Application-Service Interoperation without Standardized Service Interfaces

Shankar R. Ponnekanti and Armando Fox
Computer Science Dept
Stanford University
Stanford, CA 94305
$\{$*pshankar, fox*$\}$*@cs.stanford.edu*

Abstract

To programmatically discover and interact with services in ubiquitous computing environments, an application needs to solve two problems: (1) is it semantically meaningful *to interact with a service? If the task is "printing a file", a printer service would be appropriate, but a screen rendering service or CD player service would not. (2) If yes, what are the mechanics of interacting with the service – remote invocation mechanics, names of methods, numbers and types of arguments, etc.? Existing service frameworks such as Jini [1] and UPnP [22] conflate these problems—two services are "semantically compatible" if and only if their interface signatures match. As a result, interoperability is severely restricted unless there is a single, globally agreed-upon, unique interface for each service type. By separating the two subproblems and delegating different parts of the problem to the user and the system, we show how applications can interoperate with services even when globally unique interfaces do not exist for certain services.*

1 Introduction

A key problem in ubicomp (ubiquitous computing) environments is allowing applications to discover and interact with network services. (By a service, we mean either a device or a software application that is always available and accessible over the network by other apps and services.) For an application to automatically discover and interact with the services in its local environment, two issues need to be addressed:

1. I_1: Identifying *semantically compatible* services: By a "semantically compatible service", we mean a service that meets the needs of the application. For example, if the task is "printing a file", a printer service is a compatible service, but a screen rendering service or CD player service is not.
2. I_2: Determining *invocation mechanics*: Once a compatible service is identified, the application needs to determine the mechanics for invoking various operations on the service over the network and obtaining the result (if any). In other words, after a printing application has identified a printer, it still needs to know the exact method name(s) and parameter types for the print method(s) before printing can be realized.

Today's ubicomp service frameworks such as Jini and UPnP "solve" these issues by requiring service vendors and application writers to agree on a unique interface standard for each service type – i.e., a single printer interface, a single CD player interface, a single projector interface, and so on for *every* service type. We call this the **single-standard service** requirement. With a single standard for each service type, both I_1 and I_2 are easy to handle. For I_1, the application simply looks for services exporting the standard interface for the needed service type, and for I_2, the application uses the methods and parameters of this standard interface. Unfortunately, however, agreement on a single interface for every service type is unlikely to occur for all services. For a set of vendors to agree on a standard interface is hard, even for well-understood services. For example, as noted in [13], the Jini printer forum required 15 months to announce a draft of version 1.0 of the printer standard, and it is far from clear if this will be widely adopted. (The cited source itself points out that this draft is just a "first step".) For newer and less understood services, standardization is even harder. Very often, standardization occurs through market-driven consolidation, and such a process can take several years. Since new services will continue to appear, there will always be services for which standardization/consolidation has not yet occurred, if at all.

Thus, we argue that ubicomp frameworks should allow interoperation for for *multi-standard services* – service types for which multiple interfaces exist and different providers may subscribe to different interfaces. For such multi-standard services, applications under Jini or UPnP will not discover and interact with a service if the service exports a different interface, even if the service was actually semantically compatible. A "solution" of the form "every application should be written to understand all the interfaces for each service type it requires" is infeasible, because some of these interfaces may appear after the application was written, aside from the fact that such a solution places too much burden on the application programmers.

In this paper, we propose a framework that generalizes current approaches by allowing interoperation for multi-standard services. Note that both the issues I_1 and I_2 become much harder under multi-standard services, and our framework addresses these issues as follows:

- For I_1, we present techniques to identify related services, even in the absence of a single-standard for the needed service type.
- For I_2, we propose a strategy that allows an application to interoperate with the service, even when the service exports a different interface than the one used by the application.

