Adaptation to Connectivity Loss in Pervasive Computing Environments

Manel Fredj Nikolaos Georgantas Valérie Issarny
INRIA-Rocquencourt, France
{Manel.Fredj, Nikolaos.Georgantas, Valerie.Issarny}@inria.fr

Apostolos Zarras
Dept. of Computer Science, University of Ioannina, Greece
zarras@cs.uoi.gr

Abstract

Pervasive computing environments aim at providing users with advanced services, dynamically composed our of networked services. In these open environments, availability of specific networked service instances cannot be guaranteed over time as users move and services leave and join the network accordingly. A major challenge in pervasive environments is thus to maintain services functionalities despite the dynamics of the environment, which induces connectivity loss with service instances. In this paper, we analyse the requirements to make distributed composite services able to face connectivity loss, i.e., able to dynamically adapt their configuration according to the networking environment. We then discuss the adaptation of relevant techniques that originate in the fault tolerance domain to the specifics of pervasive computing.

1 . Introduction

Pervasive computing systems are among today's most attractive vision of the future of distributed computing systems. Computational power will be available everywhere. Mobile and stationary devices will dynamically connect and coordinate to seamlessly help people in accomplishing their tasks. For this vision to become a reality, services must adapt themselves according to the constantly changing conditions that characterize the environment within which they operate.

In this paper, we consider a pervasive computing environment as an open space populated of mobile and possibly stationary devices (e.g., PDAs, laptops, smartphones, PCs, etc) hosting services and communicating through the use of an IP network. The network that connects the devices may be a pure MANet (Mobile Ad-Hoc Network) or infrastructure-based. The main characteristics of mobile devices is that they arbitrarily join and leave the environment's network as their owner changes their location.

In our open, dynamic pervasive environment, devices expose networked services according to the Service-Oriented Architecture (SOA) paradigm [9]. *Services* are self-describing. They are computational elements that support rapid, low-cost composition of distributed applications. Services perform functions, which can be anything varying from simple requests to complicated collaboration. Services are offered by service providers that procure the service implementations, supply service descriptions, and provide related conversation schema to service requester. Since services may be offered by different devices available in the environment and communicate through the network, they provide a distributed computing infrastructure for application composition and collaboration. Services can be simple or composite. A *simple service* has a pre-defined functionality. The requester interacts with a simple service without a third part intervention. *Composite services* involve assembling existing – simple or composite – services that access and combine information and functions from possibly multiple service providers.

Service composition consists in selecting the services to be composed, to decide how they are going to coordinate with each other (i.e., sequential, parallel, transactional, etc). Services are composed dynamically towards realizing advanced functionalities. There are two essential models for service composition are: (i) *service orchestration*, where a customer invokes a set of services in a coordinated way, and (ii) *service choreography*, where a set of services interact with each other in a peer-to-peer fashion. In this paper, we focus more specifically on service orchestration.

In the pervasive computing environment, the orchestration may loose connectivity with some component service(s) due to, e.g., mobility of service hosts. Our goal is to guarantee *continuity of composite services* despite connectivity loss, including maintaining consistency of composite and component services. In this context, relevant solutions to connectivity loss were introduced in the nomadic computing domain [6], where the connection between service provider and requester is intermittent. However, unlike nomadic computing systems, pervasive computing systems cannot assume/expect eventual reconnection of the disconnected service. Specifically, nomadic computing systems enable users to be mobile and to carry around wireless devices. The key concept in such systems is that a service requester is connected to some given remote server providing required services. Connectivity between the service requester and provider may be intermittent due to insufficient wireless network coverage or limited bandwidth shared between multiple users. However, it is assumed that the requester, eventually will reconnect to the same server or some replica of the server. Then, the objective is to enable users to use their mobile devices even during periods of low or non-connectivity. The distinctive feature that differentiates pervasive systems is that service instances making up the environment do not have any *a priori* knowledge of each other before their dynamic composition. Bindings between services are ad hoc and temporary. Thus, after a disconnection, a service requester is unlikely to reconnect to the same service provider or even a replica of it. It will rather connect to another provider. Nevertheless, disconnection is a common event in nomadic computing systems, which makes the experience gained from the nomadic system solutions useful.

