to which other service components can post messages. The *Preference manager* offers rudimentary handling of preference entries (key and value pairs) for the user. Service components can store and fetch entries, as well as subscribe to changes in the preference database. The user can inspect the database, and control which services should be allowed access to which entries.

5.4. Lessons Learned

The *s*View design and implementation realizes personal service environments, and fulfils the requirements on an open infrastructure for personal mobile services discussed in Section 2. In contrast to related technologies (see Section 4), the use of personal service environments addresses a number of challenges at the same time; such as network independence, control over personal information and the use of different types of devices and user interfaces. The mere work with designing,

6.1. Service Designer

A generic problem with service communication is that it is difficult for services to collaborate unless they are explicitly designed to do so, or at least share ontological information.

In the Service Designer project [24], Fredrik Espinoza and et al. use the *s*View system and the SOAP standard [8] in order to explore how end-users themselves can create GUIs to services with only programmatic descriptions of the service available. They have developed a Service Designer service component, which allows users to download descriptions of net-based services, and provides simple means to create a GUI for the services. Once the GUI is finished, a new *s*View service component is compiled based on the net-based service and the GUI.

The service designer also allows the user to combine functionality from several net-based services in one GUI, thus creating explicit collaboration between services that have not been designed to communicate with each other.

Two of the design considerations for *s*View proved essential to this project. Firstly, the open design of *s*View enables the service generator to both support UI design, and the actual generation and installation of the created service. Secondly, the fact that *s*View provides a common locale for service logics makes it possible to create a service that actually inspects service code (in the form of SOAP objects), presents the result of the inspection to the user, and allows users to combine service functionality and create UI's.

6.2. Universal Device Access

The *s*View system promotes use of services from many different types of devices and interfaces. However, the basic *s*View implementation requires that service components are aware of, and designed for, each of the device and interface types that should be used for interaction. If a new device or interface type shows up, existing service components need to be modified.

In the Universal Device Access project [25] we have designed a method that simplifies the process of developing services that can be accessed from different user interface types. The method allows both service- and user-driven interaction, unlike UIML-based [26] applications or Web-style-interaction that are entirely user-driven.

In analogy with HTML [7] and similar script languages, we separate service logic and user interface with a general language for specifying interaction. Different devices can then implement this script language in a way that is specific for the device and interface type.

In this project, the open nature of *s*View has proven useful, in particular to enable services to fully or partly override this functionality. If the service component wishes to tailor the user interface on a particular device it can send a customization form to the interaction interpreter. The interpreter will then replace its standard interpretation of the interaction language by using this form to render output and interpret input.

6.3. GeoNotes – Virtual Notes in the Real World

GeoNotes [27-29] is an *s*View service component that lets its user annotate physical locations with virtual notes. Such a note is intended to say something about the location, just as a Post-it note can express something about the object it is attached to. Other users of the same service get the notes as they pass by an annotated location.

The GeoNotes service relies heavily on mobile and context-sensitive usage. Developing GeoNotes as an *s*View service component has therefore been an important test of how suitable *s*View is for mobile and context dependent services. *s*View provided good support for both of these requirements [28]. In particular, *s*View proved useful for separating personal and common information in a way that protects user privacy: the GeoNotes server is restricted to holding common information about the posted notes, whereas the personal information is kept within the user's (mobile) personal service environment.

7. Conclusions

We have shown that the personal service environment concept provides a powerful tool for handling electronic services. The key issues are openness and user control. Systems based on personal service environments support adding and removing services and users without affecting others. Furthermore, such systems are focused on creating a personal space that the user has full control over.

The *s*View system is a fully functional implementation that is based on the personal service environment concept, as well as a specification of how to implement services, which proves that the concept is realistically feasible. Even though the *s*View system is only a prototype intended as a research tool, we are satisfied with its performance and we have used it for internal development in several projects [24, 25, 27-29].

Our solution is deliberately thin and generic tools are needed in order to make it accessible and useful. However, the core *s*View specification is flexible enough to allow for almost unlimited development of such extensions.

8. Acknowledgements

The work presented in this paper has been funded by The Swedish Institute for Information Technology (www.siti.se). Thanks to the members of the HUMLE laboratory at the Swedish Institute of Computer Science (www.sics.se/humle), in particular Fredrik Espinoza, for inspiration and thoughtful comments.

References

[1] A. Dearle, "Toward Ubiquitous Environments for Mobile Users," *IEEE Internet Computing*, vol. 2, pp. 22-32, 1998.

[2] L. Kleinrock, "Nomadicity anytime, anywhere in a disconnected world," *Mobile Networks and Applications*, vol. 1, pp. 351-357, 1996.

[3] B. Myers, S. E. Hudson, and R. Pausch, "Past, present, and future of user interface software tools," *ACM Transactions on Computer-Human Interaction*, vol. 7, pp. 3-28, 2000.

[4] J. S. Hansen, T. Reich, B. Andersen, and E. Jul, "Dynamic Adaptation of Network Connections in Mobile Environments," *IEEE Internet Computing*, vol. 2, pp. 39-47, 1998.

[5] N. Davies, A. Friday, G. S. Blair, and K. Cheverst, "Distributed Systems Support for Adaptive Mobile Applications," *Mobile Networks and Applications*, vol. 1, 1996.

[6] E. Volokh, "Personalization and Privacy," *Communications of the ACM*, vol. 43, pp. 84-88, 2000.

[7] T. Berners-Lee, R. Caillau, J.-F. Groff, and B. Pollermann, "World-Wide Web: The Information Universe," *Electronic Networking: Research, Applications and Policy*, vol. 2, pp. 52-58, 1992.

[8] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple Object Access Protocol (SOAP) 1.1," World Wide Web Consortium, W3C Note 27 July 1999, May 8, 2000.

[9] A. Di Stefano and C. Santoro, "NetChaser: Agent Support for Personal Mobility," *IEEE Internet Computing*, vol. 4, pp. 74-79, 2000.

[10] J. E. White, "Mobile Agents," in *Software Agents*, J. M. Bradshaw, Ed. Menlo Park, CA: AAAI Press/MIT Press, ISBN 0-262-52234-9, 1997, pp. 437-472.

[11] A. Waern, M. Tierney, Å. Rudström, and J. Laaksolahti, "ConCall: An information service for researchers based on EdInfo," Swedish Institute of Computer Science, Kista, T98-04, 1998.

[12] M. Tierney, "ConCall: An Exercise in Designing Open Service Architectures," Ph.Lic. thesis, The Royal Institute of Technology and Stockholm University, Stockholm, Sweden, 2000.

[13] N. Minar, M. Gray, O. Roup, R. Krikorian, and P. Maes, "Hive: Distributed Agents for Networking Things," presented at First International Symposium on Agent Systems and Applications, Third International Symposium on Mobile Agents featuring the Third Dartmouth Workshop on Transportable Agents, Rancho Las Palmas Marriott's Resort and Spa, Palm Springs, CA, 1999.

[14] C. Pullela, L. Xu, D. Chakraborty, and A. Joshi, "A Component Based Architecture for Mobile Information Access," Department of Computing Science and Electrical Engineering, University of Maryland Baltimore County, Technical Report, TR-CS-00-05, March 31, 2000.

[15] H. L. Chen, "Developing a Dynamic Distributed Intelligent Agent Framework Based on the Jini Architecture," M.Sc. thesis, University of Maryland Baltimore County, Baltimore, 2000.

[16] T. Sandholm and Q. Huai, "Nomad: Mobile Agent System for an Internet-Based Auction House," *IEEE Computer*, vol. 4, pp. 80-86, 2000.

[17] "OSGi Service Gateway Specification Release 1.0," Open Services Gateway Initiative, May, 2000.

[18] "Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); Mobile Station Application Execution Environment (MExE); Functional description; Stage 2," European Telecommunications Standards Institute, ETSI TS 123 057 v.3.0.0, January, 2000.

[19] “Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); Mobile Station Application Execution Environment (MExE); Service description; Stage 1,” European Telecommunications Standards Institute, ETSI TS 122 057 v.3.0.1, January, 2000.

[20] F. Reynolds, J. Hjelm, S. Dawkins, and S. Singhal, “Composite Capability/Preference Profiles (CC/PP): A user side framework for content negotiation,” World Wide Web Consortium, W3C Note 27 July 1999, July 27, 1999.

[21] M. Bylund, “*s*View - Architecture Overview and System Description,” Swedish Institute of Computer Science, Kista, Sweden, SICS Technical Report T2001:06, May, 2001.

[22] M. Boman, “Implementing services for a PSE,” M.Sc. thesis, Uppsala University, Uppsala, Sweden, 2000.

[23] “The Jalda Charging API, Release 1.3,” EHPT Sweden AB, available at: http://www.jalda.com/ [2001, April 18], 2000, May.

[24] F. Espinoza and O. Hamfors. “ServiceDesigner: Enabling End-Users Access to Web Services,” Unpublished manuscript, available at: http://sview.sics.se, 2001.

[25] S. Nylander and M. Bylund. “Providing Universal Device Access to Mobile Services,” Unpublished manuscript, available at: http://sview.sics.se, 2001.

[26] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster, “UIML: an Appliance-Independent XML User Interface Language,” *Computer Networks*, vol. 31, pp. 1695-1708, 1999.

[27] F. Espinoza, P. Persson, A. Sandin, H. Nyström, E. Cacciatore, and M. Bylund, “GeoNotes: Social Filtering of Position-Based Information,” Swedish Institute of Computer Science, SICS Technical Report T2001:08, May, 2001.

[28] H. Nyström and A. Sandin, “Social Mobile Services in an Open Service Environment - an Overview, Analysis and Implementation,” M.Sc. thesis, Uppsala University, Uppsala, Sweden, 2001.

[29] P. Persson, F. Espinoza, and E. Cacciatore, “GeoNotes: Social Enhancement of Physical Space,” presented at CHI'2001, Seattle, WA, 2001.

Paper C

*s*View – Architecture Overview and System Description

Markus Bylund

Swedish Institute of Computer Science, Box 1263, SE-164 29 Kista, SWEDEN
bylund@sics.se

Abstract. This report presents an architecture overview and a system description of the *s*View system. The system provides developers, service providers, and users of electronic services with an open and extendible service infrastructure that allows far-reaching user control. This is accomplished by collecting the services of an individual user in a virtual briefcase. Services come in the form of self-contained service components (i.e. including both logic and data), and the briefcase is mobile to allow it to follow as the user moves between different hosts and terminals. A specification of how to build such service components and the infrastructure for handling briefcases is presented. A reference implementation of the specification as well as extensions in the form of service components is also described. The purpose of the report is to serve as a technical reference for developers of sView services and software infrastructure that builds on sView technology.

Keywords. Electronic services, personal service environments, user interfaces, mobility, personalization, service collaboration, component-based software engineering.

May 2001
SICS Technical Report T2001/06
ISSN 1100-3154
ISRN: SICS-T--2001/06-SE

1. Introduction

The use of electronic services is spreading more and more to an increasingly broader group of users, and there is a growing need for support for continuous interaction with multiple services, via different types of devices, and from all sorts of places and locations. Further more, it is desirable that this is done in a way that assures the user control over personal information that services gather and maintain. The user should also be able to control what services do and whether or not, and how, they collaborate with each other.

All these demands represent current research topics such as privacy in the context of electronic service usage, service collaboration, and ubiquitous user interface design. The *s*View system has been designed as a solution to some of these research topics, and to cater for further research on others. The system assumes a client server model. But instead of having a uniform client without service specific functionality for access to all servers (as in the case with the world wide web), access to the servers is channeled through a virtual service briefcase. The briefcase in turn, supports access from many different types of devices and user interfaces. It is also private to an individual user, and it can store service components containing both service logic and data from service providers. This allows the service provider to split the services in two parts. One part with commonly used functionality and user specific data that executes and is stored within the virtual briefcase. The other part provides network-based functionality and data that is common between all users. Finally, the service briefcase is mobile and it can follow its user from host to host. This allows local and partially network independent access to the service components in the briefcase.