We are not against standardization per se - after all, standards such as 802.11 and TCP have had a vital role in the development of computer science. However, 802.11 and TCP do *not* try to capture and standardize application-specific semantics, and are not specific to a search service, a book-buying service, or a printer. Unlike TCP/802.11, Jini/UPnP also attempt to achieve global agreement on application-level semantics by standardizing interfaces for each service type, which is difficult because there are numerous services and new ones will continue to be developed. The approach we describe relies on a standardized underlying application-independent layer similar to Jini/UPnP, but does not require global agreement at the application level.

Note that frameworks such as UPnP [22], Hodes et al. [11, 10] and ICrafter [19] allow end-users (as opposed to software applications) to directly discover and interact with the available services. Direct end-user control, while certainly useful, is not enough because functionality more complicated than the direct control of one or more services should ideally be provided as a bundled application to end-users. For example, giving a lecture in a seminar room typically involves the following functionality: turning on/off the lights/cameras/displays, recording the audio/video, printing out slide handouts for the audience, and making effective use of the multiple displays in the seminar room for displaying the slides. While all these activities can also be performed by manually controlling each individual service, it gets clumsy and repetitive. So, this functionality needs be coded as a lecture service/application that in turn accesses the light, display, camera, and printer services programmatically. It is in the context of programmatic control by applications that the issues I_1 and I_2, and this paper become relevant.

A comparison to the Web further clarifies the difference between end-users accessing the services directly vs. applications accessing the services programmatically. [1] Current Web services are designed for direct control by end-users, where each web service exports a UI (a Web form) that is used by the users to interact with the service. Much like the ubicomp world, programmatic access through applications is clearly desirable on the Web too – to enable automation of repetitive tasks, etc. Industry efforts currently underway to allow applications/agents to access Web services programmatically through automated discovery (UDDI [21]) and remote invocation (SOAP/WSDL [23, 24]) also require solving the exact same issues I_1 and I_2. For example, unless all current (and future) search service providers such as Google, Hotbot, AltaVista, etc agree to export an identical SOAP search interface, a search application written to a particular search interface will fail to interoperate with all the search service providers.

The ideas presented here are also relevant to Web services, and in general to any domain characterized by three aspects:

1. Need for *programmatic access*: Software applications (that provide bundled functionality and automate repetitive tasks etc) need to discover and interoperate with services.
2. Need for service *substitutability*: The same application needs to work with different (compatible) services at different times. For example, a print application may need to work with an HP printer in one environment and a Canon printer in a different environment.
3. Presence of *interactivity*: The application is run interactively by the end-user, and the user is available (if necessary) for making semantic decisions.

In this paper, we will largely focus on the ubicomp domain, but the example service we describe (search service) in some detail later is relevant for both the Web and ubicomp contexts. The rest of the paper is organized as follows. Sections 2 and 3 describe the framework, while section 4 presents a prototype implementation. In sections 5 and 6, we survey related work and conclude.

2 Framework

A typical existing framework such as Jini relies on single-standard services and works as follows: Services advertise to local registries the interfaces implemented by the

[1] For our purposes, the distinction between a Web service (such as Amazon) and a ubicomp service (such as a printer) is that the scope of the latter is limited to the local environment.

service as well as attributes describing various features of the service. To determine compatible services, applications search for services exporting the standardized interface for the needed service type. In addition to the interface, applications also use the attributes advertised by services to choose the service that is most appropriate for the task at hand. The application then uses the operations in the standard interface to interact with the service.

Below, we explain how applications perform these functions even for multi-standard services in our approach. In the framework to be proposed, we assume:

1. Standardized application-independent mechanics: remote invocation framework and discovery/advertisement protocols.

2. Descriptions: Service advertisements include a human-understandable natural language description of the service supplied by the service author or the local admin.

The first assumption is consistent with most existing frameworks. A few existing frameworks also provide human-understandable service descriptions.