This paper adresses the dynamics of pervasive computing environments, focusing on dynamic discovery and composition of networked services (§ 2). In order to maintain service composition despite mobility of services hosts and thus connectivity loss, we analyse different techniques of dynamic adaptation, originating from the fault tolerance domain, and discuss their adaptation to the requirements of pervasive computing environments (§ 3). Finally, we conclude with a summary of tour contribution, and point out open issues and future work (§ 4).

2 . Service Discovery and Composition in Pervasive Computing Environments

To illustrate in more details the connectivity loss problem in SOA-based environments, we employ a motivating scenario inspired by [10]. Based on this scenario, we introduce a service-oriented pervasive environment supporting dynamic service composition.

2.1 . An Illustrative Scenario

In our scenario, we are placed in the near territory of an island where Richard lives. Richard takes the ferry boat every day, to go from the island to his work place on the mainland and to return back home after work. Our pervasive environment consists of several services offering tourist information, hotel reservation, car reservation, etc. These services execute on stationary hosts located onshore. The environment that we consider, further comprises mobile hosts located on ferries, yachts, etc. At a short distance from the mainland, services residing on mobile hosts may have access to the services located onshore through a wireless network. If moving further from the mainland, however, their only possibility to access the services is through satellite-based connections, which are usually expensive and inefficient (especially in the case of GEO networks). To confront this problem, the mainland's local authorities realized the following setup. The stationary hosts located onshore may actually recruit volunteer mobile hosts that can serve as their proxies. Proxies provide indirect wireless access to the onshore services to mobile entities that do not have direct access to these services. As an exchange, the crew members and the travellers onboard may take benefit from more favourable transport, hotel, restaurant and car rental prices.

Richard targets to plan his holidays while travelling. He has a specification of a ”holiday planner” service on his PDA. The service composes three simple services: *airline tickets booking*, *car rental* and *hotel booking* services. The composite service orchestration is expressed as follow. The airline tickets booking service gives the list of available flights from which Richard can choose a flight, the corresponding airport and the departure and arrival dates. Afterwards, the car rental service provides a list of companies near/in the chosen airport from which Richard selects the car and the price that suit him. Finally, according to whether Richard has chosen to rent a car or not, the hotel booking service provides a list of hotels in the destination city that favour public transport availability or touristic areas or both of them if possible.

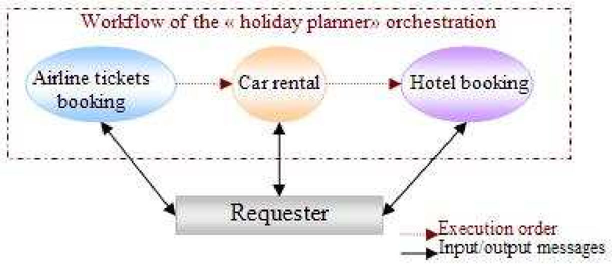


Figure 1. Composite service specification in terms of simple services

The ”holiday planner” service is specified at Richard's device as an orchestration of three simple services (Figure 1). When Richard invokes the ”holiday planner” service, a service composition process is triggered, which consists of: performing Service Discovery (*SD*) and then running the Service Configurator (*SC*) that actually composes/integrates the discovered services to match the orchestration specification. During the execution of the composite service, the Reconfiguration Manager (*RM*) supports the adaptation of the composition to potential connectivity losses with the services participating in the orchestration, due to the inherent mobility of mobile service providers and consumers.

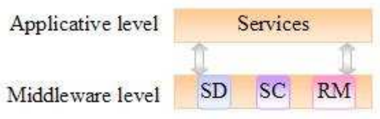


Figure 2. Adaptation support level

As depicted in Figure 2, the SD, SC and RM components are implemented at the middleware level and are deployed on each service host taking part in the orchestration.

2.2 . Service Discovery and Composition

As illustrated above, in the open pervasive computing environment, it is important that users can discover new services on-the-fly and use them as means to share information/computation with other users. In our scenario, Richard should be able to execute the ”holiday planner” service according to the services available in his environment. Consequently, the service discovery process initiated on his device should locate services that are suitable for the realization of the ”holiday planner” orchestration from Richard's specification. At the time when Richard joins a pervasive environment, Richard's SD obtains information about services provided by his neighbours – mobile and stationary – devices (i.e., devices in the network(s) to which he has access). More specifically, building on the WSAMI middleware [5] enabling mobile Web services for pervasive computing environments, SD periodically checks the environment for other instances of SD services hosted by neighbour devices. This task is realized by multicasting a discovery request using a standard discovery protocol (e.g., SLP [14]). Then, SD provides an operation for syntactic service search. Building upon the Web service architecture, the syntactic search takes as input the WSDL [13] interface specification of a required service *ws*. When invoked, SD makes corresponding calls to the neighbour SD services. The replies of all neighbours concerning provided services that syntactically match *ws* are merged into a single set *RESws*, which is returned back to requesting SD. Caching the most recent replies further enables optimising service discovery latency and bandwidth consumption. We assume that Richard's SD discovers in the environment an ”Airline tickets booking” service and a ”Car rental & hotel booking” service. This leads the service configurator, *SC*, to instantiate the ”holiday planner” composition as depicted in Figure 3. Afterwards, the ”holiday planner” service instance starts its orchestration.

