A Framework for Run-time Behavioural Service Adaptation in Ubiquitous Computing

Javier Cámara[1], Carlos Canal[2], and Nikolay Vasilev[2]

[1] INRIA Rhône-Alpes, France
Javier.Camara-Moreno@inria.fr
[2] Department of Computer Science, University of Málaga, Spain
canal@lcc.uma.es, nikolay.vasilev@gmail.com

Abstract. In Ubiquitous Computing, users interact with multiple small networked computing devices on a daily basis, accessing services present within their physical environment. In particular, the need to discover and correctly access those services as users move from one location to another and the conditions of the environment change, is a crucial requirement in the design and implementation of such systems. This work addresses the discovery and adaptation of services with potentially mismatching interfaces in ubiquitous computing environments, where applications are directly subject to the availability of services which may be discovered or depart from the system's environment at any given moment. In particular, we discuss the design of a framework to enable scalable adaptation capabilities.

1 Introduction

Since the appearance of modern computers, we have shifted from what Mark Weiser once defined as the *Mainframe Age* in which a single computer was shared by many people, to the *Ubiquitous Computing Age* [12], in which a single person commonly interacts with many small networked computing devices, accessing services present within the physical environment. In particular, the need to discover and correctly access those services as the conditions of the environment change, is a crucial requirement in the design and implementation of such systems.

Services are independently developed by different service providers as reusable black boxes whose functionality is accessed through public interfaces. The heterogeneity of service implementations, which are not designed to interoperate with each other in most situations, commonly results in the appearance of mismatch among their public interfaces when they are composed. Specifically, we can distinguish four interoperability levels in existing Interface Description Languages (IDLs): *(i) Signature*. At this level, IDLs (*e.g.,* Java interfaces, WSDL descriptions) provide operation names, type of arguments and return values. Signature interoperability problems are related to different operation names or parameter types between provided and required operations on the different interfaces; *(ii) Protocol* or *behaviour*. Specifies the order in which the operations available on an interface should be invoked. If such orders are incompatible among services, this may lead situations such as *deadlocks* or infinite loops. Notorious examples of behavioural interface descriptions include Abstract BPEL, automata-based languages such as UML state diagrams, or high-level MSCs; *(iii) Functional* or *Semantic*. Even if service interfaces match at the other levels, we must ensure that they are going to fulfill their expected behaviour. This level of description provides semantic information about services using ontology-based notations such as OWL-S (used in Web Services), which are interesting for service mining; and *(iv) Service*. Description of other non-functional properties like temporal requirements, security, etc.

Software Adaptation [4, 13] is the only way to compose services with mismatching interfaces in a non-intrusively manner by automatically generating mediating *adaptor* services able to solve interoperability problems between them at all levels. Particularly, in this work we focus on the protocol or behavioural level, currently acknowledged as one of the most relevant research directions in software adaptation [4, 2, 10, 13]. Most approaches in behavioural software adaptation build adaptors for the whole system at design-time, a costly process that relies on the specific set of services involved in the system. Hence, these techniques (often referred to as static approaches) are not suited for ubicomp environments, since the adaptor would have to be recomputed each time a new service is discovered or departs from the current system configuration. On the contrary, in some recent developments [11, 5, 7] a composition engine enacts adaptation at run-time. In particular, Cámara et al. [3] addressed run-time adaptation in the specific context of ubicomp environments. However, although this approach lays out the formal foundations of adaptation in such environments, the reference architecture used by the authors (and in general by all the aforementioned run-time approaches) does not consider scalability issues, and includes a single central adaptation unit which performs all the adaptation between all services in the system, something that turns this central node into a performance bottleneck.

Here, we extend the work in [3], providing architectural support for service adaptation in ubicomp environments, where applications are directly subject to the availability of services which may be discovered or depart from the system's environment anytime. In particular, we discuss the design of a framework to enable scalable adaptation capabilities and support of service discovery which considers behavioural information about the services available in the environment.

To illustrate our approach, we will use a running example described in the context of an airport: let us suppose a traveller who walks into an airport with a PDA or a smartphone equipped with a client containing a client application based on the different services which may be accessed through a local wireless network at the airport. First, the user needs to contact his airline and check-in in order to obtain a seat on the flight. This is achieved by approaching a kiosk which provides a local check-in service available to the handheld device. Next, the traveller may browse the duty-free shops located at the airport. The selected shop should be able to access the airport information system in order to check if a passenger has checked-in on a particular flight, and apply a tax exemption on the sale in that case. The payment is completed by means of credit card information stored in the traveller's device.