2.1 Addressing I_1: Semantic Compatibility

In general, fully automated discovery and determination of the target services for an application requires that all relevant aspects of the user intent be codified and expressible by the application using the service attributes. In reality, however, user intervention may often be needed because:

- An application designer may not have taken into consideration one or more relevant aspects because they were hard to conceive in advance. For example, in the days prior to flat screen TV's, an application designer may not have anticipated this feature.

- When no available service matches all the needed attributes, the application may not be able to choose from among the available services. For example, suppose that a user wants both color printing and a certain resolution, and that printer 1 provides color printing but not the needed resolution, while printer 2 provides the needed resolution but only gray-scale printing. Which printer to select (if any) depends on the user intent, and cannot be automated by the application.

Thus, even for single-standard services, the application should always provide the user a “manual-select” mode to override the automated selection process and select the service(s) manually. We generalize this strategy and apply it to multi-standard services as follows:

1. An application first attempts to discover services exporting the interface known to the application.

2. If the application discovers such a service, it can simply use the discovered service and user intervention is not triggered, assuming that user has enabled the “auto-select” mode.

3. On the other hand, if the application fails to discover a service that exports the known interface, or if the application is running in the manual-select mode, the application displays a list of all the available services along with their advertised (human-understandable) descriptions, and requests the user to select the appropriate service to interact with.

Relying on users to select semantically compatible services is not unique to our work. In particular, the SpeakEasy project [4] at Xerox also uses this approach. However, they do not address the multi-standard service interoperation problem. Instead, they still use single-standard services but advocate that applications be written to use domain-independent interfaces (such as generic data exchange interfaces) rather than domain-specific interfaces (such as print interface and display interface). We do not mandate that the service standards be domain-independent, even though domain-independent interfaces can be easily accommodated into our framework (as described in section 3.3). In addition, we also present important techniques in section 3.2 for improving the basic approach outlined here. (Displaying the list of all the locally available services is clumsy, especially when the environment contains a number of services.)

2.2 Addressing I_2: Invocation Mechanics

The challenge is to achieve interoperation in the case where the user identifies a compatible service but the service exports a different interface (say Y) than the one known to the application (say X). Fortunately, we can leverage a wealth of existing techniques and approaches to deal with this issue. First, as per the well known adapter design pattern [6], an *adapter* can be written from interface X to interface Y that implements operations of X using Y. However, in our case, interoperation has to occur on-the-fly, and we cannot assume that the end-user will supply an adapter. However, if there exist repositories where programmers have registered such adapters, the application can use mobile code technology to dynamically locate and load such adapters. Henceforth, we will refer to a repository of adapters as a *glue directory*.

A key observation is that transitive chaining of adapters – an X-Y adapter and a Y-Z adapter can be chained to obtain a “composite” X-Z adapter – implies that we do not need

$N*(N-1)$ adapters to allow interoperation between N different interfaces. In the best case, we only need $2*(N-1)$ handwritten adapters to allow for interoperation between N interfaces, which is only two adapters per interface. To represent such chaining, we introduce some terminology below:

- **Stub:** A stub ${}_IS$ exports interface I and implements the operations of I by performing operation invocations over the network on a service S exporting the interface I. The interface I is known as the source interface of ${}_IS$.
- **Adapter:** An adapter ${}_IA_J$ exports the interface I, and uses the interface J to implement the operations of I. The interface I is referred to as the source interface of the adapter, while the interface J is referred to as the target interface.
- **Proxy:** A proxy ${}_IP_J$ is a collection of adapters and stubs that exports the interface I but uses a service exporting interface J to implement the operations of I. A proxy can be obtained by chaining one or more adapters and a stub. For example, the sequence ${}_IA_{I_1}\ {}_{I_1}A_{I_2}\ \ldots\ {}_{I_{m-1}}A_J\ {}_JS$, containing m adapters followed by a stub, represents a proxy from I to J.

In the general case, we allow an adapter to have more than one target interface. For example, an adapter ${}_IA_{J_1,J_2,\ldots,J_n}$ implements the operations of I using the interfaces $J_1, J_2, \ldots, J_n$. This implies that the proxy may be a tree (instead of a linear chain) of adapters and stubs.