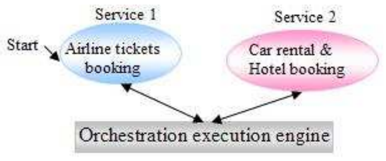


Figure 3. Composite service deployment in the environment

We are conscious that syntactic discovery of services has some limits in open pervasive computing environments; supporting semantic description and matching of services is an essential requirement [2, 8]. Indeed, enforcing an agreement on a common syntax for denoting common semantics is impossible to achieve in open environments, such as pervasive computing environments. Thus, the latest tendency is towards adopting semantic representation paradigms for specifying and matching services even when these differ in their syntactic interfaces. Such paradigms employ ontologies to represent concepts and related well-founded formalisms to enable machine reasoning about them, which we plan to adopt in our future work.

While Richard interacts with the ”Car rental & hotel booking” service, the device *D* on which the service is deployed is switched off. The service is no longer available. Richard's RM have to deal with this situation to let Richard fulfilling his ”holiday planning” despite the disconnection, whilst *D*'s RM has to restore the ”Car rental & hotel booking” service to a consistent state. For instance, *D*'s RM have to deal with the problem of validating or not the result of the interaction with Richard, e.g., if Richard has validated his booking for a flight, *D*'s RM has to take into consideration Richard's validation.

In fact, in an open, dynamic, ad hoc environment, services are not aware of their future collaborations. Thus, reconfiguration cannot rely on a central manager that would supervise all service collaborations and perform all adaptations needed by the services in the environment. Indeed, the central manager can be a bottleneck as the number of collaborations increases. Furthermore, the central manager would have to be informed of each new collaboration as soon as it is initiated, which is not feasible in an open environment such as the one we consider.

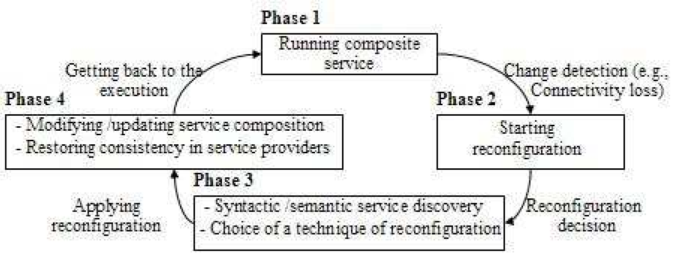
3 Adaptation to Pervasive Computing Environment

In order to introduce the reconfiguration techniques that meet the specifics of pervasive computing environments, we define a generic reconfiguration process, which we call *reconfiguration cycle*. This cycle introduces the general phases of reconfiguration starting from change occurrence and ending by returning back to execution after performing reconfiguration.

3.1 . Reconfiguration Cycle

We call *reconfiguration cycle* the sequence of phases that take place during the execution of the composite service, reconfiguring the service and taking it from a consistent state to another. We introduce the reconfiguration cycle, depicted in Figure 4, where boxes denote reconfiguration phases, and arrows denote events that take the system from one reconfiguration phase to another.