At a high level, the *s*View system consists of two parts. The *core sView specification* provides APIs (Application Programming Interfaces) to developers of service components and service infrastructure that builds on *s*View technology. Implementing these APIs and adhering to the design guidelines that accompany the APIs, assures compatibility between *s*View services and service infrastructure of different origin. The *sView reference implementation* provides developers with a development and runtime environment for service components as well as a sample implementation of an *s*View server

The report is structured as follows. Section 2 describes a number of basic concepts and entities. Section 3, specifies the main requirements that has influenced the design of the *s*View system. Section 4 provides a detailed description of the core *s*View specification. Section 5 describes the *s*View reference implementation, and Section 6 concludes with a summary.

2. Basic Concepts

The *s*View system builds on the concept of *personal service environments* [1]. A personal service environment is an individually collected and tailored set of services, available to the user at all times. The services are retrieved from service providers around the Internet, but after retrieval they are at least partially independent of Internet access. The personal service environment itself is mobile, following its user around in the network. The interaction state of the services is saved as the personal service environment moves between hosts on the Internet. This allows for continuous interaction sessions as the user of the services switches between different interaction devices. In the reminder of this text, the personal service environment is referred to as the service environment or simply the environment.

The *s*View system defines three main entities: *service components*, *service briefcases*, and *service briefcase servers*.

- A service component is an entity that provides *services* to the user, and/or other service components within the same service environment. It is a collection of class definitions and resources that together define a component that can be loaded and executed in a personal service environment. This allows service components to collaborate about e.g. content provision, personalization, and user interface handling.
- A service briefcase is a data structure in which a personal service environment is stored. A service briefcase contains service component definitions, saved execution states of service components, and settings. It also includes functionality for loading, saving, and creating new service components.
- A service briefcase server constitutes an API that offers service briefcase handling such as create new service briefcase, start service briefcase (i.e. create a personal service environment based on a service briefcase), and synchronization between instances of a service briefcase on different service briefcase servers.

An illustration of the different parts of the *s*View system and their relations is given in **Fig. 1**. On the computer marked I a briefcase server and a service environment is executing. In this case the user is sitting next to the same computer as the service environment (represented by the cloud together with service components A, B, and C) is executing on. This makes it possible to use a standard GUI for user-service interaction. The computer marked II hosts service briefcases and environments for several users, which use remote interfaces. One user is using a web-kiosk with a web browser for user-service interaction (III) and another user uses a WAP phone (IV). Stored service environments, in the form of service briefcases (illustrated between I and II), can migrate between any computers that run a briefcase server.

Finally, in the reminder of this document I will refer to an *sView server* as a combination of a service briefcase server and the server software with which personal service environments execute.

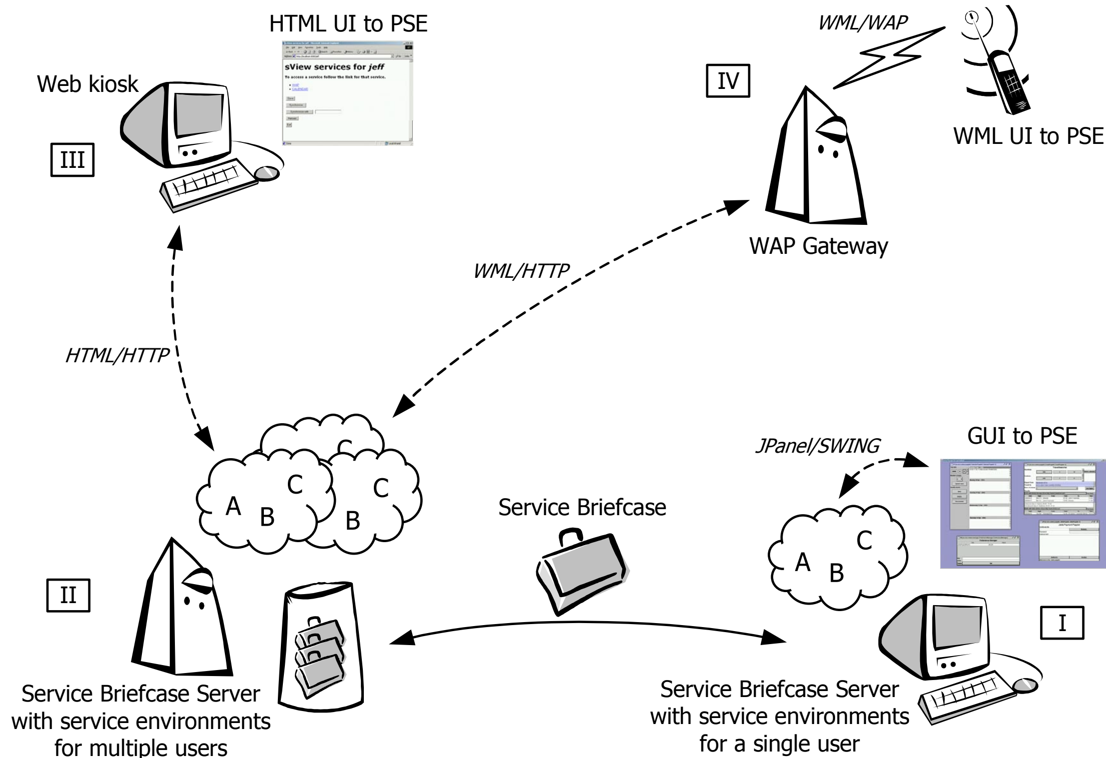


Fig. 1. The main entities of the *s*View system and their relations.

3. Design Requirements

The high-level design goals with the *s*View system have been specified as *openness* and *user control* [1]. Openness implies that it should be possible to add service components and users to the system without affecting other parts of the system. User control implies that the system should give the user control over which services to use, what information about the user that services handle, how services collaborate, etc. Some users may chose not to use make use of this control, but the possibility for user control should always exist. Furthermore, the user should be in control of the usage situation. In practice this means that services should be reachable from everywhere using many different types of devices, both the user's own devices and publicly available devices.

Openness and user control can be further analyzed in the terms of five more specific requirements: *heterogeneity*, *extendibility*, *accessibility*, *adaptability*, and *continuity*. The three former requirements are closely related to the personal service environment concept, and are discussed in more detail in [1]. The two latter requirements however, heterogeneity and extendibility, have had a profound impact on the design and implementation of *s*View and are discussed further below.

3.1. Heterogeneity

Many electronic services already exist, both in the form of commercial and research products. A sound requirement on an open infrastructure for user-service interaction is to allow a heterogeneous mix of service components to utilize features of each other. We approach the requirement on heterogeneity in a number of ways.

The *s*View system is implemented in Java. This brings at least two advantages: a reasonable chance of creating a platform independent system and easy integration of other Java based electronic service infrastructures [2-5]. The *s*View system puts few constraints on the implementation of the service components, which makes the integration of other service infrastructures straightforward.

Developers of an *s*View service component can chose between implementing all of the functionality in the service component, or placing all functionality on a server in the network (in which case the *s*View service component only serves as a proxy to the network based functionality). Any combination of the two alternatives is also possible. This allows for integration of already existing network-based services into the *s*View system.

*s*View service components are free to communicate with external resources (such as network-based services) using any protocol of their like. This communication is not in any way limited by the *s*View system.

3.2. Extendibility

Openness also implies a demand for extendibility. As new services are added to the system it should be possible to add support for new protocols. This would make it possible to add functionality for user-service interaction, communication between service components, collaboration between services of different kinds, enhanced security handling, etc. With the current design of *s*View, we approach the requirement on extendibility in three different ways.

Firstly, the functionality of an *s*View service component need not be targeted towards the user of the system, but can instead provide functionality to other service components in the user's service environment. This makes it easy to extend the *s*View system with new functionality. For this purpose, it is useful that *s*View service components can build on, and include in its distribution, existing Java packages.

Secondly, the *s*View system is separated in two parts: a core specification and a reference implementation. The core specification includes the APIs that are necessary in order to implement *s*View servers that are compatible with each other. The API also ensures that all *s*View service components are executable in any implementation of an *s*View server.

Thirdly, the core specification includes a method for *s*View servers to dynamically load new implementations of server-server communication protocols. *s*View servers can therefore communicate in any protocol that can be implemented in Java.

Class/*Interface*	**Service Component**	**Server Functionality**
Constants	Can implement/Must use	Must use
Mobile	Can implement	Must use
Monitor		Must use
Persistent	Can implement	Must use
ServerProxy		Can implement/Must use
ServiceBriefcase		Must use
ServiceBriefcaseServer		Must implement
ServiceComponent	Must implement	Must use
ServiceComponentPermission		Must use
ServiceContainer		Must use
ServiceContext		Must implement
ServiceInterfaceFactory	Can implement	Must use
ServiceListener		Must implement
ServiceProxy		Can implement/Must use
TransactionCoordinator		Can implement/Must use
TransactionInitiator		Can implement/Must use
TransactionParticipant		Can implement/Must use

Table 1. The main classes and interfaces in the core specification (`se.sics.sview.core`).

4. The Core *s*View Specification

The core *s*View specification consists of about 60 Java classes and interfaces that are needed in order to implement service components and *s*View servers. The total size of the specification is less than 40 KB. **Table 1** lists the most important classes and interfaces and relates them to either service components or server functionality. See Appendix I for a full listing of all classes and interfaces.

4.1. API Overview

The basic architecture of the core *s*View specification can be described with four entities (see **Fig. 2**): service component, service briefcase, service briefcase server, and service context.

The three former entities were briefly described in Section 2. The latter entity, the service context, constitutes the context in which a service component executes. The service context offers a service component an API that allows the component to e.g. register services, subscribe to other services, and manage other service components.

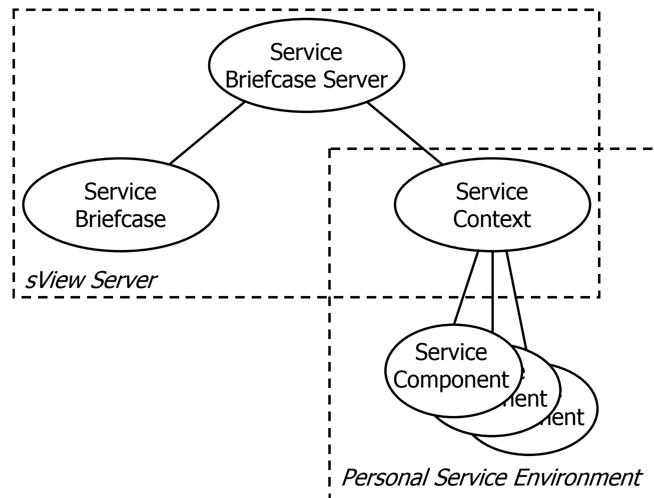


Fig. 2. Overview of the basic architecture of the core *s*View specification.

4.2. Service Component

An *s*View service component is created by implementing the Java interface `se.sics.sview.core.ServiceComponent`. The class definitions of the service component needs to be packaged in a JAR file together with a manifest with information about the component. During runtime, the service component follows a lifecycle that is defined by a set of states and a state transition graph. Finally, the service component can be extended to allow persistence and mobility.

Packaging and Distribution of a Service Component. A service component is packaged and distributed as a JAR file [6]. All class definitions and resources of the service component should be included in this file, as well as information about e.g. the name and structure of the service component.

The most important part of the JAR file is the set of class files that define the functionality of the service component. Class files can be included in the JAR file in two ways: either directly as main JAR entries (which is the usual way), or in nested JAR files. The latter way is convenient if the service component depends upon external packages in JAR files. If classes are included in this way an internal class path must be given in the JAR manifest (see below).

If the service component registers services for other service components to use, it should export class definitions from its own JAR file to these other service components. Every class or interface definition that is needed in order to use the service should be exported. In the simplest case, only a single interface is needed, but for more advanced services whole packages might have to be exported. Class definition exports are specified in the JAR manifest (see below).

Resources in the form of images, databases or just about anything that can be stored in a file can be included in the JAR file. Upon request, resources are made available to the service component (via the service context) in the form of byte arrays.

```
Manifest-Version: 1.0
ServiceComponentName: Sample 1 Service Component
ServiceComponentActivator: Sample1
ServiceComponentClasspath: ., javamail.jar, servlet.jar
ServiceComponentExport: Sample1ServiceInterface
ServiceComponentPermission: ServiceComponent, ServiceEnvironment
```

Listing 1. An example of a JAR manifest of a service component. When listing permissions from the package `se.sics.sview.core.permission`, the package name can be omitted.

The JAR file of a service component must contain a manifest with information about the service component. Following is a list of entries that can (must) be specified in the manifest.