The notion of chaining format converters has been studied in data transformation as in TOM [17] and data type compatibility based service composition frameworks such as CANS [5] and Paths [14]. However, in our case, chaining is based on interfaces, and not data types. Input/output data type based chaining has limited applicability because it can only be applied for components whose behavior can be modeled as consumers and/or producers of data. On the other hand, interfaces can be used to model arbitrary component behaviors, and thus interface-based chaining has much wider applicability.

2.3 Application-level vs Application-independent Standards

Note that although our approach allows interoperation for multi-standard services, it requires additional standardization for locating and loading stubs and adapters. It may appear that all the framework does is to move standardization from one place to another. However, there is a crucial difference: While service standards involve standardization of application-level semantics, locating and loading stubs is application-agnostic. Widely deployed existing standards such as TCP, HTTP and XML do not try to capture and standardize application-specific semantics. Application-level semantics are much harder to standardize because:

- Hundreds of applications can be built atop SOAP and HTTP. Standardization of HTTP and SOAP is a one-time effort, but standardizing the specific SOAP interfaces for each type of application is a daunting task. Web service providers such as Google and Amazon have unilaterally announced their SOAP interfaces, and there is little reason to believe that other search and book sales providers will adopt Google and Amazon SOAP interfaces, all the more if other providers become dominant in the years to come.
- Much like other technologies, new services can be expected to go through a lifecycle. Multiple standards are likely to exist during the initial years, and standardization or market-driven consolidation could occur over a period of time. At any point in time, there will always be services at different points in the lifecycle, since new types of ubicomp and web services will continue to be introduced. Thus, multi-standard services are always likely to exist.
- Application semantics are often tied to vendor innovations and their value propositions. New application features are often introduced independently by different vendors and can only be standardized over a period of time.

3 Framework Details

In this section, we further develop the framework and address several issues in greater detail.

3.1 Lossy Adapters

An adapter may be *lossless* or *lossy*. A lossless adapter ${}_XA_Y$ provides a valid and complete implementation of interface X (as per X's specs), while a lossy adapter either does not provide some of the functionality of X (since it is not supported by Y), or results in side-effects not intended by interface X. If a programmer certifies an adapter ${}_XA_Y$ as lossless (much as a programmer certifies a native implementation as conforming to interface X), then [the ${}_XA_Y$ adapter + a service implementing Y] may be considered a "native implementation" of X that uses Y as a third-party library/service. After all, a native implementation could also use third-party libraries or services. In general, we leave it to the discretion of the programmer to determine whether a lossless adapter can be written, and if not, if a useful lossy adapter can be designed. With lossy adapters, the key is to allow an application to detect the nature of the loss in functionality (ideally, before an operation is invoked), so it can adapt its behavior suitably such as by disabling some widgets in its UI or by notifying the user.

One strategy to facilitate the detection of loss of functionality is require that all stubs and adapters provide a method `isSupported(methodname, paramname)`. This method is not the same as reflection, but is supposed to return `true` if the method and parameter have been implemented completely by this adapter, and return `false` otherwise. In particular, a lossless adapter should always return `true` when `isSupported` is invoked with any methodname and parameter. On the other hand, a lossy adapter should return `false` for those methods and parameters that are not implemented by the adapter. Much like a lossless adapter, a stub ${}_IS$ is expected to always return `true` when the `isSupported` operation is invoked, since the stub is interacting with a service that exports I, and should hence support all operations of I. [2]

With chaining, an adapter cannot determine by itself if a method and parameter (m,p) can be supported, since the answer may depend on whether the following adapter supports the methods needed to implement (m,p). However, this issue can be resolved cleanly by using a "recursive" strategy – an adapter can use the `isSupported` method of the target interface to implement the `isSupported` method of the source interface. For example, if an adapter ${}_XA_Y$ implements an operation `foo(p1)` of X using the methods `bar(p2)` and `baz(p3)` of Y, then the adapter should return true for `isSupported('foo','p1')` iff the following adapter (or stub) in the chain returns true for both `isSupported('bar','p2')` and `isSupported('baz','p3')`. This "recursion" finally terminates at the stub.