In the rest of this paper, Section 2 presents our service model and the run-time adaptation process in ubicomp environments. Next, Section 3 discusses the design and implementation of our framework. Finally, Section 4 compares our approach with the related work in the field, and Section 5 concludes.

2 Behavioural Adaptation in Ubiquitous Computing

In this work we use a service interface model [3] which includes both a signature, and a behavioural interface. In order to enable composition with services which are discovered at run-time, we extend the behavioural interfaces with additional information. In particular, our behavioural interface consists of: **(i)** a protocol description (STS); and **(ii)** a set of correspondences between abstract and concrete service operations.

Definition 1 (STS). *An STS is a tuple (A, S, I, B, T) where: A is an alphabet that corresponds to message events relative to the service's provided and required operations, S is a set of states, $I \in S$ is the initial state, $B \in S$ are stable states in which the service can be removed from the current system configuration if it is not engaged in any open transactions and, and $T \subseteq S \times A \times S$ is a transition relation.*

Generic Correspondences. Adaptor generation approaches commonly rely on interface mappings that express correspondences between operations names, as well as parameter types and ordering on the different interfaces. However, these mappings can be produced only once the description of the different interfaces is known. In ubicomp environments, this information is only available when services are discovered at run-time. Specifically, when we are describing the protocol of a service to be reused in such systems, we know what the required operations are, but we do not know which specific service implementation is going to provide them, or even the specific name of the concrete operation implementing the required functionality.

Approaches to run-time service discovery use *service ontologies* that describe the properties and capabilities of services in unambiguous, computer-interpretable form. Here, we assume that services available in the environment are exposed using such descriptions, which will be used as a reference to relate service interfaces. For simplicity, instead of using any of the emerging standards for the semantic description of services, we will use in the remainder a notation which abstracts away specific notations for service descriptions such as OWL-S [6].

Definition 2 (Abstract Operation Signature). *An* abstract operation signature *is the name of a generic operation, together with its arguments and return types defined within the context of a service ontology.*

Definition 3 (Abstract Role). *We define an* abstract role *as a set of abstract operation signatures associated with a common task or goal.*

Our model of interface makes explicit correspondences of how labels in the protocol description (STS) are related with generic operations described by the service ontology. To do so, we also rely on vectors (based on synchronous vectors [1]). However, in this application domain, we have to make a distinction between the generic correspondences established in vectors, and what we call *vector instances*, which relate actual service interface operations.

Definition 4 (Vector). *A vector for a service STS (A, S, I, B, T) and an abstract role, is a couple $\langle e_l, e_r \rangle$ where e_l is a label term for A, and e_r is an abstract label term for* *an abstract role. A label term t contains the name of the operation, a direction(?/!), and as many untyped fresh names as elements in the argument type list (placeholders). An* abstract label term *is defined in the context of an abstract role, instead of a service interface.*

Definition 5 (Vector Instance). *A vector instance for a pair of service STSs $(A_l,\ S_l,\ I_l,\ B_l,\ T_l)$ and (A_r,S_r,I_r,B_r,T_r) is a couple $\langle e_l, e_r \rangle$ where e_l, e_r are label terms in A_l and A_r, respectively.*

Vector instances are obtained from vectors by binding concrete service interface operations at run-time. Specifically, for each vector v, we extract all the other vectors on the counterpart interfaces including abstract label terms corresponding to the same abstract operation signatures.

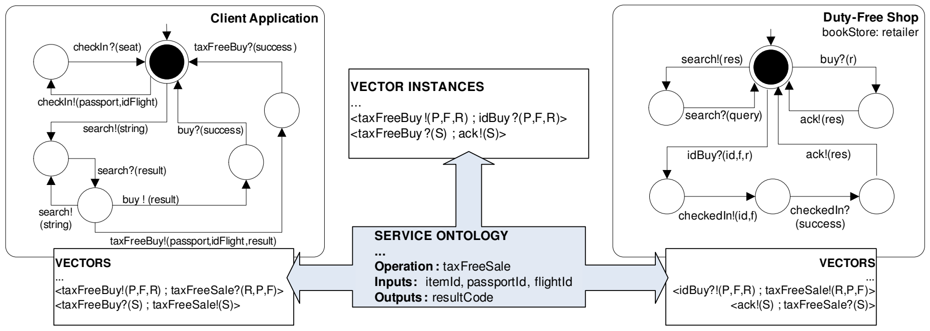

Fig. 1. Protocol STS and vector instantiation example for the client and the store.