- `ServiceComponentName` (*mandatory*) – a symbolic name of the service component.
- `ServiceComponentActivator` (*mandatory*) – the fully qualified class name of the class of the service component that implements the interface `se.sics.sview.core.ServiceComponent`.
- `ServiceComponentClasspath` – the internal class path of the JAR file. Should be a comma separated list of JAR entries (being themselves JAR files) or '.' (which stands for the classes in the root JAR file). List entries are searched for class definitions in order of appearance.
- `ServiceComponentExport` – a comma separated list of package names or fully qualified class names that should be exported to other service components.
- `ServiceComponentDepend` – a list of names of services (offered by other service components) that this service component depends upon.
- `ServiceComponentPermission` – a list of permissions that grants the service component rights to functionality in the system (see Section 4.3).

An example of a JAR manifest for a service component is given in **Listing 1**. The JAR file includes two nested JAR files (javamail.jar and servlet.jar) that are both included in the service component classpath. The service component also exports a class definition: the class `Sample1ServiceInterface`. Finally, the service component is given two permissions: `ServiceComponent` and `ServiceEnvironment`.

Service Component Lifecycle. The lifecycle of a service component is described by a set of states and a state transition graph (see **Fig. 3**). Half of the transitions are initiated by the service context and the other half by the service component. Service context initiated state changes always occur as a result of the service context calling one of the methods of the service component (initialize, start, suspend, resume, or stop). Service component initiated transitions can occur in one of two ways. The service component either initiates the state change by returning the value of the new state from the methods that the service context calls, or if the state change should be delayed after returning from the method, by explicitly setting the state by calling the `setState` method of the service context.

Following is a description of the different states of the service component.

- `INACTIVE` – The service component is either newly created and not yet added, or recently removed from, a service environment. In this state the service component

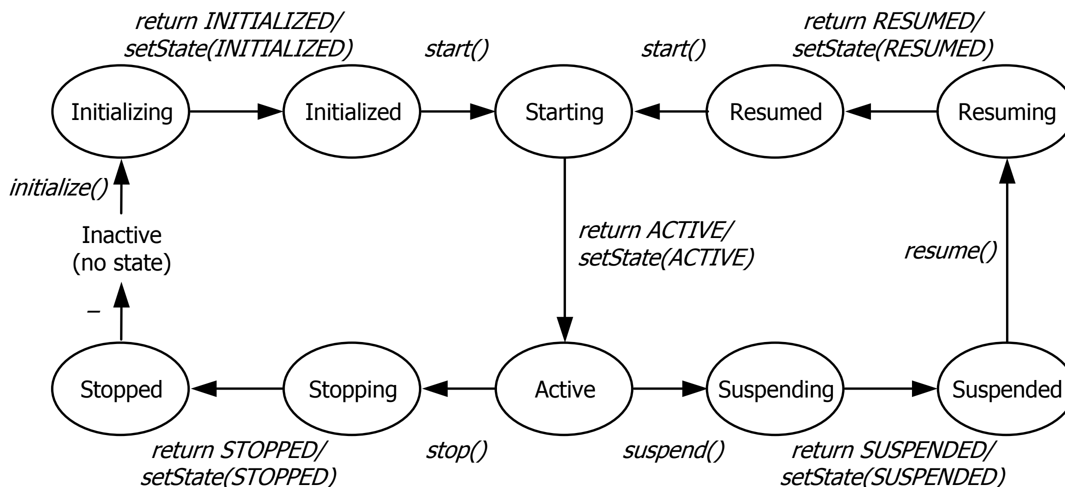


Fig. 3. The state graph describing the lifecycle of a service component.

is not allowed to interact with either its service context or with other service components.

- `INITIALIZING` – The service component automatically reaches this state when the service context calls the `initialize` method of the service component. This is done as a first step to add the component to the service environment. In this state, the service component is expected to perform initialization that is only done once during the lifetime of a service component. This is the first chance of the service component to interact with the service context, but interaction with other service components is not allowed yet. The service component signals that initialization is done either by having the `initialize` method return `INITIALIZED`, or, if initialization continues after returning from the `initialize` method, by calling `setState(INITIALIZED)` on the service context. In the latter case a negative number should be returned by the `initialize` method to signal that initialization is not finished.
- `INITIALIZED` – The service component reaches this state when it has finished initialization.
- `STARTING` – The service component automatically reaches this state when the service context calls the `start` method of the service component. In this state, the service component should perform tasks that should be done every time it is about to start. Interaction with the service context is allowed, but not with other service components. The service component signals that starting is done either by having the `start` method return `ACTIVE`, or, if starting continues after returning from the `start` method, by calling `setState(ACTIVE)` on the service context. In the latter case a negative number should be returned by the `start` method to signal that starting is not finished.
- `ACTIVE` – The service component reaches this state when it has finished starting. This is the state where most of the lifecycle of a service component is spent. The service component is allowed to interact with both the service context and other service components from here.

- `SUSPENDING` – The service component automatically reaches this state when the service context calls the `suspend` method of the service component. This is done as a first step to suspend the component. In this state, the service component is expected to unregister all services that it offers other service components, as well as unsubscribe to services of other service components. The service component is allowed to interact with the service context in this state. It is also allowed to interact with other service components, but only for the purpose of handling unsubscriptions and unregistrations. The service component signals that suspension is done either by having the `suspend` method return `SUSPENDED`, or, if suspension continues after returning from the `suspend` method, by calling `setState(SUSPENDED)` on the service context. In the latter case a negative number should be returned by the `suspend` method to signal that suspension is not finished.
- `SUSPENDED` – The service component reaches this state when it has finished suspension. In this state the service component is not allowed to interact with either its service context or other service components. The service component can now be saved to persistent media or moved to another server.
- `RESUMING` – The service component automatically reaches this state when the service context calls the `resume` method of the service component. This is done as a first step to resume the component after suspension. This state is comparable to the `INITIALIZING` state, with the exception that the state can occur more than once. The service component signals that resumption is done either by having the `resume` method return `RESUMED`, or, if resumption continues after returning from the `resume` method, by calling `setState(RESUMED)` on the service context. In the latter case a negative number should be returned by the `resume` method to signal that resumption is not finished.
- `RESUMED` – The service component reaches this state when it has finished resumption.
- `STOPPING` – The service component automatically reaches this state when the service context calls the `stop` method of the service component. This is done as a first step to stop the component. In this state, the service component is expected to unregister all services that it offers other service components, as well as unsubscribe to services of other service components. The service component is allowed to interact with the service context in this state. It is also allowed to interact with other service components, but only for the purpose of handling unsubscriptions and unregistrations. The service component signals that stopping is done either by having the `stop` method return `STOPPED`, or, if stopping continues after returning from the `stop` method, by calling `setState(STOPPED)` on the service context. In the latter case a negative number should be returned by the `stop` method to signal that stopping is not finished.
- `STOPPED` – The service component reaches this state when it has finished stopping. In this state the service component has reached the end of its lifecycle. Only a reload of a previously saved copy or creating a new instance of the service component can bring the service component back to the service environment. In this state the service component is not allowed to interact with either its service context or other service components.

```
import se.sics.sview.core.*;
import se.sics.sview.core.event.*;

public class Sample1 implements Constants, ServiceComponent, Runnable {
  Thread ct;
  ServiceContext sc;
  ServiceContextEvent ce;

  // Implementations of interface ServiceComponent

  public int initialize(ServiceContext context, ServiceContextEvent evt) {
    // do initialize here - NOT computation intensive
    return INITIALIZED;
  }

  public int start(ServiceContext context, ServiceContextEvent evt) {
    // do start here
    sc = context;
    new Thread(this).start();
    return -1;
  }

  public int suspend(ServiceContext context, ServiceContextEvent evt) {
    if (ct==null) {
      // already suspended
      return SUSPENDED;
    } else {
      interrupt(context, evt);
      return -1;
    }
  }

  public int resume(ServiceContext context, ServiceContextEvent evt) {
    // do resume here - NOT computation intensive
    return RESUMED;
  }

  public int stop(ServiceContext context, ServiceContextEvent evt) {
    if (ct==null) {
      // already stopped
      return STOPPED;
    } else {
      interrupt(context, evt);
      return -1;
    }
  }

  // Implementations of interface Runnable

  public void run() {
    sc.setState(ACTIVE);
    ct = Thread.currentThread();

    try {
      // do run here - computation intensive
    } catch (InterruptedException e) {
      if (ce instanceof SuspendEvent) {
              // do suspend here - computation intensive
              sc.setState(SUSPENDED);
      } else if (ce instanceof StopEvent) {
              // do stop here  - computation intensive
              sc.setState(STOPPED);
      }
    }
    ct = null;
  }

  // Misc.

  public void interrupt(ServiceContext context, ServiceContextEvent evt) {
    ce = evt;
    sc = context;
    ct.interrupt();
  }
}
```

Listing 2. A sample implementation of a threaded service component with state handling.

```
import se.sics.sview.core.*;
import se.sics.sview.core.event.*;

public class Sample2 implements Constants, ServiceComponent, Persistent {
  Vector users = new Vector(42);             // a vector for user information
  Hashtable mediaCache = new Hashtable();    // a media cache for video clips

  <snip>

  // Implementations of interface Persistent

  public void freeze() {
    users.trimToSize();                      // compact the vector of users
    mediaCache = null;                       // remove the media cache
  }

  public void thaw() {
    mediaCache = new Hashtable();            // recreate the media cache
  }
}
```

Listing 3. An example of an implementation the interface `Persistent`.

Listing 2 gives an example of the state handling of a threaded service component. The purpose is to have code that requires lots of computation (in the example the calls to `start`, `suspend`, and `stop`) execute in a separate thread. Initialization and resumption, which are not computation intensive in the example, are run from the thread of the service context (i.e. within the call to `initialize` and `resume`). In the `start` method, the thread of the service component is started, but state `ACTIVE` is not entered until the service component executes in its own thread. Suspension and stopping is also handled within the thread of the service component, but only if `suspend` or `stop` are called while the thread of the service component is running. Otherwise it is handled in the same way as `initialize` and `resume`.

Persistence and Mobility. Service components can be made persistent and have its execution state (as a serialization of the objects that constitute the service component) saved in the service briefcase. They can also be made mobile which means that they will follow the service briefcase as it migrates between servers.

A service component is made persistent by implementing the interface `se.sics.sview.core.Persistent`. This requires the service component to implement two methods: `freeze` and `thaw` (see **Listing 3** for an example).

The service briefcase calls the `freeze` method when it saves the service component. This occurs after the service component has reached state `SUSPENDED`, but before state `RESUMING` is reached. The `freeze` method should be used to prepare for serialization by optimizing or removing data structures. The service component could e.g. compact a hash table or empty a media cache for more efficient storage After returning from the `freeze` method all external references (such as references to the service context, file and socket handles etc.) must have been set to `null`[1].

The service briefcase calls the `thaw` method when a saved version of the service component is loaded. This occurs after the `freeze` method has been called (possibly in a different VM and on a different host), but before state `RESUMING` is reached. The `thaw` method should be used to, if needed, recreate data structures that were removed

[1] Unless the reference is declared as `transient`.

Mobile / Persistent	Yes	No
Yes	Follows the user and preserves its state (e.g. a calendar).	Does not follow the user but preserves its state (e.g. a printer queue).
No	Follows the user but does not preserve its state (e.g. a proxy to a web based service).	Does not follow the user nor preserve its state (e.g. a driver for a loudspeaker).

Table 2. Examples of four types of service components.

or converted in the `freeze` method. It should also be used to re-associate references that were set to `null` in the `freeze` method or during serialization.

A service component is made mobile by implementing the marker interface[2] `se.sics.sview.core.Mobile`. This will allow the service briefcase to include the service component when migrating to other hosts.

The properties of service component persistence and mobility are orthogonal. A persistent service component that is not mobile can save its state locally, but not migrate to a different node. A mobile service component cannot save its state, neither locally nor while migrating; every time such a service component is restarted it will start from scratch (which is fine for many services). The most powerful service component however combines the two properties. Such a service component can both save its state and migrate. **Table 2** lists and exemplifies the four possible combinations of the two properties.

4.3. Service Context

The service context provides runtime handling of service components. It controls the lifecycle of service components by setting the states of the components. While doing so, the context informs the service component about the reason for the state change by sending an event. The service context gives service components access to three different kinds of properties (simple databases for storing settings). The context also provides an API for communication between service components, as well as handling of other service components and even the server itself. For the latter part, service components needs privileges that are granted to the component based on permissions.