Note that the `isSupported` mechanism as described above does not work well when an adapter only partially supports a method and parameter. An obvious extension is to allow the `isSupported` method to return `partial` in addition to `true` and `false`, but we are exploring a more systematic solution.

3.2 Techniques for Automated Service Selection

The strategy for selecting compatible services described in section 2.1 works in principle, but is clumsy in practice. Leaving the problem of selecting compatible services entirely to the end-user is not desirable since it places too much burden on the users. In particular, consider a user running a print application that expects services exporting printer interface P_1. When discovery fails for interface P_1, the solution described in section 2.1 simply displays a list of *all* services in the environment – lights, projectors, document search services, audio/video players, etc – many of which are completely unrelated to the print service. The list shown to the print application user should preferably show only "related" services, such as printers exporting P_1 or other print interfaces. In the absence of single-standard services, we apply the following strategy to automate the selection of "related" services:

- An application requiring interface I first searches for services exporting interface I.
- If the above fails, the application searches for all services that export any interface J such that there exists a proxy ${}_IP_J$. This implies that there exists a sequence of adapters and stub from interface I to interface J, as per the proxy definition of section 2.2. We refer to such services as *proxy-related services*.

Further improvements are possible. In particular, if the service selections chosen by the user are cached on a per-environment basis by an application, user intervention would be needed only the first time the application is run in a new environment. The application can allow the user to simply reuse the selections made during the first run for further runs in the same environment. This strategy can also applied across users: If there exist a large number of proxy-related services, selections chosen by fellow users can be prioritized.

3.3 Composition and Domain-independent Interfaces

In addition to interoperation, the approach described in this paper also enables composition scenarios where an interface can be realized using two or more other interfaces. For example, consider an application that uses a copier interface running in an environment that does *not* contain a copier, but contains a scanner and a printer. With our framework, the copier application can still function in the said environment by dynamically loading an adapter that implements the copier interface using the printer and scanner interfaces. As another example, in an environment lacking a printer, printing can be achieved by utilizing an online fax service and a local fax machine.

Besides composition scenarios, adapter-based chaining also accommodates domain-independent interfaces. As noted earlier, the proponents of the SpeakEasy [4] project advocate that applications be written to use domain-independent interfaces (such as data exchange interfaces) rather than domain-specific interfaces (such as print inter-

[2] Some interfaces could specify certain methods and parameters as optional, in which case the stub can query the service using the appropriate service-specific mechanism to determine which of the optional methods and parameters are supported.

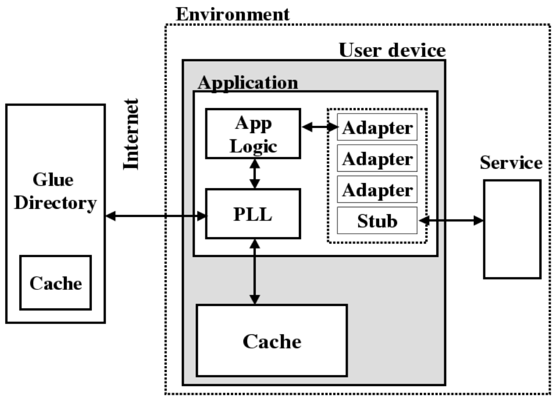

Figure 1. Implementation block diagram of the system.

face and display interface). We do not necessitate that applications only be written to domain-independent interfaces, since this restricts the automation of domain-specific features such as print attributes. On the other hand, domain-independent interfaces can be incorporated into our framework by creating adapters between the domain-specific and domain-independent interfaces.