Fig. 1 depicts an example of vector instantiation between the client and the online store in our case study: concrete interface operations taxFreeBuy and idBuy are related through the taxFreeSale abstract operation in the service ontology. Placeholders P,F, and R are used to relate sent and received arguments between the operations.

Once we have described interfaces and processed vector instantiation, we have to compute the reachability analysis of stable states being given a set of service protocols, and a set of instantiated vectors. A stable state of the system is one, where each of the services in the system is on a stable state. It is only at this point that services can be incorporated or removed, and the system properly reconfigured. To perform this adaptability analysis we use a depth-first search algorithm which seeks stable system states, and stops as soon as a final state for the whole system has been found. If the analysis determines that global stability can be reached, then the execution of the system can be launched (Fig. 2).

3 Framework Architecture

Although the service composition process presented in last section enables interoperability between services with potentially mismatching interfaces at run-time, this pro-

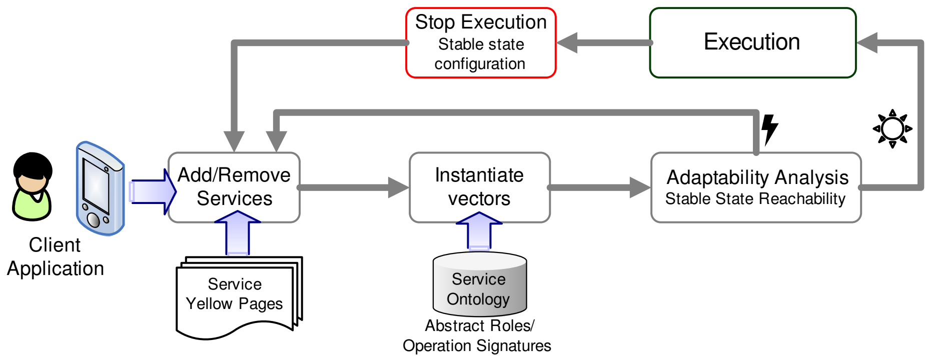

Fig. 2. Overview of the adapted service composition process.

cess also requires support from an architectural point of view in order to put this approach into practice. Specifically, we can identify two important problems that need to be addressed in order to allow scalable adapted service interaction in ubiquitous computing environments:

- Adaptation. We have to rule out design choices which consider a single central adaptation unit performing all the adaptation between all services in the system. This can quickly turn into a performance bottleneck.
- Service Discovery. Our framework must facilitate the discovery of candidate service implementations for client applications. The discovery mechanism must take into account behavioural descriptions of services. Moreover, we cannot load a single service registry with the task of checking if a service is adaptable for the purposes of a client application at the behavioural level.

In order to provide support for the adaptation process described in the previous section, we have designed a framework that tackles the aforementioned problems. Fig. 3 depicts the architecture of our framework, where we can distinguish the following components:

Client Applications and Service Providers Both clients and services expose their functionality public interfaces (*Behavioural Interface Descriptions* or BIDs) which describe provided and required operations, as well as a description of their protocol (STS), which are necessary for behavioural adaptation. In the case of service providers, BIDs are accompanied by a thin *proxy component* that is imported to the client side and is used for communication with the service's implementation. A service BID, along with its corresponding proxy is known in our framework as a *service entry*. These entries are used by clients to discover suitable service implementations. Further details about their use can be found in Section 3.1.