Events. The service context controls the state of service components by calling the methods `initialize`, `start`, `suspend`, `resume`, and `stop` on the activator objects of the components. These methods take two arguments: the first is a reference to the service context itself. The second is an event, a subclass of `se.sics.sview.core.ServiceContextEvent`, with information about the reason

[2] A marker interface is an interface with an empty body. The purpose of such an interface is to signal that the implementation of the interface should be considered to have a certain property. In this case to be mobile.

behind the state change. The service component might want to use this information when deciding how to act upon the state change.

There are three main types of events.

- `StartEvent` – is used to take the service component from state `INACTIVE` and `RESUMED` through all the states to `ACTIVE`. Examples of this event include `CreateEvent` and `LoadEvent`.
- `SuspendEvent` – is used to take the service component from all states except `INACTIVE`, `STOPPING`, and `STOPPED` through the states to `SUSPENDED`. Examples of this event include `SaveEvent` and `SynchronizeEvent`.
- `StopEvent` – is used to take the service component from all states except `INACTIVE` and `STOPPED` to the state `STOPPED`. Examples of this event include `RemoveEvent` and `ReloadEvent`.

The documentation of for the events in package `se.sics.sview.core.event` contains a more detailed description of the information that individual events carry (see Appendix I).

Properties. The service context administers three sets of properties for storing settings of different kinds.

- *Local properties* deal with settings that are shared between all personal service environments on a particular server (such as references to means of interacting with the user from the server). Local properties cannot be set or changed by the service context or individual service components; instead, the administrator of the server controls these properties.
- *Stationary properties* are not shared between users, but they are still bound to a particular server. They can for example store settings such as the user's UI preferences, which is likely to differ between servers. Stationary properties can be created and modified by the service context and individual service components.
- *Mobile properties* are personal, just like stationary properties, but in contrast they do not vary with server. Mobile properties typically deal with settings that are not location dependent (e.g. user information such as name, address, etc.).

The three types of properties are convenient for pushing server settings to service components (local properties) and for saving and sharing information between service components (stationary and mobile properties).

Service Component Communication. Service components can communicate and collaborate by offering services to each other. The establishment of a service provider/consumer relationship is described in **Fig. 4**, in which service component A (SCA) offers service component B (SCB) a service.

I. The process is initiated by SCA by registering its service (S1) to the service context, during which SCA passes two parameters: a name of the service and a *service interface factory*. The latter should be an implementation of the interface `se.sics.sview.core.ServiceInterfaceProxy`, which is used by the service context to create interfaces to the service.

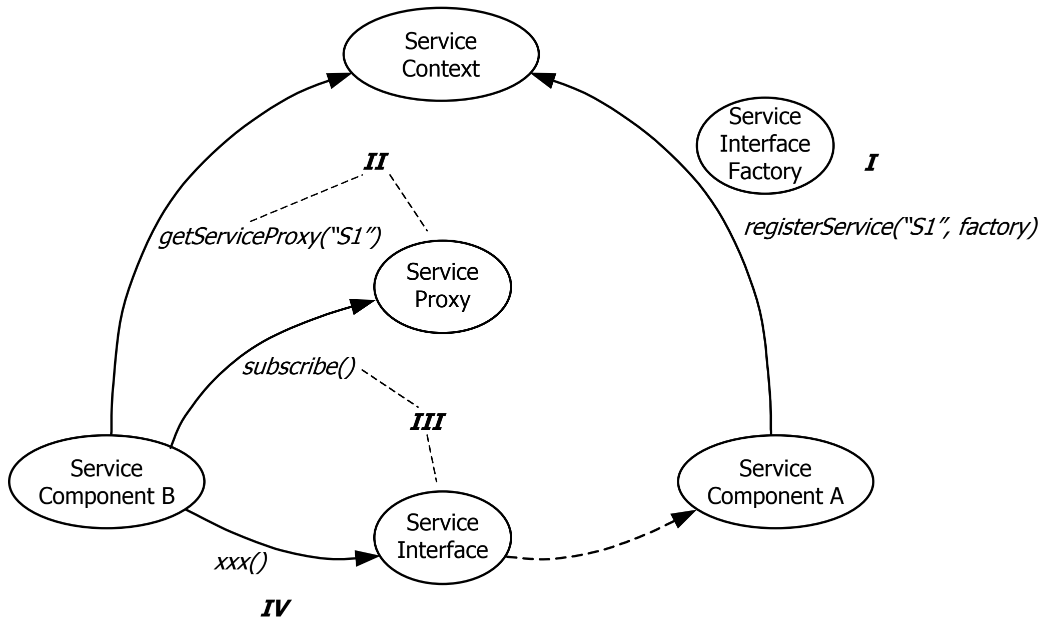


Fig. 4. A graph describing the establishment of a service provider–consumer relationship.

II. SCB requests a proxy to the service that SCA registered from the service context. The service proxy is of the same type regardless of which service is requested (`se.sics.sview.core.ServiceProxy`).
III. Using the service proxy, SCB starts a subscription to service S1. The service proxy uses the service interface factory that SCA provided to create an interface the service.
IV. SCB can now use service S1 by invoking methods on the service interface. Note that SCB needs to know the type of the service interface for this scheme to be effective.

The above description is only an example of how a relationship can be established; alternative ways are also possible. Phase II could for example happen before phase I (even without the existence of SCA). In such a case, SCB could attempt to start a subscription to a service that was not registered already, resulting in a `null`-reference instead of a reference to a service interface. The service proxy includes functionality for handling such situations. SCB could for example specify that if the requested service is not registered, the call should wait until it is. The service component could also register itself as a listener to (un)register notifications of the service, in which case SCB would know when to start the subscription.

The providing service component can unregister its services at any time. Consumers of those services are then obliged to unsubscribe and to stop using the services as soon as possible.

Server and Service Component Handling. The service context provides an API that allows service components to handle its server as well as other service components. The API lets service components reload, save, and synchronize the service environment, as well as shutting down the service environment (or, in the case of a single-user server, the server).

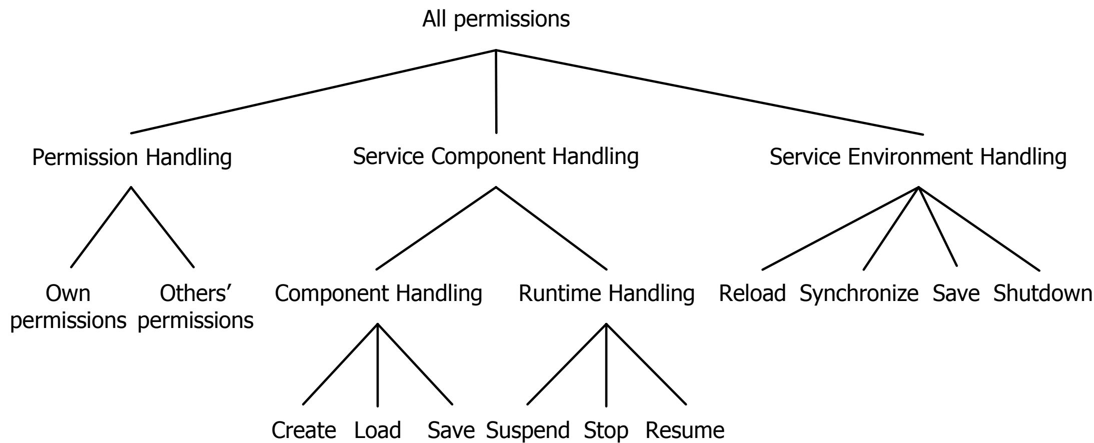


Fig. 5. The hierarchy of permissions for accessing service context functionality.

The API allows service components to create, load, and save other service components. It also allows service components to partially control the lifecycle of other components in that they can force other components to suspend, resume, and stop[3].

Finally, service components can also control their own lifecycle with this API by requesting the service context to start, suspend, resume, and stop the service component.

Permissions. The service context functionality for handling the service environment and other service components is protected by permissions. Permissions specifications are included in the core specification, and they are arranged in a hierarchy so as to allow both specific and general permissions (see **Fig. 5**). By extending an interface with several permissions at the same time, combinations of permissions can be implemented.

Permissions for a service component are specified by a comma-separated list of the class names that corresponds to the permissions should be included in the JAR manifest (see **Listing 1**).

4.4. Service Briefcase

The service briefcase (`se.sics.sview.core.ServiceBriefcase`) contains functionality for creating, loading and saving service components. It also provides storage of the JAR files of service components, persistent service components, and properties.

The service briefcase is serializable and it can be stored on persistent media and sent between servers, or have its content synchronized with service briefcases on other servers.

Much of the functionality of the service briefcase is delegated to *service containers* (`se.sics.sview.core.ServiceContainer`), of which there is one for each

[3] Initialize occurs implicitly as a result of adding a component to the service environment and start occurs automatically whenever a component has finished initializing or resuming.

service component in the briefcase. The service container provides storage and serialization handling of individual service components. It includes functionality for creating, loading, and saving service components, storing persistent service components, and caching the JAR file of service components.

Service component creation and loading requires that a class loader is provided by the server implementation. The server typically uses separate class loaders for every service component in the system. This ensures that no service component should be able to manipulate other service components without permission.

Service Briefcase State. An important step when synchronizing content between different service briefcases is to compare the *state of service briefcases*. The service briefcase state includes the names, keys, creation dates, change dates, and JAR status, of every mobile service component in the service briefcase.

User id and password. Most of the functionality of the service briefcase is protected with a user id and a password. Upon creation of the service briefcase, the user has to provide a user id and a password. The user id is used to uniquely identify the owner of the briefcase when moving briefcases between servers or synchronizing content between several instances of the same briefcase. However, *s*View does not provide a method of assigning unique identifiers to every user. Users are instead encouraged to use an already existing unique Internet identifier (such as an e-mail address) as their user id.

The password is encrypted using the MD5 Message-Digest Algorithm [7] and stored in the service briefcase as a 128 bit long ‘fingerprint’ of the password. In order to use the protected functionality, the user has to provide the password, which is encrypted and compared with the original password ‘fingerprint’.

Note that the user id and password by no means represents a complete, or even partially satisfying, protection of the service briefcase. The scheme is merely used as an illustration of the need for protection. A true protection of the service briefcase must include at least two parts: authentication and encryption of the content. This should be implemented as a plugable solution, allowing the user to freely select which implementation, and therefore also which algorithm, for each of the two parts to use.

4.5. Service Briefcase Server

The service briefcase server provides an API for service briefcase handling such as creating new and removing existing briefcases, as well as starting and stopping the execution of personal service environments. The API also includes functionality for moving service briefcases between servers, and for synchronizing content between different instances of a briefcase on different servers. The API is specified as a Java interface (`se.sics.sview.core.ServiceBriefcaseServer`).

Server-Server Communication. Since this server is designed to communicate with servers on other hosts, a Java interface will not be sufficient for most purposes. What is missing is a protocol that is capable of wrapping the server interface (e.g. Java RMI

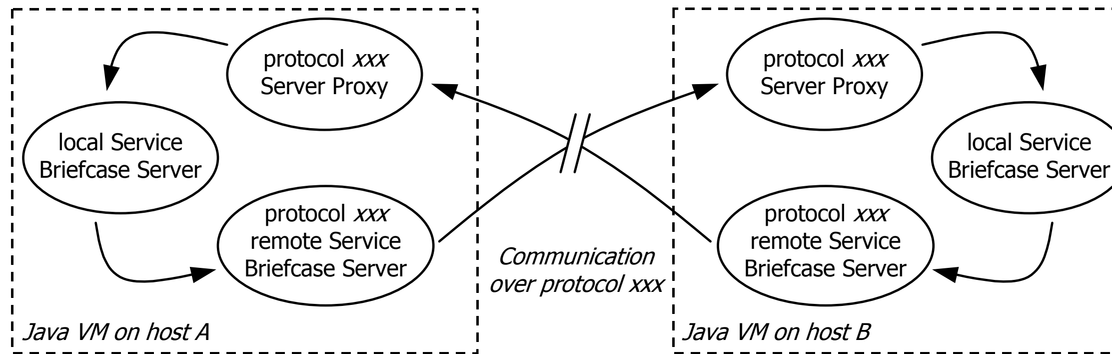


Fig. 6. A graph describing communication between two service briefcase servers over the fictive protocol *xxx*.