To summarize, we believe that a mechanism that allows for interoperation between distinct but "approximately" compatible interfaces can be an effective basis for addressing a wide variety of interoperation and composition problems.

4 Implementation

We have implemented a prototype of the proposed framework in 2000 lines of Java code, excluding third-party libraries. Due to lack of space, we only highlight the salient aspects of the implementation here. Figure 1 shows the implementation block diagram. For scalability, the implementation allows the glue directories to be configured into a hierarchy, and low-level glue directories can cache the adapters and stubs obtained from top-level directories, much like DNS servers. PLL refers to the *proxy loader library* – an application library that exports the following helper methods to ease application programmer's task:

- getProxy(sourceInterface,targetInterfaces): Locate and load a proxy. This involves requesting the glue directory for the proxy, checking the signatures on stubs and adapters to verify that they are from trusted providers, and instantiating the component stubs and adapters. PLL also caches frequently used adapters and stubs so that further executions are efficient.
- getProxyRelatedServices(interface): Determine all the proxy-related services for the given interface. Recall the definition of proxy-related services from section 3.2.

4.1 Algorithm for Proxy Construction

There are two non-trivial algorithmic problems that need to be solved by our implementation: constructing proxies and determining proxy-related services.

Our implementation of the proxy construction algorithm using a rule-based system Jess [12]. Interfaces are modeled as facts and adapters are modeled as rules that determine how interfaces can be implemented using other interfaces. As an example, the adapter ${}_IA_{J_1,J2,\ldots,J_n}$ is represented as the rule $J_1, J_2, \ldots, J_n \rightarrow I$. The problem of constructing a proxy then reduces to finding a sequence of rules that derive the source interface fact given that the target interface fact(s). The adapter-stub tree can be constructed from the derivation sequence.

For simplicity, our initial prototype implements a brute force algorithm for determining proxy-related services. Given an interface I, this algorithm finds all services S, such that S exports some interface J and there exists a proxy ${}_IP_J$.[3] The brute force algorithm iterates over every service S in the environment and determines if a proxy ${}_IP_K$ exists for every interface K exported by S. The running time for this algorithm equals the number of services in the environment multiplied by the time taken for the proxy construction algorithm of the previous paragraph. Initial experiments measuring the scalability of this algorithm are reported in the extended version of this paper [20] and indicate that the algorithm scales reasonably, although improvements are desirable.

4.2 Example

The prototype was tested with two concrete examples, one of which is described here: We selected three different web search engines: Google, Hotbot, and AltaVista, and designed programmatic interfaces for each based on their Web user interfaces. (Even though this service is actually a web service, it is easy to imagine a similar search service for meeting/conference rooms that allows searching across all documents and images displayed and exchanged in the meeting room.) Note that these interfaces are fairly involved since they are based on the advanced search pages of these search engines (e.g., `http://www.google.com/advanced_search?hl=en`). The advanced search pages for different search engines support

[3] Actually, this is a restricted definition of proxy-related services that only finds services that can be "reached" by linear chain of adapters. A more general algorithm would also find services that can be reached by adapter trees, and not just by linear chains.

different subsets and variants of various features such as search based on patterns ("all the words", "any of the words", "none of the words", "not the exact phrase"), occurrences (word/phrase occurs in title, body, link, etc), domains/regions (.com, .stanford.edu, "Asia", "Africa", etc), languages, file formats, offensive content filtering, date (pages updated in a given date-range), etc. The actual interfaces are omitted here due to lack of space, but may be found in the extended version [20].

We then wrote an application that displays a Java Swing UI to the user and uses the Google interface to actually perform the search. We also wrote three stubs, one for each search engine, and two adapters, a Google-AltaVista and an AltaVista-Hotbot adapter, and registered them with the glue directory. The application can interoperate with any of Google, Hotbot and AltaVista services on-demand. With the AltaVista service, the application dynamically loads the Google-AltaVista adapter and the AltaVista stub. With the Hotbot service, the application dynamically loads the Google-AltaVista and AltaVista-Hotbot adapters and the Hotbot stub. With the Google service. none of our mechanisms are triggered.