Adaptation Manager In our framework, the adaptation process is performed always at the client's side by a component called *adaptation manager*, which stays in between

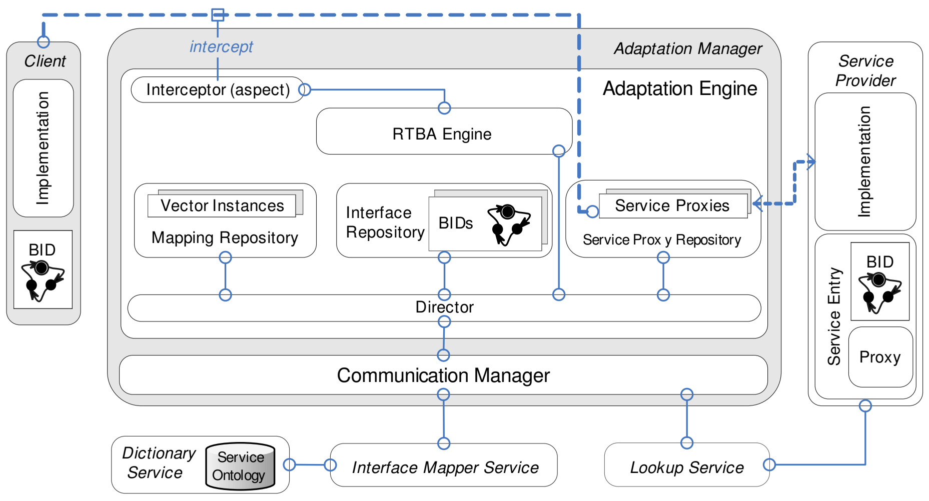

Fig. 3. Framework architecture.

the client application and the rest of the elements in the environment. This manager consists of two main components: **(i)** the *adaptation engine* in charge of handling all adaptation-related tasks. The main subcomponent of the adaptation engine is the *run-time behavioural adaptation engine* (RTBAE for short) where the global stability check and the adaptation algorithms are implemented. In the process of the adaptation other subcomponent - the *interceptor*, implemented as an aspect, is used to notify the RT-BAE when an operation is invoked on the client application or on the *service proxy*. To perform adaptation, the RTBAE needs the client and service BIDs and also appropriate vector instances. The vector instances, along with the BIDs and all service proxies are stored in dedicated components of the adaptation engine, the *mapping*, *interface* and *service proxy repositories*. Finally, the *director* subcomponent is a mediator that is in charge of coordinating all operations within the adaptation engine; and **(ii)** the communication manager, which is responsible for providing the adaptation manager with all the elements required to perform the adaptation, communicating with the lookup and interface mapper services.

Network Services The framework also includes a set of network services that enable service discovery and matching: **(i)** the *Lookup service* performs service matching over the set of service implementations registered in it, determining the set of service entries that match the request criteria of the client. To do this, it uses the client's BID provided by the client's communication manager, and the BID in the service entry of each candidate service, **(ii)** the *interface mapper service* produces vector instances for the adaptation process using one or more **(iii)** *dictionary services*, which contain a representation of the valid abstract service types and operations which may be exposed in the environment.

3.1 Framework Component Interaction

Initially, after the lookup service starts up, the dictionary and interface mapper services start up and register with the lookup service (Fig. 4). Afterwards, when the service providers appear in the network, they register with the lookup service sending their service entries.

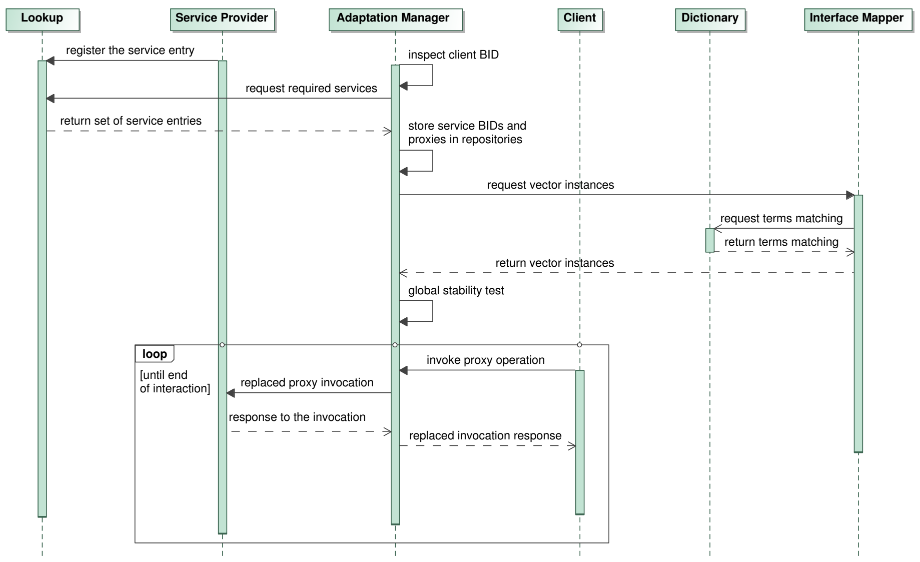
Lookup
Service Provider
Adaptation Manager
Client
Dictionary
Interface Mapper
register the service entry
inspect client BID
request required services
return set of service entries
store service BIDs and proxies in repositories
request vector instances
request terms matching
return terms matching
return vector instances
global stability test
loop
[until end of interaction]
invoke proxy operation
replaced proxy invocation
response to the invocation
replaced invocation response