[8], SOAP[9], and HTTP [10]). However, every such protocol has its special characteristics with both strengths and weaknesses, and it would be impossible to select one or a few as the preferred protocols for *s*View. For this reason we have chosen not to specify any protocol at all in the core *s*View specification. Instead, an interface that provides access to implementations of service briefcase servers is specified (`se.sics.sview.core.ServerProxy`).

Note that this solution for server-server communication allows the implementation of different types of secure communication schemes. A server proxy could e.g. implement a protocol for secure authentication to avoid synchronizing service briefcases with fake servers. Different types of channel encryption and protocols to ensure information integrity could also easily be implemented.

Fig. 6 illustrates a communication path between two service briefcase servers. Without knowing anything about the communication protocol itself, local service briefcase servers can establish a communication link by creating instances of the 'protocol *xxx* Server Proxy' (which must implement the `se.sics.sview.core.ServerProxy` interface). Upon request, the server proxy creates the 'protocol *xxx* remote Service Briefcase Server'. In the above example, the server proxy acts as a server for incoming protocol *xxx* communication. It would also be possible let the remote service briefcase server take on this role, in which case the server proxy would only act as a factory for remote service briefcase servers.

Service Briefcase Synchronization. Service briefcase synchronization is a process that involves two or more service briefcase servers, and whose purpose is to synchronize the content of instances of a service briefcase on different servers. Note that this process concerns synchronization of the service briefcase instances of one user at a time. It can be described in a number of steps.

1. The initiating server (the initiator) requests the states of the service briefcase instances on the other servers (the participants).
2. The initiator requests the mobile properties of the service briefcase instances on the participants.
3. Based on its own and the participants states and mobile properties, the initiator generates a new state and a new set of mobile properties that represent the most up-to-date state and mobile property set of the service briefcase.

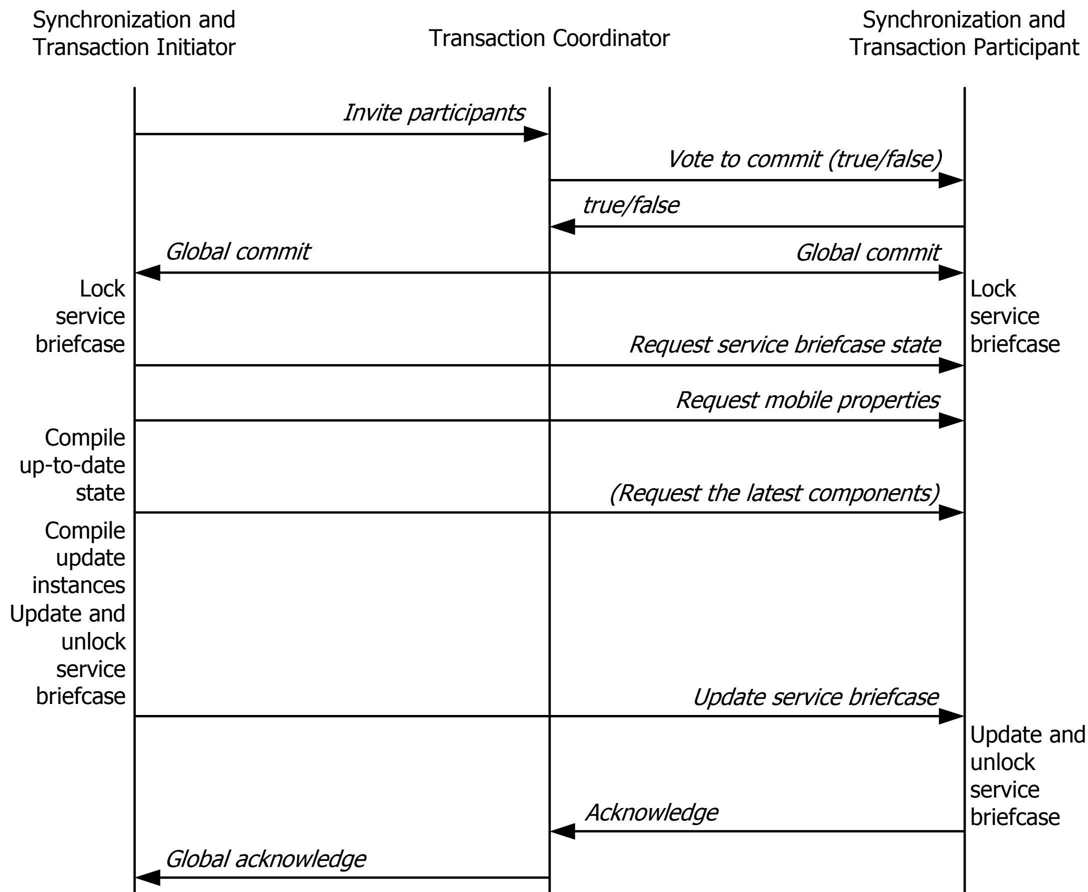


Fig. 7. A diagram describing the synchronization process with transaction handling.

4. The initiator generates a new instance of the service briefcase that reflects the latest state. This may involve requests of service components from one or more participants.
5. The initiator generates update instances of the new service briefcase. This is done exclusively for each participant, taking into account only the information that is needed to make that participant up-to-date.
6. The update instances are sent to the participants.

During this process, it is crucial that the service briefcase instances of the initiator and the participants are not modified, or else consistent behavior cannot be guaranteed. For this purpose, the service briefcase is equipped with a monitor (`se.sics.sview.core.Monitor`) that allows the service briefcase server to prevent modifications of the service briefcase. The monitor is designed to allow for concurrent modification of the service briefcase while unlocked.

It is also important to be able to handle both initiator and participant failure during the synchronization process. This is accomplished by wrapping the synchronization process in a modified version of the two-phase commit transaction protocol.

The whole process (i.e. both synchronization and transaction handling) is described in **Fig. 7**. At any time, the transaction coordinator may send an abort message to both initiator and participants. Participants that receive an abort message before getting the 'Update service briefcase' message will simply reset the transaction and unlock the

```
java.awt                              java.swing.event
java.awt.event                        java.swing.table
java.io                               java.text
java.net                              java.util
java.lang                             java.util.zip
java.swing                            java.util.jar
```

Listing 4. The packages that the reference implementation is built upon.

service briefcase. This will also happen if the response time from either the transaction coordinator or the initiator is too long. Abort messages that participants receive after the service briefcase update are discarded.

If the initiator receives an abort message before the 'Global acknowledge' message, the transaction is reset and the synchronization has to be restarted. However, if the initiator has updated its service briefcase before abort is received, the synchronization process is likely to require fewer steps than otherwise since the initiator has an up-to-date instance of the service briefcase. Note that it does not matter if an abort occurs when some of the participants have updated their briefcases and some have not. The briefcases that have not been updated will be so during a following retry.

5. The *s*View Reference Implementation

The reference implementation of the *s*View system was developed for two purposes. Firstly, it should serve as a development and runtime environment for developers of *s*View service components. Secondly, it should serve as a sample implementation of the core *s*View specification for developers of *s*View server functionality. It is freely available for download from the *s*View web site (http://sview.sics.se/) for everyone to use.

5.1. Current Implementation

The reference implementation is based on J2SE (Java 2 Platform, Standard Edition) version 1.3. The implementation is, apart from the core *s*View specification, only based on a number of packages from the J2SE runtime libraries (see **Listing 4**).

The current version of the reference implementation (version 2.0, alpha 1) supports most of the features of the core sView specification. However, it is not intended as an optimized, secure, and fully scalable runtime environment. The support for such features is therefore limited or non-existent. It is also limited to serving one personal service environment at a time and it does not support briefcase retrieval by date (see `se.sics.sview.core.ServiceBriefcaseServer`). The implementation consists of about 40 classes and its size is less than 125 KB.

5.2. Extensions

For server-server communication, the reference implementation includes an IP socket based implementation of the server proxy communication wrapper (described in Section 4.5). This allows different implementations *s*View servers to communicate over an IP socket based protocol.

In order to be open and customizable, the core *s*View specification does not include UI handling. This is instead left as a task for service components to handle. The reference implementation includes service components that handle user interfaces of three types: GUIs specified in Java Swing as well as HTML and WML user interfaces.

To complement the set of user interface managers, the reference implementation includes a set of service components for handling of other miscellaneous functionality. The *IntraCom (Intra Communication) manager* let service components register a mailbox to which other service components can post messages. This allows spontaneous communication between service components that are new to each other. The *Preference manager* offers rudimentary handling of preference entries (key and value pairs) of the user. Service components can store and fetch entries, as well as subscribe to changes in the preference database. The user can inspect the database, and control which services should be allowed access to which entries. The Preference manager stores its database of preference entries as mobile properties.

6. Conclusions

We have described the overall architecture and the basic design and implementation of the *s*View system. In general, the design is motivated by the two requirements openness and user control. In particular, demands on heterogeneity and extendibility have influenced the design.

In order to allow extensions to the system it is separated into two parts: one core specification that provides APIs to main components of the system, and one reference implementation that provides developers of *s*View components and server functionality with a development and runtime environment.

The core specification builds roughly on three main components: a service component, a service briefcase, and a service briefcase server. In combination these three components provides developers, service providers, and end-users of electronic services with an open and extensible service infrastructure that allows far-reaching user control.

7. Acknowledgements

The design and implementation described in this report builds upon the author's experiences from participating in the development of similar systems [11-14].

The work presented in this report has been funded by The Swedish Institute for Information Technology (www.siti.se). Thanks to the members of the HUMLE

laboratory at the Swedish Institute of Computer Science (www.sics.se/humle), in particular Fredrik Espinoza, for inspiration and thoughtful comments. Special thanks to Mikael Boman and Anna Sandin for help with the implementation of *s*View.

References

[1] M. Bylund and A. Waern, "Personal Service Environments – Openness and User Control in User-Service Interaction," Swedish Institute of Computer Science, Kista, Sweden, SICS Technical Report T2001:07, May, 2001.

[2] "OSGi Service Gateway Specification Release 1.0," Open Services Gateway Initiative, May, 2000.

[3] H. L. Chen, "Developing a Dynamic Distributed Intelligent Agent Framework Based on the Jini Architecture," M.Sc. thesis, University of Maryland Baltimore County, Baltimore, 2000.

[4] N. Minar, M. Gray, O. Roup, R. Krikorian, and P. Maes, "Hive: Distributed Agents for Networking Things," presented at First International Symposium on Agent Systems and Applications, Third International Symposium on Mobile Agents featuring the Third Dartmouth Workshop on Transportable Agents, Rancho Las Palmas Marriott's Resort and Spa, Palm Springs, CA, 1999.

[5] C. Pullela, L. Xu, D. Chakraborty, and A. Joshi, "A Component Based Architecture for Mobile Information Access," Department of Computing Science and Electrical Engineering, University of Maryland Baltimore County, Technical Report, TR-CS-00-05, March 31, 2000.

[6] "JAR File Specification," Sun Microsystems, Inc., available at: http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html [2001, April 18], 1999.

[7] R. Rivest. "RFC1321: The MD5 Message Digest Algorithm," MIT and RSA Data Security, Inc., available at: http://www.faqs.org/rfcs/rfc1321.html [2001, April 17], 1992.

[8] "Java Remote Method Invokation Specification," Sun Microsystems, Inc., available at: http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmiTOC.html [2001, April 18], 1999.

[9] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple Object Access Protocol (SOAP) 1.1," World Wide Web Consortium, W3C Note 27 July 1999, May 8, 2000.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. "RFC2616: Hypertext Transfer Protocol - HTTP/1.1," World Wide Web Consortium, available at: http://www.w3c.org/Protocols/rfc2616/rfc2616.txt [2001, April 17], 1999.

[11] P. Charlton, Y. Chen, F. Espinoza, A. Mamdani, O. Olsson, J. Pitt, F. Somers, and A. Waern, "An Open Agent Architecture Supporting Multimedia Services on Public Information Kiosks," presented at Practical Applications of Intelligent Agents and Multi-Agent Systems, PAAM'97, London, UK, 1997.

[12] F. Espinoza, "sicsDAIS: Managing User Interaction with Multiple Agents," Ph.Lic. thesis, The Royal Institute of Technology and Stockholm University, Stockholm, 1998.