We are currently implementing other examples such as printing, where the interfaces are more complicated than the search example illustrated here, and the initial results are positive.

4.3 Future Work

An important open issue is selecting the "optimal" tree when multiple possibilities exist (as when we can chain A-B and B-C or A-D and D-C when searching for a proxy from A to C.) Criteria for optimality include degree of lossiness and performance. With respect to lossiness, trees containing only lossless adapters are clearly preferable to trees containing one or more lossy adapters, but choosing among multiple lossy trees is non-trivial. Currently, the lossiness information of an adapter can be determined only at runtime – by instantiating the proxy and calling the `isSupported` method. If this information is made available statically, it can be used for selecting better chains from among multiple lossy chains.

5 Related Work

The databases and AI communities have expended much effort [15, 8, 7, 18, 2] on integrating data sources using relational, object-based, semi-structured and XML-based data models for integration. These techniques apply for ubicomp services that can be modeled using declarative data schemas (relational, OEM, ODL, or XML) and associated query languages (conjunctive queries, SQL, Datalog, OQL, XQuery etc), but do not apply to several other ubicomp services where the services only allow access through opaque procedures (as in the UPnP or Jini frameworks). Services may choose to allow only limited procedure-based access for several reasons: they are inherently not data sources, they are not easily modeled using existing data models and query languages, or for encapsulation reasons.

Although adapter and wrapper based techniques have been extensively used for integration purposes in desktop applications and distributed systems, few systems we are aware of dynamically locate and load *interface adapters* directly from network repositories/directories. Jini [1] dynamically locates and loads stubs (not adapters) for interoperation, while operating systems, browsers and media players dynamically locate and load drivers, plug-ins, and codecs, which can also essentially be considered as stubs.

To the best of our knowledge, while chaining of format converters has been studied with some rigor in TOM [17] and data type compatibility based service composition frameworks such as CANS [5] and Paths [14], adapter chaining for procedural interfaces has not been similarly studied in the past. For example, we need a different model of lossiness for interface adapters as opposed to format converters. For interfaces, we need to determine which methods and parameters are preserved, and we have provisioned the recursive `isSupported` mechanism for this purpose.

Distributed object/component frameworks such as CORBA, COM/DCOM and .NET associated Web service standards such as SOAP [23] and WSDL [24] do not (by themselves) provide substitutability. In other words, if two different vendors independently chose different CORBA (or SOAP) interfaces, an application written to one of these interfaces will not interoperate with the other service. Jini [1] and UPnP [22] add unique interfaces per service type to component frameworks to provide for substitutability, but our approach to substitutability is also designed for services without unique interfaces. A related approach in the Web context that provides substitutability is the semantic web initiative [3, 16, 9], where service functionality is marked up with standard domain-specific ontologies allowing agents that understand the ontologies to automate the tasks of invocation and composition. The key difference is that we do not require the development of standardized service-specific ontologies.

The SpeakEasy [4] project also requires the users to select compatible services – but we also provide filtering techniques to automatically identify related services. Also, they do not address the multi-standard service interoperation problem. Instead, they still use single-standard services but advocate that the service standards should be domain-independent. Our framework can accommodate domain-independent interfaces (as noted in section 3.3), but we do not constrain that applications only be written to such interfaces.

6 Conclusions

Considering that new services will continue to appear, and given the nature of standardization processes in the real world, there will always exist service types that do not possess unique standards – services we call multi-standard services. Interoperation for multi-standard services requires solving two subproblems:

- Finding a compatible service from among the services in the local environment.
- Actually invoking the operations on the target service per the interface exported by this service.