Fig. 4. Dynamics of the process followed for service discovery and adaptation.

Service Discovery When a client enters the network, its associated director stores its BID in the interface repository. The communication manager then queries the lookup service for available service implementations in the network that could provide operations required by the client. This query includes the information in the client's BID. As a result, a list of service entries is sent to the client's communication manager. The list contains only one service for each type required by the client. The director stores for each received service its BID and proxy into their respective repositories. Then, the communication manager bundles the BIDs of the client and the services obtained in the previous matching, and sends them to the interface mapper, which returns the set of vector instances needed for the adaptation. Finally, the director stores them in the mapping repository.

The director extracts the client and required services BIDs from the interface repository and supplies them to the RTBAE, which uses them to execute the global stability analysis in order to ensure that during execution, the system is not going to end up in a deadlock state. If the result from this check is positive, the real adaptation process starts. Otherwise, the RTBAE starts to discard proxies. In the latter case, the director first removes each discarded proxy and BID from their respective repositories, followed by the related vector instances from the mappings repository. Then the communication manager tries to find a replacement service from the discarded type of service. The discarded service entry will not be accepted by the communication manager if the lookup service sends it back again, unless the set of services in the different repositories has changed since the time of the last service entry request. When a replacement service implementation is found, the process starts all over again. When the global stability check succeeds, the framework proceeds with the actual adaptation process.

Adapted Interaction The adaptation process in our framework implements an extended version of the algorithm presented in [2]. During this process, the interceptor component intercepts all operation invocations in the service proxies and notifies the RTBAE about the invoked operation. The RTBA engine replaces the invocation according to the indications of the RTBAE, which in turn determines the invocation substitutions to apply according to the adaptation algorithm and the message correspondences encoded in vector instances stored into the interface mapping repository. Once the system reaches stable state, the director removes the proxies, BIDs and vector instances which are not needed anymore from the corresponding repositories. If from the current stable state, the client application has all the required service proxies and BIDs to continue operating, the adaptation process starts all over again. Otherwise, the adaptation manager keeps on performing service discovery until this condition is fulfilled.

4 Related Work

So far, there are few proposals in the literature where a composition engine enacts behavioural adaptation at run-time, not requiring the explicit generation of an adaptor protocol description at design-time. Dumas et al. [11] introduce a service mediation engine which interprets an interface mapping obtained at design-time. Internally, this engine relies on an abstract representation of behavioural interfaces as FSMs. The engine manipulates the exchanged messages according to the interface mapping provided (expressed as a set of production rules). It is worth observing that a deadlock is only detected once the current execution of the engine terminates. Hence, deadlocks are only considered as a termination anomaly, but cannot be prevented.

Cavallaro and di Nitto [5] propose an approach to allow the invocation of services at run-time with mismatches at the signature and protocol levels. The authors consider that the user is going to specify the task to be carried out by implementing a service composition with respect to a set of abstract service interfaces, instead of real service implementations, which will only be known at run-time. Their approach includes a framework for service composition which takes a BPEL process specified with respect to a set of abstract services as input. When an abstract service is invoked by the process, the call is forwarded to a proxy for that abstract service, which subsequently forwards the call to the concrete implementation of the service selected at run-time. The actual adaptor is inserted in the communication between the proxy and the concrete service implementation. However, the authors propose to map states and transitions between abstract and concrete service protocols, assuming that protocols have equivalent states and transitions. This is a strong assumption that reduces drastically the number of cases where adaptation can be performed. Indeed, no means are provided to systematically deal with deadlocks in the resulting adaptors.