[13] F. Espinoza, "sicsDAIS: A Multi-Agent Interaction System for the Internet," presented at WebNet 99—World Conference on the WWW and Internet, Hawaii, 1999.

[14] M. Tierney, “ConCall: An Exercise in Designing Open Service Architectures,” Ph.Lic. thesis, The Royal Institute of Technology and Stockholm University, Stockholm, Sweden, 2000.

Appendix I

This appendix has been removed from the reprint of this technical report. Please see Appendix I of SICS Technical Report T2001:06 (available at http://www.sics.se), or Appendix I of this thesis for a shortened documentation of the core *s*View specification.

Appendix I

Contents

1 Package se.sics.sview.core **2**
1.1 Interfaces . . . 4
1.1.1 INTERFACE **Constants** . . . 4
1.1.2 INTERFACE **Mobile** . . . 6
1.1.3 INTERFACE **Persistent** . . . 7
1.1.4 INTERFACE **ServerProxy** . . . 7
1.1.5 INTERFACE **ServiceBriefcaseServer** . . . 8
1.1.6 INTERFACE **ServiceComponent** . . . 9
1.1.7 INTERFACE **ServiceComponentPermission** . . . 11
1.1.8 INTERFACE **ServiceContext** . . . 11
1.1.9 INTERFACE **ServiceInterfaceFactory** . . . 14
1.1.10 INTERFACE **ServiceListener** . . . 15
1.1.11 INTERFACE **ServiceProxy** . . . 15
1.1.12 INTERFACE **TransactionCoordinator** . . . 16
1.1.13 INTERFACE **TransactionInitiator** . . . 16
1.1.14 INTERFACE **TransactionParticipant** . . . 17
1.2 Classes . . . 18
1.2.1 CLASS **Monitor** . . . 18
1.2.2 CLASS **ServiceBriefcase** . . . 19
1.2.3 CLASS **ServiceContainer** . . . 22

Chapter 1

Package se.sics.sview.core

Package Contents *Page*

Interfaces

Constants 4
A set of constants used by service briefcases, service contexts, servers, etc.
Mobile 6
A service component that implements this interface will be included in the service briefcase during synchronization with other servers.
Persistent 7
A service component that implements this interface will be offered to save its state in the service briefcase before service briefcase synchronization and save.
ServerProxy 7
This interface should be used to wrap implementations of remote service briefcase server communication and transaction initiator to transaction participant communication.
ServiceBriefcaseServer 8
This interfcase specifies an API to sView service briefcase servers.
ServiceComponent 9
Should be implemented by service components that wish to execute in an sView PSE.
ServiceComponentPermission 11
Superclass of all service component permissions.
ServiceContext 11
This interface specifies an API to the runtime environment of a service briefcase.
ServiceInterfaceFactory 14
Service components that wish to register services for other service components to use must implement this interface.
ServiceListener 15
The listener interface for receiving service events.
ServiceProxy 15
A service component that wish to subscribe to a service requests a service proxy (defined by this class) to the service from its service context.
TransactionCoordinator 16
A transaction wraps the steps in service briefcase synchronization in order to make it atomic, and to provides exception handling.
TransactionInitiator 16

A transaction wraps the steps in service briefcase synchronization in order to make it atomic, and to provides exception handling.

TransactionParticipant 17

A transaction wraps the steps in service briefcase synchronization in order to make it atomic, and to provides exception handling.

Classes

Monitor 18

Implements a 'one-to-many monitor' for exclusive access to critical sections.

ServiceBriefcase 19

Contains functionality for creating, loading and saving service components.

ServiceContainer 22

A ServiceContainer wraps a service component, a JAR cache, and information about the service.

1.1 Interfaces

1.1.1 Interface **Constants**

A set of constants used by service briefcases, service contexts, servers, etc. Implement this interface to get easy access to the constants.

Declaration

```
public interface Constants
```

Fields

- public static final String JAR_NAME
  The JAR manifest key to the symbolic name of the service component.
- public static final String JAR_ACTIVATOR
  The JAR manifest key to the activator of a service component. The value of this key should be the fully qualified class name of the class of the service component that implements the interface ServiceComponent .
- public static final String JAR_CLASSPATH
  The JAR manifest key to the JAR-internal classpath that should be used when loading the service component. The value of this key should be a comma separated list of JAR entries (being themselves JAR files) or '.' (which stands for the classes in the root JAR file). List entries are searched for class definitions in order of appearance.
- public static final String JAR_EXPORT
  The JAR manifest key to the list of classes that this service component exports to other components. The value of this key should be a comma separated list of package names or fully qualified class names.
- public static final String JAR_IMPORT
  Currently not used.
- public static final String JAR_DEPEND
  The JAR manifest key to the list of names of services (offered by other service components) that this service component depends upon. The value of this key should be a comma separated list of service names.
- public static final String JAR_PERMISSION
  The JAR manifest key to the list of permissions that grants the service component rights to functionality of the system. The value of this key should be a comma separated list of permission interfaces in package se.sics.sview.core.permission or fully qualified class names that implement (or extend) one or more of the permission interfaces in se.sics.sview.core.permission.
- public static final String MP_HOSTS
  The mobile property key to the list of servers that the service briefcase has visited. The value of this key is used when the service briefcase is synchronized with other servers.

- public static final int INACTIVE
  The service component is either newly created and not yet added, or recently removed from, a service environment. In this state the service component is not allowed to interact with either its service context or with other service components.

- public static final int INITIALIZING
  The service component automatically reaches this state when the service context calls the initialize method of the service component. This is done as a first step to add the component to the service environment. In this state, the service component is expected to perform initialization that is only done once during the lifetime of a service component. This is the first chance of the service component to interact with the service context, but interaction with other service components is not allowed yet. The service component signals that initialization is done either by having the initialize method return `INITIALIZED`, or, if initialization continues after returning from the initialize method, by calling setState(INITIALIZED) on the service context. In the latter case a negative number should be returned by the initialize method to signal that initialization is not finished.

- public static final int INITIALIZED
  The service component reaches this state when it has finished initialization.

- public static final int STARTING
  The service component automatically reaches this state when the service context calls the start method of the service component. In this state, the service component should perform tasks that should be done every time it is about to start. Interaction with the service context is allowed, but not with other service components. The service component signals that starting is done either by having the start method return `ACTIVE`, or, if starting continues after returning from the start method, by calling setState(ACTIVE) on the service context. In the latter case a negative number should be returned by the start method to signal that starting is not finished.

- public static final int ACTIVE
  The service component reaches this state when it has finished starting. This is the state where most of the lifecycle of a service component is spent. The service component is allowed to interact with both the service context and other service components from here.

- public static final int SUSPENDING
  The service component automatically reaches this state when the service context calls the suspend method of the service component. This is done as a first step to suspend the component. In this state, the service component is expected to unregister all services that it offers other service components, as well as unsubscribe to services of other service components. The service component is allowed to interact with the service context in this state. It is also allowed to interact with other service components, but only for the purpose of handling unsubscriptions and unregistrations. The service component signals that suspension is done either by having the suspend method return `SUSPENDED`, or, if suspension continues after returning from the suspend method, by calling setState(SUSPENDED) on the service context. In the latter case a negative number should be returned by the suspend method to signal that suspension is not finished.

- public static final int SUSPENDED
  The service component reaches this state when it has finished suspension. In this state the service component is not allowed to interact with either its service context or other service components. The service component can now be saved to persistent media or moved to another server.

- public static final int RESUMING
  The service component automatically reaches this state when the service context calls the resume method of the service component. This is done as a first step to resume the component after suspension. This state is comparable to the `INITIALIZING` state, with the exception that the state can occur more than once. The service component signals that resumption is done either by having the resume method return `RESUMED`, or, if resumption continues after returning from the resume method, by calling setState(RESUMED) on the service context. In the latter case a negative number should be returned by the resume method to signal that resumption is not finished.

- public static final int RESUMED
  The service component reaches this state when it has finished resumption.

- public static final int STOPPING
  The service component automatically reaches this state when the service context calls the stop method of the service component. This is done as a first step to stop the component. In this state, the service component is expected to unregister all services that it offers other service components, as well as unsubscribe to services of other service components. The service component is allowed to interact with the service context in this state. It is also allowed to interact with other service components, but only for the purpose of handling unsubscriptions and unregistrations. The service component signals that stopping is done either by having the stop method return `STOPPED`, or, if stopping continues after returning from the stop method, by calling setState(STOPPED) on the service context. In the latter case a negative number should be returned by the stop method to signal that stopping is not finished.

- public static final int STOPPED
  The service component reaches this state when it has finished stopping. In this state the service component has reached the end of its lifecycle. Only a reload of a previously saved copy or creating a new instance of the service component can bring the service component back to the service environment. In this state the service component is not allowed to interact with either its service context or other service components.

- public static final String stateNames
  An array of symbolic names of the states of the service component. The value of the state variables of this class work as index to its corresponding name.

1.1.2 Interface **Mobile**

A service component that implements this interface will be included in the service briefcase during synchronization with other servers.

Declaration

```
public interface Mobile
```

1.1.3 Interface **Persistent**

A service component that implements this interface will be offered to save its state in the service briefcase before service briefcase synchronization and save.

Declaration

public interface Persistent
implements java.io.Serializable

Methods

- *freeze*
  `public void freeze( )`

  The service briefcase calls the `freeze` method when it saves the service component. This occurs after the service component has reached state `SUSPENDED`, but before state `RESUMING` is reached. The `freeze` method should be used to prepare for serialization by optimizing or removing data structures. The service component could e.g. compact a hash table or empty a media cache for more efficient storage After returning from the `freeze` method all external references (such as references to the service context, file and socket handles etc.) must have been set to `null`.

- *thaw*
  `public void thaw( )`

  The service briefcase calls the thaw method when a saved version of the service component is loaded. This occurs after the `freeze` method has been called (possibly in a different VM and on a different host), but before state `RESUMING` is reached. The `thaw` method should be used to, if needed, recreate data structures that were removed or converted in the `freeze` method. It should also be used to re-associate references that were set to `null` in the `freeze` method or during serialization.

1.1.4 Interface **ServerProxy**

This interface should be used to wrap implementations of remote service briefcase server communication and transaction initiator to transaction participant communication.

Declaration

public interface ServerProxy

Methods

- *getProtocol*
  `public String` **getProtocol**`( )`

  Returns the protocol that this server proxy implements.

- *getServiceBriefcaseServerProxy*
  `public ServiceBriefcaseServer` **getServiceBriefcaseServerProxy**`( java.lang.String` **uri** `)`

  Creates a new proxy to a service briefcase server.

- *getTransactionParticipantProxy*
  `public TransactionParticipant` **getTransactionParticipantProxy**`( java.lang.String` **uri**`, java.lang.String` **id** `)`

  Creates a new proxy to a transaction participant.

- *initialize*
  `public void` **initialize**`( se.sics.sview.core.ServiceBriefcaseServer` **localServer**`, java.lang.String []` **args** `)`

  This method should be called by the service briefcase server before any calls to #getServiceBriefcaseServerProxy(java.lang.String) or java.lang.String) are made.

1.1.5 INTERFACE **ServiceBriefcaseServer**

This interfcase specifies an API to sView service briefcase servers. It specifies methods for exchanging service briefcases and starting and stopping PSEs.

DECLARATION

public interface ServiceBriefcaseServer

METHODS

- *getMobileProperties*
  `public Properties` **getMobileProperties**`( java.lang.String` **uid**`, java.lang.String` **pwd**`, java.lang.String` **transactionId** `)`

  Returns the mobile properties of a service briefcase

- *getServiceBriefcase*
  `public ServiceBriefcase` **getServiceBriefcase**`( java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Returns a service briefcase.

- *getServiceBriefcase*
  `public ServiceBriefcase` **getServiceBriefcase**`( java.lang.String` **uid**`, java.lang.String` **pwd**`, java.util.Date` **date** `)`

  Returns a backuped service briefcase. The will return the version that was the latest at the time specified by the parameter date.

- *getServiceBriefcaseState*
  `public Properties` **getServiceBriefcaseState**`( java.lang.String` **uid**`, java.lang.String` **pwd**`, java.lang.String` **transactionId** `)`

  Returns an array containing the keys of the service components

- *getServiceComponents*
  `public ServiceContainer` **getServiceComponents**`( java.lang.String` **uid**`, java.lang.String` **pwd**`, java.lang.String []` **keys**`, java.lang.String` **transactionId** `)`

  Returns an array of service components that corresponds to an array of service component keys.