In Phase 1, the composite service executes normally while RM monitors the service's execution, including its conversations with the orchestrated services, i.e., at which point the execution of the composite service is and which services are in interaction with the requester. In Phase 2, a cause for reconfiguration emerges, generated by either the composite service or its environment. An example of the



Phase 1
Running composite service
Change detection (e.g., Connectivity loss)
Phase 2
Starting reconfiguration
Reconfiguration decision
Phase 3
- Syntactic /semantic service discovery
- Choice of a technique of reconfiguration
Applying reconfiguration
Phase 4
- Modifying /updating service composition
- Restoring consistency in service providers
Getting back to the execution

Figure 4. Reconfiguration cycle

former case may be the disconnection or failure of a composed service, or drop in the available bandwidth, while an example of the latter case may be the availability of a new service that offers enhanced quality of service. This phase triggers a reconfiguration. In Phase 3, RM decides to reconfigure the composed service according to the execution stage and the role/importance of disconnected service(s). In order to accomplish this aim, RM triggers the most suited technique to face the problem. Reconfiguration techniques are predefined means to face occurrence of connectivity loss. For instance, depending on the case of stateful or stateless disconnected service and the service state availability, RM can invoke a state transfer technique or not. In the next section, we investigate four techniques from the fault tolerance domain that allow dealing with system reconfiguration and adress their relevance to our pervasive computing vision. In Phase 4, RM applies the sequence of actual reconfiguration actions and updates the composite service configuration according to the new constituent services.

3.2 . Techniques of Dynamic Reconfiguration

Building upon dependability means [7], there are four means to face connectivity loss: (1) *Change prevention* targets preventing the loss of connectivity with services, (2) *Change elimination* targets reducing the number of connectivity losses and the negative effects of their impact, (3) *Fault tolerance* provides to the system the ability to fulfill system function despite the connectivity loss, and (4) *Change anticipation* estimates the presence and the impact of a connectivity loss on the composite service.

Change prevention and *elimination* (removal) can be encompassed in the notion of *change avoidance*. In fact, preventing users mobility or networked services disconnection is not realizable in pervasive computing environments. Change prevention introduces constraints on the environment, e.g., limiting user mobility or services that can be invoked by users (e.g., only local services) to guarantee connectivity in all situations, whilst, we aim at dealing with the environment's dynamicity and heterogeneity rather than constraining them. Thus, we do not consider change prevention as a way to face connectivity loss. Similarly, for change elimination, we consider that it is difficult to reduce the number of disconnections since disconnections are usually caused by users mobility or services crashes. Thus, reducing the number of disconnections includes limiting users mobility and services types, which constitutes a heavy constraint on the composition and adaptation processes.

Because of the aforementioned reasons, we will only consider *change anticipation* and *tolerance* as means to deal with connectivity loss. In change anticipation, we investigate two techniques: *state transfer* (if the service is stateful) and *replication* since these techniques require some actions to be performed without having any prior awareness of the potential changes that may emerge in the environment, such as invoking two equivalent services to perform the same functionality so that if one of them is disconnected the other service can proceed the conversation with the requester. In change tolerance, we investigate a *rollback* technique and a technique that derives from it, which is the *replay* technique. Both rollback and replay techniques are performed when change anticipation techniques fail or can not be applied. They target returning back to a point of execution from which the composite service can resume its execution without any negative effect of the disconnection on the consistency of service states.

3.3 . Change Anticipation

A *state transfer* technique consists in transferring the *conversation state* of the requester with the disconnected service. The conversation state gives the internal state of the service instance relevant to the execution of a given conversation together with the related execution point. In case of concurrency within the service, we assume that the internal state of the service instance is subdivided into independent session states, where each session corresponds to an active conversation.

There are three necessary requirements for state transfer. First, both the disconnected and its substitute services must provide state transfer capabilities, i.e., operations to export and import their state. Second requirement consists of transferring the state of the disconnected service before its disconnection, which is a form of anticipation of connectivity loss. Third, both the disconnected and its substitute services must be state-compatible in order to initiate the substitute service with the transferred state and proceed the execution from the point where the last version of the state was saved.

In order to satisfy the first requirement, we assume that stateful services give the option to transfer their internal state by providing state transfer operations, e.g., *ExportState()*, which includes packaging the latest consistent session state of the service and sending it to the appropriate storage service and, *ImportState()*, which allows synchronizing the internal state of the substitute service with the disconnected service state for a given conversation.

The second requirement can be satisfied by saving periodically the conversation state between the service requester and the service provider on an elected host. The election process can concern more than one host to increase the chances to not loose the state. The elected host is, by default, the requester host. In case of resource limit, another host must be elected based on resource availability in the environment. The environment can further provide users with storage services (e.g., surrogate service [11]).