We have presented solution approaches for both of these subproblems: For the first subproblem, our approach first narrows down the possible targets from all the services in the environment to a small set of related services, and then relies on the user to select the desired service from among the related services. For the second subproblem, our system dynamically constructs a suitable proxy by assembling the needed stubs and adapters. An important issue that occurs in this context is "lossiness", where an adapter does not implement all the functionality of its source interface. To handle this case, we have provisioned the `isSupported` method, which allows a lossy adapter to indicate the methods and parameters that the adapter does not implement. In conclusion, we have demonstrated that by suitably delegating different parts of the problem to the end-user and the system, a feasible solution approach can be obtained for the problem of ubicomp application-service interoperation for multi-standard services.

References

[1] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison Wesley, 1999.

[2] C. K. Baru et al. XML-Based Information Mediation with MIX. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 597–599. ACM Press, 1999.

[3] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[4] W. K. Edwards, M. W. Newman, J. Sedivy, and T. Smith. Recombinant Computing and the Speakeasy Approach. In *Eighth ACM Conference on Mobile Computing and Networking (MobiCom 2002)*, Atlanta, Georgia, September 2002.

[5] X. Fu et al. CANS: Composable, Adaptive Network Services Infrastructure. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS 01)*, San Francisco, USA, March 2001.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Prentice-Hall, 1995.

[7] Garcia-Molina et al. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[8] A. Halevy. Logic-based techniques in data integration. *Logic Based Artificial Intelligence*, 2000. `http://www.cs.washington.edu/homes/alon/site/files/levy-di00.ps`.

[9] J. Hendler. Agents and the Semantic Web. *IEEE Intelligent Systems (Special Issue on Semantic Web)*, 16(2):30–37, 2001.

[10] T. D. Hodes and R. H. Katz. A Document-based Framework for Internet Application Control. In *2nd USENIX Symposium on Internet Technologies and Systems (USITS 99)*, Boulder, Colorado, USA, October 11-14 1999.

[11] T. D. Hodes, R. H. Katz, E. Servan-Schreiber, and L. Rowe. Composable Ad-hoc Mobile Services for Universal Interaction. In *Third ACM Conference on Mobile Computing and Networking (MobiCom 97)*, Budapest, Hungary, September 1997.

[12] Java Expert System Shell (Jess). `http://herzberg.ca.sandia.gov/jess/`.

[13] Jini Forum. Report on the Fourth Jini Community Meeting. `http://www.javaworld.com/javaone00/j1-00-jinicomm_p.html`.

[14] E. Kıcıman and A. Fox. Using dynamic mediation to integrate COTS entities in a ubiquitous computing environment. In *Handheld and Ubiquitous Computing (HUC 2000), First International Symposium*, Sept. 2000.

[15] A. Y. Levy. Answering Queries using Views: A Survey. `http://www.cs.washington.edu/homes/alon/views.ps`, 1999.

[16] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems (Special Issue on Semantic Web)*, 16(2):46–53, 2001.

[17] J. Ockerbloom. *Mediating among Diverse Data Formats*. PhD thesis, Carnegie Mellon University, January 1999.

[18] Y. Papakonstantinou and V. Vassalos. Query Rewriting for Semistructured Data. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 455–466. ACM Press, 1999.

[19] S. R. Ponnekanti et al. ICrafter: A Service Framework for Ubiquitous Computing Environments. In *UBICOMP 2001*, pages 56–75, 2001.

[20] S. R. Ponnekanti and A. Fox. Application service interoperation without standardized service interfaces. Technical report, Stanford University, Stanford, CA, January 2003. `http://iwork.stanford.edu/pubs/interop-tr.pdf`.

[21] UDDI community. Universal Description, Discovery, and Integration. `http://www.uddi.org`.

[22] UPnP Forum. Universal Plug and Play. `http://www.upnp.org`.

[23] W3C. SOAP specification. `http://www.w3.org/TR/SOAP/`.

[24] W3C. WSDL specification. `http://www.w3.org/TR/WSDL/`.