- *newServiceBriefcase*
  `public void` **newServiceBriefcase**`( java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Creates a new service briefcase.

- *removeServiceBriefcase*
  `public void` **removeServiceBriefcase**`( java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Removes a service briefcase from this server.

- *startPse*
  `public void` **startPse**`( java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Starts a PSE.

- *stopPse*
  `public void` **stopPse**`( java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Stops a PSE.

- *updateServiceBriefcase*
  `public void` **updateServiceBriefcase**`( java.lang.String` **uid**`, java.lang.String` **pwd**`, se.sics.sview.core.ServiceContainer []` **serviceComponents**`, java.util.Properties` **mobileProperties**`, java.lang.String` **transactionId** `)`

  Updates a service briefcase of a remote service briefcase server with new service containers and properties.

1.1.6 INTERFACE ServiceComponent

Should be implemented by service components that wish to execute in an sView PSE. See ServiceContext for a description of the context in which the service component will execute.

DECLARATION

```
public interface ServiceComponent
```

Methods

- *initialize*

  `public int` **`initialize`**`( se.sics.sview.core.ServiceContext` **`context`**`, se.sics.sview.core.ServiceContextEvent` **`evt`** `)`

  Instructs the service component to initialize. The implementation of this method should execute fast. If initialization finish before the method terminates, it should return INITIALIZED . Otherwhise it should return a negative value to indicate that initialization is ongoing. In this case the service component must call setState with INITIALIZED when initialization is done to signal that the service component is ready to start.

- *resume*

  `public int` **`resume`**`( se.sics.sview.core.ServiceContext` **`context`**`, se.sics.sview.core.ServiceContextEvent` **`evt`** `)`

  Instructs the service component to resume. The implementation of this method should execute fast. If resumption finish before the method terminates, it should return RESUMED . Otherwhise it should return a negative value to indicate that resumption is ongoing. In this case the service component must call setState with RESUMED when resumption is done to signal that the service component is ready to start.

- *start*

  `public int` **`start`**`( se.sics.sview.core.ServiceContext` **`context`**`, se.sics.sview.core.ServiceContextEvent` **`evt`** `)`

  Instructs the service component to start. The implementation of this method should execute fast. If the service component is started before the method terminates, it should return ACTIVE . Otherwhise it should return a negative value to indicate that starting is ongoing. In this case the service component must call setState with ACTIVE when the service component is started to signal that the service component is active.

- *stop*

  `public int` **`stop`**`( se.sics.sview.core.ServiceContext` **`context`**`, se.sics.sview.core.ServiceContextEvent` **`evt`** `)`

  Instructs the service component to stop. The implementation of this method should execute fast. If the service component is stopped before the method terminates, it should return STOPPED . Otherwhise it should return a negative value to indicate that stopping is ongoing. In this case the service component must call setState with STOPPED when the service component is stopped to signal that the service component can be terminated.

- *suspend*

  `public int` **`suspend`**`( se.sics.sview.core.ServiceContext` **`context`**`, se.sics.sview.core.ServiceContextEvent` **`evt`** `)`

  Instructs the service component to suspend. The implementation of this method should execute fast. If suspension finish before the method terminates, it should return SUSPENDED . Otherwhise it should return a negative value to indicate that suspension is ongoing. In this case the service component must call setState with SUSPENDED when suspension is done to signal that the service component is suspended.

1.1.7 Interface **ServiceComponentPermission**

Superclass of all service component permissions. See the interfaces in package se.sics.sview.core.permission for a full listing of predefined permissions.

Declaration

```
public interface ServiceComponentPermission
```

Fields

- public static final String description
  A textual description of the permission. Override this field to describe what the permission grants access to.

1.1.8 Interface **ServiceContext**

This interface specifies an API to the runtime environment of a service briefcase. It specifies an methods for handling service components (creation, maintenance, removal, etc.), the runtime environment (save, synchronize, reload, and shutdown), and the state of the service component.

A service component has access to three types of properties via its service context: local, stationary, and mobile. *Local* properties are controlled by the administrator of the server on which the PSE executes. These properties can be read but not be set nor modified by service components. *Stationary* properties can both be read, set, and modified by service components. However, every service component has its own view of stationary properties which means that the one service component cannot reach the stationary properties of another. Stationary properties are local to a specific server. *Mobile* properties can both be read, set, and modified by service components. However, every service component has its own view of mobile properties which means that the one service component cannot reach the mobile properties of another. Mobile properties follow the PSE as it migrates from server to server.

Declaration

```
public interface ServiceContext
implements Constants
```

Methods

- *createServiceComponent*
  `public void` **createServiceComponent**`( java.lang.String` **jarName** `)`

  Creates and adds a service component based on a JAR file containing a specification of an service component.

- *getJarAttribute*
  `public String` **getJarAttribute**`( java.lang.String` **name** `)`

  Gets an attribute from the JAR file of the service component.

- *getJarEntry*
  `public byte` **getJarEntry**`( java.lang.String` **name** `)`

  Gets a JAR entry from the JAR file of the service component.

- *getLocalProperty*
  `public String` **getLocalProperty**`( java.lang.String` **key** `)`

  Searches for the property with the specified key in local property list. The method returns `null` if the property is not found.

- *getLocalProperty*
  `public String` **getLocalProperty**`( java.lang.String` **key**`, java.lang.String` **def** `)`

  Searches for the property with the specified key in the local property list. The method returns the default value argument if the property is not found.

- *getMobileProperty*
  `public String` **getMobileProperty**`( java.lang.String` **key** `)`

  Searches for the property with the specified key in the mobile property list. The method returns `null` if the property is not found.

- *getMobileProperty*
  `public String` **getMobileProperty**`( java.lang.String` **key**`, java.lang.String` **def** `)`

  Searches for the property with the specified key in the mobile property list. The method returns the default value argument if the property is not found.

- *getServiceProxy*
  `public ServiceProxy` **getServiceProxy**`( java.lang.String` **name** `)`

  Acquire a proxy to a service.

- *getState*
  `public int` **getState**`( )`

  Returns the state of the service.

- *getStationaryProperty*
  `public String` **getStationaryProperty**`( java.lang.String` **key** `)`

  Searches for the property with the specified key in the stationary property list. The method returns `null` if the property is not found.

- *getStationaryProperty*
  `public String` **getStationaryProperty**`( java.lang.String` **key**`, java.lang.String` **def** `)`

  Searches for the property with the specified key in the stationary property list. The method returns the default value argument if the property is not found.

- *loadServiceComponent*
  `public void` **loadServiceComponent**`( java.io.InputStream` **is** `)`

  Loads and adds a saved service component from an input stream.

- *registerService*
  `public void` **registerService**(`java.lang.String` **name**, `se.sics.sview.core.ServiceInterfaceFactory` **interfaceFactory**)

  Registers a service.

- *reload*
  `public void` **reload**( )

  Resets the PSE to the last saved state. This method will cause the PSE to shutdown temorarily. Unsaved data and modifications will be lost.

- *remove*
  `public void` **remove**( )

  Schedules the service component for removal.

- *removeServiceComponent*
  `public void` **removeServiceComponent**(`java.lang.String` **key**)

  Removes a service component from the PSE.

- *resumeServiceComponent*
  `public void` **resumeServiceComponent**(`java.lang.String` **key**, `se.sics.sview.core.ServiceContextEvent` **evt**)

  Resumes a service component.

- *save*
  `public void` **save**( )

  Saves the state of the PSE in a service briefcase. This method will cause the PSE to shutdown temporarily.

- *setMobileProperty*
  `public void` **setMobileProperty**(`java.lang.String` **key**, `java.lang.String` **value**)

  Sets the property with the specified key in the mobile property list.

- *setState*
  `public void` **setState**(`int` **state**)

  Sets the state of the service. This method is only effective if the service is currnetly engaged in a state change (i.e. the ServiceContext has called one of the state modifying methods, to which the service has returned a negative value to indicate that the state modification is ongoing).

- *setStationaryProperty*
  `public void` **setStationaryProperty**(`java.lang.String` **key**, `java.lang.String` **value**)

  Sets the property with the specified key in the stationary property list.

- *shutdown*
  `public void` **shutdown**( )

  Performs a shutdown of the PSE without saving. Unsaved data and modifications will be lost.

- *stop*
  `public void` **stop**`( )`

  Schedules the service component for termination.

- *stopServiceComponent*
  `public void` **stopServiceComponent**`( java.lang.String` **key**`,`
  `se.sics.sview.core.ServiceContextEvent` **evt** `)`

  Stops a service component.

- *suspend*
  `public void` **suspend**`( )`

  Schedules the service component for suspension.

- *suspendServiceComponent*
  `public void` **suspendServiceComponent**`( java.lang.String` **key**`,`
  `se.sics.sview.core.ServiceContextEvent` **evt** `)`

  Suspends a service component.

- *synchronize*
  `public void` **synchronize**`( )`

  Synchronizes the PSE with the default service briefcase server. This method will cause the PSE to shutdown temporarily.

- *unregisterService*
  `public void` **unregisterService**`( java.lang.String` **name** `)`

  Unregisters a service.

1.1.9 Interface **ServiceInterfaceFactory**

Service components that wish to register services for other service components to use must implement this interface. An instantiation of the implementation should be sent to the service context during service registration, and is used to create interfaces to the service when other service components requests subscriptions.

Declaration

public interface ServiceInterfaceFactory

Methods

- *createServiceInterface*
  `public Object` **createServiceInterface**`( java.lang.String` **key** `)`

  Invoked to create a service interface to a service of a service component.

1.1.10 Interface **ServiceListener**

The listener interface for receiving service events. The class that is interested in processing a service event implements this interface. An instantiation of the implementation is sent to the service proxy of the service of interest by calling the addServiceListener method. When the service event occurs, the serviceRegistered/serviceUnregistered method of the implementation are invoked.

Declaration

```
public interface ServiceListener
```

Methods

- *serviceRegistered*
  `public void` **serviceRegistered**`( java.lang.String` **name** `)`

  Invoked when the service registers.

- *serviceUnregistered*
  `public void` **serviceUnregistered**`( java.lang.String` **name** `)`

  Invoked when the service unregisters.

1.1.11 Interface **ServiceProxy**

A service component that wish to subscribe to a service requests a service proxy (defined by this class) to the service from its service context. Via the service proxy, the service component can (un)subscribe to the service, and register for notifications of when the service (un)registers (a service need not be registered in order for a service component to acquire a service proxy to it).

Declaration

```
public interface ServiceProxy
```

Methods

- *addServiceListener*
  `public void` **addServiceListener**`( se.sics.sview.core.ServiceListener` **listener** `)`

  Adds a service listener to this service proxy. The listener will be notified when the service of this service proxy (un)registers.

- *removeServiceListener*
  `public void` **removeServiceListener**`( se.sics.sview.core.ServiceListener` **listener** `)`

  Removes a service listener from this service proxy.

- *subscribe*
  `public Object` **subscribe**`( )`

  Register a subscription to this service.

- *subscribe*
  `public Object` **subscribe**`( long` **timeout** `)`

  Subscribe to this service. If the service is not registered yet, wait `timeout` milliseconds for it to registered. If the service is not registered within that time, return `null`.

- *unsubscribe*
  `public void` **unsubscribe**`( )`

  Unregister a subscription to this service.

1.1.12 Interface **TransactionCoordinator**

A transaction wraps the steps in service briefcase synchronization in order to make it atomic, and to provides exception handling.

A transaction coordinator shold implement this interface. The coordinator of a transaction can be, but need not be, the initiator if the synchronization.

Declaration

public interface TransactionCoordinator

Methods

- *abort*
  `public void` **abort**`( )`

  Aborts the transaction.

- *acknowledge*
  `public void` **acknowledge**`( se.sics.sview.core.TransactionParticipant` **tp** `)`

  The participants of a transaction calls this method in order to acknowledge that the transaction has completed successfully.

1.1.13 Interface **TransactionInitiator**

A transaction wraps the steps in service briefcase synchronization in order to make it atomic, and to provides exception handling.

A transaction initiator shold implement this interface.

Declaration

---

public interface TransactionInitiator

Methods

- *globalAcknowledge*
  `public void` **globalAcknowledge**`( )`

  The coordinator of the transaction calls this method when all participants have acknowledge that the transaction has completed successfully.