Replication of computation is an effective way to achieve change anticipation in distributed systems. In our pervasive computing environment, replication is defined as an invocation of two or more equivalent services available in the environment. Replicas selection is performed from the functional specification of the service to replicate. Since, we consider syntactic replacement, two equivalent services must have the same interface specification but not necessarily the same implementation. Thus, the result type of the services is the same but the result itself can be different. For example, two ”Airline tickets booking” services may return two different lists of flights, depending on the associated flying companies. We call this kind of replication, *loose replication*. Loose replication is decided at runtime and have to be managed by the RM, in opposition to *strong replication* or built-in replication, where replicated services are by design and construction. In the latter case, replication management is already implemented within the application, thus, the adaptation is integrated and can be used without external intervention. Focusing on enabling reconfigurable orchestration that does not necessarily support service replication, we are interested in loose replication. Replication is managed by RM, which sends the client messages to all the replicas and adapts client interaction accordingly. However, in loose replication, the problem of modifying provider state emerges from the multiple interactions with independent replicas. This issue can be illustrated by the problem of validation change. Indeed, if a requester validates his choice, e.g., a reservation or a payment, the validation must be effective only once. Hence, *critical operations* such as payment or validation of a reservation cannot be replicated. Services must specify critical operations that can be performed only once in order to avoid multiple validations. For such operations, replication is applicable only if all replicas synchronize their internal states at the validated operation and resume their execution based on the validated results.

Replication can be applied to all services that take part in a composite service or only to some of them. In fact, replicating all service is costly in terms of resources consumption, whilst the devices populating pervasive computing environments mostly suffer from resources limits. Thus, selecting services that play important role in the composite service and replicating only the critical services is more efficient [4].

Although the aforementioned techniques prepare services to face the occurrence of connectivity loss, there still be many cases that cannot be supported by the change anticipation. Thus, in such cases, reconfiguration must be managed after change occurrence, calling for change tolerance means.

3.4 . Change Tolerance

Rollback [3] is a change tolerance technique that allows returning back the orchestration into a *previous* consistent state when inconsistencies occur because of connectivity loss. Rollback is a checkpoint-based technique that stores – at pre-defined phases of the execution – component services states. The recovery information includes, at a minimum, the states of participating services, called *checkpoints*. Upon a connectivity loss, the composite service’s RM uses the saved information to resume the computation from a previous state, thereby reducing the amount of lost computation. However, considering an orchestration, component services may have input/output dependencies. These dependencies may force some connected services participating to the orchestration to roll back, even though they are not concerned by the disconnection, creating, thus, what is called *rollback propagation*. An example of service dependency in the ”holiday planner” service can be illustrated as follows. The ”airline tickets booking” service has, among its outputs, the arrival airport where the ”car rental” service will provide the chosen car when Richard arrives to the destination country. A Rollback at the ”airline tickets booking” leads the ”car rental” to reconsider its inputs according to the new landing airport. Rollback propagation is an important issue, in pervasive computing environments, specifically when services hosts are mobile. Returning back to a previous state of a service may be complex if the service host is no longer available in the environment. In this case, a state transfer needs to be performed to synchronize the substitute of an unavailable service with the rest of the orchestration participants, which assumes service state availability. In case of unavailability of the disconnected service state, rollback has to be applied on an earlier checkpoint (e.g., before the unavailable service was invoked) or the substitute service can be executed separately according to on its dependencies with the other services. Furthermore, rollback propagation is an important issue concerning provider state modification (e.g., performing critical operations). In this case, the service providers must be *mobility-aware* to allow requesters fulfilling their orchestration despite the previous validations. Mobility awareness can be related to atomicity. Specifically, service providers validate user choices only at the orchestration termination. The orchestration defines an ending checkpoint where all the intermediate validations are actually validated by sending a *validation request* to all component services. A *time out* process can manage the waiting period for the final validation request, at the service provider side.

Replay is a log-based rollback-recovery technique [12], which uses checkpointing and logging to allow services to replay their execution after connectivity loss from the most recent checkpoint. Checkpointing enables the reconfiguration manager to return back to the last consistent state, taking into consideration the connectivity loss. For instance, a selected checkpoint can be situated just before the first invocation of the disconnected service. Logging enables

RM to have the exchanged messages related to the checkpoint, and the exchanged messages until the connectivity loss. The first part of the exchanged messages is used to restore a consistent orchestration state, and the second part is used to replay offline the lost computation. Indeed, logging enables to replay messages *offline*, i.e., without the user intervention, by invoking equivalent services to the disconnected ones, and replaying the execution in order to reach the point where the execution has stopped before the connectivity loss.