Moser et al. [7] present a system that allows the monitoring of BPEL processes according to Quality of Service (QoS) attributes, as well as the replacement of partner services involved in the process at run-time. The alternative service replacements can either be *syntactically* or *semantically* equivalent to the original. To enable interoperability, this approach addresses adaptation through the use of special components named *transformers*. These are mediators that compensate interface mismatches between the original service and its replacement by applying transformation rules to incoming and outgoing messages. Both monitoring and adaptation facilities are implemented using AOP. Although this approach enables substitution of services at run-time with a certain degree of adaptation, important behavioural aspects of services are not addressed (e.g., guaranteeing deadlock freedom when services are integrated into the BPEL process).

Finally, Cámara et al. [3] contribute an interface model which relates abstract service types with concrete implementations and an extended version of a run-time adaptation engine that enables behavioural service adaptation at run-time. In this case, behavioural interface descriptions are used to perform adaptability check of the services before they start their (adapted) interaction, ruling out undesired behaviours such as deadlocks or livelocks. Unfortunately, there is no architectural support provided for their approach. Specifically, although the authors mention the possibility of distributing the adaptation process, this point is not addressed and they assume a single centralized adaptation unit which performs all the adaptation of service interactions.

5 Conclusions and Future Work

In this work, we have presented a proposal to support run-time behavioural adaptation in ubiquitous computing environments. In particular, we have discussed the design of a framework that enables run-time service discovery, taking into account behavioural service interface information and transparent adaptation of service interactions. Moreover, we have implemented a prototype of our framework, extending the Java Jini service platform using Aspect-Oriented Programming (AspectJ).

With respect to future work, in the current version of our framework only one service instance of each required type is returned by the dispatcher to the client. This is intended to reduce network traffic, but if the reachability check in the client virtual machine fails, the process of retrieving services would need to restart, consuming too much time and client resources, and potentially causing heavy network traffic. Hence, we think that optimizing the interaction protocol between the client and the dispatcher, as well as the service matching mechanisms is an interesting direction that could represent a noticeable improvement of the framework's performance. Moreover, right now if a service disappears in a non-stable state, the current transaction has to be aborted. In relation with this problem, we aim at enabling reconfiguration, using execution trace equivalence checking in order to replace departing services in the middle of a running transaction.

References

1. A. Arnold. *Finite transition systems: semantics of communicating systems*. Prentice Hall International (UK) Ltd., 1994.
2. J. Cámara, C. Canal, and G. Salaün. Composition and run-time adaptation of mismatching behavioural interfaces. *Journal of Universal Computer Science*, 14:2182–2211, 2008.
3. J. Cámara, C. Canal, and G. Salaün. Behavioural self-adaptation of services in ubiquitous computing environments. *Software Engineering for Adaptive and Self-Managing Systems, International Workshop on*, pages 28–37, 2009.
4. C. Canal, J. M. Murillo, and P. Poizat. Software adaptation. *L'Objet*, 12(1):9–31, 2006.
5. L. Cavallaro and E. D. Nitto. An approach to adapt service requests to actual service interfaces. In *Proceedings of Software Engineering for Adaptive and Self-Managing systems (SEAMS 2008)*, pages 129–136. ACM, 2008.
6. T. O. S. Coalition. OWL-S: Semantic markup for web services, 2004. http://www.daml.org/services.
7. O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for WS-BPEL. In J. Huai, R. Chen, H. Hon, Y. Liu, W. Ma, A. Tomkins, and X. Zhang, editors, *Proceedings of the 17th International Conference on World Wide Web, (WWW '08)*, pages 815–824. ACM, 2008.
8. J. Newmarch. *Foundations of Jini TM 2 Programming*. Apress, Inc., 2006.
9. M. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Fourth International Conference on Web Information Systems Engineering (WISE'03)*, 2003.
10. K. Wang, M. Dumas, C. Ouyang, and J. Vayssière. The service adaptation machine. In *ECOWS*, pages 145–154. IEEE Computer Society, 2008.
11. K. Wang, M. Dumas, C. Ouyang, and J. Vayssiere. The service adaptation machine. In *Proceedings of the 6th IEEE European Conference on Web Services (ECOWS'08)*, pages 145–154. IEEE Computer Society, 2008.
12. M. Weiser. Hot topics: Ubiquitous computing. *IEEE Computer*, 1993.
13. D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.