- *globalCommit*
  `public void` **globalCommit**`( se.sics.sview.core.TransactionParticipant [] ` **tps** `)`

  The coordinator of the transaction calls this method when all participants have voted for participation. The participants that voted in favor of the transaction are representd in the given array of participants.

1.1.14 Interface **TransactionParticipant**

A transaction wraps the steps in service briefcase synchronization in order to make it atomic, and to provides exception handling.

A transaction participant shold implement this interface.

Declaration

---

public interface TransactionParticipant

Methods

- *globalCommit*
  `public void` **globalCommit**`( se.sics.sview.core.TransactionCoordinator ` **tc** `)`

  Called by the coordinator to signal that the transaction is ready to start.

- *vote*
  `public boolean` **vote**`( )`

  Called by the coordinator to vote for participation in a transaction.

1.2 Classes

1.2.1 Class **Monitor**

Implements a 'one-to-many monitor' for exclusive access to critical sections. When no one has arrogated ownership of the monitor, everyone are free to enter and exit at will. Simultaneous consumers are not synchronized (except during the very brief call to method `enter`).

A call to `arrogate` will claim ownership of the monitor, and with that exclusive access to sections that are guarded by this monitor. Method `renounce` release ownership of the monitor.

For example, protect a code section with:

```
...

enter();

// perform protected actions

exit();

...
```

and claim ownership with:

```
...

Object monitorReference = new Object();

synchronized(monitorReference);

arrogate(monitorReference);

// perform actions that require exclusive ownership

renounce();

}

...
```

Declaration

public class Monitor
extends java.lang.Object

CONSTRUCTORS

- *Monitor*
  `public` **Monitor**`( )`

METHODS

- *arrogate*
  `public synchronized void` **arrogate**`( java.lang.Object` **ref** `)`

  Arrogate exclusive access to the monitor. This method will block until exclusive ownership can be realized. This requires two conditions to be met: all consumers must leave the monitor and the monitor cannot be arrogated by someone else.

- *enter*
  `public synchronized void` **enter**`( )`

  Enter monitor. If monitor is locked, this method will block.

- *exit*
  `public synchronized void` **exit**`( )`

  Exit monitor.

- *renounce*
  `public synchronized void` **renounce**`( )`

  Renounce exclusive access to the monitor.

1.2.2 CLASS **ServiceBriefcase**

Contains functionality for creating, loading and saving service components. It also provides storage of the JAR files of service components, persistent service components, and properties.

The service briefcase is serializable and it can be stored on persistent media and sent between servers, or have its content synchronized with service briefcases on other servers.

Much of the functionality of the service briefcase is delegated to service containers ServiceContainer , of which there is one for each service component in the briefcase. The service container provides storage and serialization handling of individual service components. It includes functionality for creating, loading, and saving service components, storing persistent service components, and caching the JAR file of service components.

Service component creation and loading requires that a class loader is provided by the server implementation. The server typically uses separate class loaders for every service component in the system. This ensures that no service component should be able to manipulate other service components without permission.

Declaration

public class ServiceBriefcase
extends java.lang.Object
implements java.io.Serializable

Constructors

- *ServiceBriefcase*
  `public` **ServiceBriefcase**`( java.util.Properties` **mobileProps**`, java.util.Properties` **stationaryProps**`, java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Creates a new service briefcase with predefined mobile and stationary properties.

- *ServiceBriefcase*
  `public` **ServiceBriefcase**`( java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Creates a new empty service briefcase.

Methods

- *changePassword*
  `public final void` **changePassword**`( java.lang.String` **uid**`, java.lang.String` **oldPwd**`, java.lang.String` **newPwd** `)`

  Changes the password of this briefcase.

- *getMobileProperties*
  `public Properties` **getMobileProperties**`( java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Returns the mobile properties of this briefcase.

- *getServiceComponents*
  `public ServiceContainer` **getServiceComponents**`( java.lang.String [] ` **keys**`, java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Returns the service components that corresponds to the given set of keys.

- *getServiceContainer*
  `public ServiceContainer` **getServiceContainer**`( java.lang.String` **key**`, java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Returns the service container that corresponds to the given key.

- *getServiceKeys*
  `public synchronized String` **getServiceKeys**`( java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Returns an array that contains the keys of the service containers in this briefcase.

- *getState*
  `public Properties` **getState**`( java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Returns the current state of this briefcase.

- *getStationaryProperties*
  `public Properties` **getStationaryProperties**`( java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Returns the stationary properties of this briefcase.

- *load*
  `public static ServiceBriefcase` **load**`( java.io.InputStream` **is** `)`

  Loads a serialized service briefcase from a given input stream.

  NOTE! Service briefcases, in order to be loaded properly, must be loaded with this method.

- *lock*
  `public void` **lock**`( java.lang.Object` **ref** `)`

  Locks this briefcase (see Monitor).

- *putServiceContainer*
  `public synchronized void` **putServiceContainer**`( se.sics.sview.core.ServiceContainer` **service**`, java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Adds/overwrites a service container.

- *removeServiceContainer*
  `public synchronized void` **removeServiceContainer**`( java.lang.String` **key**`, java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Removes the service container that corresponds to the given key.

- *save*
  `public static void` **save**`( se.sics.sview.core.ServiceBriefcase` **sb**`, java.io.OutputStream` **os** `)`

  Saves service briefcase to a given output stream.

  NOTE! Service briefcases, in order to be saved properly, must be saved with this method.

- *setMobileProperties*
  `public void` **setMobileProperties**`( java.util.Properties` **props**`, java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Sets the mobile properties of this briefcase.

- *setMonitor*
  `public synchronized void` **setMonitor**`( se.sics.sview.core.Monitor` **monitor** `)`

  Sets the monitor for this briefcase (see Monitor).

- *setStationaryProperties*
  `public void` **setStationaryProperties**`( java.util.Properties` **props**`, java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Sets the stationary properties of this briefcase.

- *toMobile*
  `public ServiceBriefcase` **toMobile**`( java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Creates a new service briefcase with all mobile properties and service components of this briefcase.

- *unlock*
  `public void` **unlock**`( )`

  Unlocks this briefcase (see Monitor).

- *updateServiceBriefcase*
  `public void` **updateServiceBriefcase**`( se.sics.sview.core.ServiceContainer []` **serviceContainers**`, java.util.Properties` **mobileProperties**`, java.lang.String` **uid**`, java.lang.String` **pwd** `)`

  Updates this briefcase with a new set of service components and mobile properties.

1.2.3 Class ServiceContainer

A ServiceContainer wraps a service component, a JAR cache, and information about the service.

Declaration

public class ServiceContainer
extends java.lang.Object
implements java.io.Serializable, java.lang.Cloneable

Constructors

- *ServiceContainer*
  `public` **ServiceContainer**`( java.lang.String` **jarUrl**`, java.lang.String` **key** `)`

  Loads a service component specification and creates a new container for a service component with the given key. This method does not *create* the service component per se.

Methods

- *createServiceComponent*
  `public ServiceComponent` **createServiceComponent**`( java.lang.ClassLoader` **loader** `)`

  Creates a new service component based on the currently cached specification (the JAR file).

- *getCacheDate*
  `public Date` **getCacheDate**`( )`

  Returns the date of the current version of the JAR file.

- *getChangeDate*
  `public Date` **getChangeDate**`( )`

  Returns the date of the latest change of the service component in this container.

- *getCreationDate*
  `public Date` **getCreationDate**`( )`

  Returns the creation date of the service component in this container.

- *getJar*
  `public byte` **getJar**`( )`

  Returns the specification of the service component as a byte array.

- *getJarStream*
  `public InputStream` **getJarStream**`( )`

  Returns the specification of the service component in an output stream.

- *getJarUrl*
  `public String` **getJarUrl**`( )`

  Returns the URL of the original specification of the service component in this container (the JAR file).

- *getKey*
  `public String` **getKey**`( )`

  Returns the key of the service component.

- *getServiceComponent*
  `public byte` **getServiceComponent**`( )`

  Returns the service component of this container.

- *isMobile*
  `public boolean` **isMobile**`( )`

  Returns `true` if the service component has been declared as mobile (see Mobile).

- *isPersistent*
  `public boolean` **isPersistent**`( )`

  Returns `true` if the service component has been declared as persistent (see Persistent).

- *load*
  `public static ServiceContainer` **load**`( java.io.InputStream` **is**`,`
  `java.lang.String` **key** `)`

  Loads a serialized service container from a given input stream.

  NOTE! Service containers, in order to be loaded properly, must be loaded with this method.

- *loadServiceComponent*
  `public ServiceComponent` **loadServiceComponent**`( java.lang.ClassLoader` **loader**
  `)`

  Loads a previously saved copy of the service component in this container.

- *merge*
  `public void` **merge**`( se.sics.sview.core.ServiceContainer` **sc** `)`

  Merges the content of the given service container to this service container.

- *removeJar*
  `public void` **removeJar**`( )`

  Empties the JAR cache.

- *save*
  `public static void` **save**`( se.sics.sview.core.ServiceContainer` **sc**`, java.io.OutputStream` **os** `)`

  Saves a serialized service container to a given output stream.

  NOTE! Service containers, in order to be saved properly, must be saved with this method.

- *saveServiceComponent*
  `public void` **saveServiceComponent**`( se.sics.sview.core.ServiceComponent` **s** `)`

  Saves the service component of this container.

- *setChangeDate*
  `protected void` **setChangeDate**`( java.util.Date` **changeDate** `)`

- *setCreationDate*
  `protected void` **setCreationDate**`( java.util.Date` **creationDate** `)`

- *setMonitor*
  `public void` **setMonitor**`( se.sics.sview.core.Monitor` **monitor** `)`

  Sets the monitor of this service container (see Monitor).

- *setServiceComponent*
  `public void` **setServiceComponent**`( byte []` **serviceComponent** `)`

  Sets the service component of this container.

- *stripJar*
  `public ServiceContainer` **stripJar**`( )`

  Returns a clone of this container, without the jar.

- *stripServiceComponent*
  `public ServiceContainer` **stripServiceComponent**`( )`

  Removes a clone of this container, without the service component.

- *toString*
  `public String` **toString**`( )`

  Returns a string representation of this container.

- *validateJar*
  `public void` **validateJar**`( )`

  Validates the JAR file of this service component. If it turns out that it is old, it will be updated.

Licentiate theses from the Department of Information Technology

2000-001 Katarina Boman: *Low-Angle Estimation: Models, Methods and Bounds*

2000-002 Susanne Remle: *Modeling and Parameter Estimation of the Diffusion Equation*

2000-003 Fredrik Larsson: *Efficient Implementation of Model-Checkers for Networks of Timed Automata*

2000-004 Anders Wall: *A Formal Approach to Analysis of Software Architectures for Real-Time Systems*

2000-005 Fredrik Edelvik: *Finite Volume Solvers for the Maxwell Equations in Time Domain*

2000-006 Gustaf Naeser: *A Flexible Framework for Detection of Feature Interactions in Telecommunication Systems*

2000-007 Magnus Larsson: *Applying Configuration Management Techniques to Component-Based Systems*

2000-008 Marcus Nilsson: *Regular Model Checking*

2000-009 Jan Nyström: *A formalisation of the ITU-T Intelligent Network standard*

2000-010 Markus Lindgren: *Measurement and Simulation Based Techniques for Real-Time Analysis*

2000-011 Bharath Bhikkaji: *Model Reduction for Diffusion Systems*

2001-001 Erik Borälv: *Design and Usability in Telemedicine*

2001-002 Johan Steensland: *Domain-based partitioning for parallel SAMR applications*

2001-003 Erik K. Larsson: *On Identification of Continuous-Time Systems and Irregular Sampling*

2001-004 Bengt Eliasson: *Numerical Simulation of Kinetic Effects in Ionospheric Plasma*

2001-005 Per Carlsson: *Market and Resource Allocation Algorithms with Application to Energy Control*

2001-006 Bengt Göransson: *Usability Design: A Framework for Designing Usable Interactive Systems in Practice*

2001-007 Hans Norlander: *Parameterization of State Feedback Gains for Pole Assignment*

2001-008 Markus Bylund: *Personal Service Environments – Openness and User Control in User-Service Interaction*

UPPSALA
UNIVERSITY