3.5 Implementation issues

We plan to extend WSAMI with the aforementioned reconfiguration techniques. An important issue is selecting the appropriate technique for a given configuration of the orchestration. For instance, the state transfer technique cannot be applied between two services unless the substitute service can understand the conversation state received in order to resume the interaction from the point where it is stopped. Furthermore, taking the example of hotel booking, the substitute service should be able to interact with the same hotels in order, for example, to continue a virtual tour conversation that the user was having with the substituted service.

So far, we have been studying algorithms of distributed checkpointing for composite services that deal with the issue of selecting the point at which the execution of the orchestration can be resumed taking into consideration consistency between – provider and requester – services, i.e., compatibility concerning the validated operations.

4 Conclusion and Future Work

In pervasive computing environments, several problems can emerge from dynamicity (e.g, mobility). In this paper, we analysed the requirements to make distributed composite services, specifically orchestrations and orchestrated services, able to face connectivity loss and adapt their configuration to the networking environment. Then, we discussed four techniques to adress these requirements. The presented techniques relate to change anticipation and tolerance. In change anticipation, we investigated the state transfer and replay techniques, while in change tolerance, we investigated the rollback and replay techniques. All the aforementioned techniques are mainly based on state availability of the disconnected service, which we will develop and experiment, in the WSAMI middleware for pervasive web services.

A number of issues are still open, which we will adress in our future work. First, service discovery employed in our scenario is basically syntactic and thus restrictive. Semantic service discovery has to be supported to enlarge the domain of component services that can be used in a service composition/orchestration. We are currently working on a more flexible approach, where interfaces and conversations required by orchestration processes can be adapted in an ad hoc way to interfaces and conversations provided by available services [1]. Also, till now, we have only considered service compositions in the form of orchestrated processes. In a service choreography, a set of services interact with each other in a peer-to-peer fashion and collaborate towards the achievement of a common objective. In a such case, service substitution is harder to perform and requires distributed coordination mechanisms for service discovery, changes detection and state transfer, and process adaptation.

References

[1] S. Ben Mokhtar, J. Liu, N. Georgantas, and V. Issarny. Qos-aware dynamic service composition in ambient intelligence environments. In *Proceedings of the 20th IEEE/ACM Int. Conf. on Automated software engineering (ASE '05)*, pages 317–320, 2005.

[2] T. Berners-Lee, J. Hendler, and O. Lassila. *The Semantic Web*. In Scientific American, 2001.

[3] E. N. M. Elnozahy, A. Lorenzo, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3), 2002.

[4] Z. Guessoum, N. Faci, and J.-P. Briot. Adaptive replication of large-scale multi-agent systems: towards a fault-tolerant multi-agent platform. In *Proceedings of the 4th int. workshop on softawre engeneering for large-scale multi-agent systems (SELMAS '05)*, 2005.

[5] V. Issarny, D. Sacchetti, F. Tartanoglu, F. Sailhan, R. Chibout, N. Levy, and A. Talamona. Developing ambient intelligence systems: A solution based on web services. *In Automated Software Engineering*, 12, 2005.

[6] A. D. Joseph, A. F. deLespinasse, J. A. Tauberand, D. K. Gifford, and M. F. Kaashoek. Rover: a toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, 1995.

[7] B. Lussier, R. Chatila, F. Ingrand, M. Killijian, and D. Powell. On fault tolerance and robustness in autonomous systems. In *Proceedings of the 3rd IARP-IEEE/RAS-EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments*, 2004.

[8] M. Paolucci, T. Kawmura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *1rst Int. Semantic Web Conference*, 2002.

[9] P. Papazoglou and D. Georgakopoulos, editors. *Service-oriented computing*, volume 46. In Communications of the ACM, 2003.

[10] T. Pitkranta, O. Riva, and S. Toivonen. Designing and implementing a system for the provision of proactive context-aware services. In *Proceedings of the Workshop on Context Awareness for Proactive Systems (CAPS)*, 2005.

[11] M. Satyanarayanan. The evolution of coda. *ACM Trans. Comput. Syst.*, 20, 2002.

[12] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3), 1985.

[13] W3C. Web Services Description Language (WSDL) v1.1. Technical report, W3C, 2001. http://www.w3c.org/TR/wsdl.

[14] W. Zhao and H. Schulzrinne. Enhancing service location protocol for efficiency, scalability and advanced discovery. *Journal of Systems and Software*, 75(1-2), 2005.