A Bridging Framework for Universal Interoperability in Pervasive Systems

Jin Nakazawa
Hideyuki Tokuda
Graduate School of Media and Governance
Keio University
5322 Endo Fujisawa, Kanagawa
252-8520 JAPAN
{*jin,hxt*}@ht.sfc.keio.ac.jp

W. Keith Edwards
Umakishore Ramachandran
College of Computing
Georgia Institute of Technology
801 Atlantic Drive NW, Atlanta, GA
30332-0280 USA
{*keith,rama*}@cc.gatech.edu

Abstract

We explore the design patterns and architectural trade-offs for achieving interoperability across communication middleware platforms, and describe uMiddle, a bridging framework for universal interoperability that enables seamless device interaction over diverse platforms. The proliferation of middleware platforms that cater to specific devices has created isolated islands of devices with no uniform protocol for interoperability across these islands. This void makes it difficult to rapidly prototype pervasive computing applications spanning a wide variety of devices. We discuss the design space of architectural solutions that can address this void, and detail the trade-offs that must be faced when trying to achieve cross-platform interoperability. uMiddle is a framework for achieving such interoperability, and serves as a powerful platform for creating applications that are independent of specific underlying communication platforms.

1 Introduction

Recent years have seen a rapidly increasing array of specialized computing devices, ranging from small network-aware gadgets such as GPS receivers and cameras to traditional general-purpose computers. Paralleling the proliferation of these devices has been the development of a range of communication middleware platforms, such as Universal Plug'n'Play (UPnP) [12], Bluetooth [1], and Jini [16]. Each platform supports the ability of devices to interoperate with and use other devices built using the same platform.

From the perspective of the developers of pervasive computing applications, one significant problem with this range of middleware solutions is that they are virtually incompatible with one another. Thus devices and services based on one platform cannot easily use those built on a different platform. For example, two devices built on Bluetooth and UPnP, respectively, would be unable to interact with each other, despite the fact that they may use semantically similar application profiles such as *Basic Imaging Profile* (BIP) in Bluetooth and *MediaRenderer* profile in UPnP. The result is isolated "islands of interoperability," in which users and developers may not actually be able to use the full range of devices available to them.

Is it possible to bridge these different platforms allowing interoperation across them in a way that is robust and extensible, and yet retaining the power of the individual platforms? Achieving such goals would open up a huge range of application possibilities. For example, many pervasive computing applications (such as [6] [11] [8] [3] [9]) could benefit from the ability to seamlessly use devices from diverse platforms. Users of such applications gain more flexibility by having a greater range of devices available to them, and researchers investigating novel applications are more free to develop and deploy their results using the most practical devices available.

We start by exploring design patterns and architectural trade-offs that arise when trying to achieve interoperability across communication middleware platforms. We then present uMiddle, a bridging framework for universal interoperability that sits at a compelling point in that design space. uMiddle is a vehicle for exploring and validating architectural approaches to interoperability, and also serves as a powerful platform for creating applications that are independent of specific underlying platforms.

The rest of the paper is organized as follows. Section 2 explores the design space. Section 3 shows the design and implementation of uMiddle. Section 4 explores the capabilities of uMiddle by looking at a range of applications built with it. Section 5 benchmarks the implementation, and Section 6 positions our approaches with regard to related work. Finally, we present our concluding remarks in Section 7.

2 Design Patterns for Interoperability

This section discusses the requirements and the design space of bridging diverse middleware platforms.

2.1 Requirements

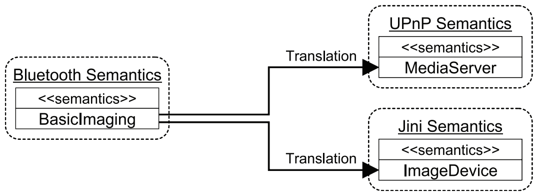
To accommodate interoperability across diverse middleware platforms, a bridging framework should support the following capabilities.

(1) Transport-level Bridging: Transport-level bridging involves translation of protocols and data types inherent in the platforms to be bridged. Each platform uses its own base protocol for device communication, such as SOAP[2] in UPnP. Using these protocols, devices exchange data formatted in their platform-specific types, for example Jini services communicate using Java objects. The bridging framework must be able to translate between these different protocols and data representations to enable devices on different platforms to communicate with one another.

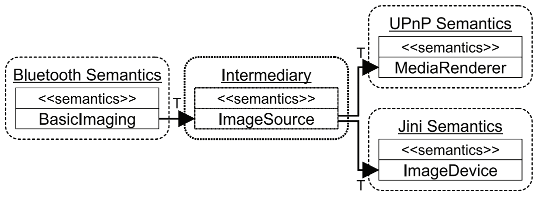
(2) Service-level Bridging: If a new device appears in one platform, the bridging framework must be able to dynamically recognize its presence and make it available for use with other devices. Different platforms utilize different discovery mechanisms, for example UPnP uses Simple Service Discovery Protocol (SSDP), while Bluetooth uses Service Discovery Protocol (SDP). The bridging framework must bridge the various discovery mechanisms used by the individual communication platforms.

(3) Device-level Bridging: This involves translating the device semantics, such as roles of devices and their compatibility, across the different platforms. They are represented differently in different platforms. For example, while UPnP provides *device types* that encapsulate states, events, and actions supported by the devices, Bluetooth exploits *device profiles* defining their own protocols over the Bluetooth base protocol. The bridging framework must be able to translate these different representations of device semantics to enable devices on different platforms to understand one another.

(4) Future Evolution: To stay current with evolving standards and technologies, the framework must be able to accommodate new device types of an already supported platform. For example, if the UPnP standard introduces a new device type, the framework must be extensible to establish a device-level bridge for this new type. Further, the framework must be extensible to accommodate entirely new communication platforms with new service-level and transport-level bridges for those platforms. Together, these four traits describe what we term to be a *universal interoperability infrastructure*, meaning one that can adapt to the presence of new devices, new device types, and new platforms.



(1-a) Direct Translation



(1-b) Mediated Translation

Figure 1. Translation Models

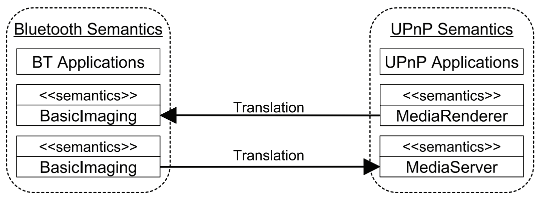
2.2 Design Patterns

The most important architectural considerations to achieve the above requirements can be captured by four dimensions. A point along each of these dimensions embodies its own trade-offs. This section explores these four dimensions, and the pros and cons associated with each.

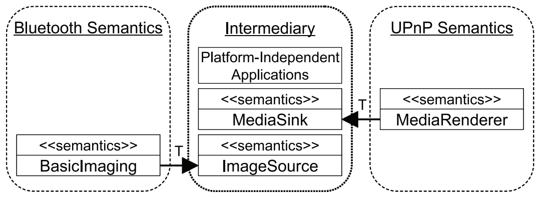
2.2.1 Translation Model

One key dimension concerns how the semantics of devices on disparate platforms are translated. For example, in the case of bridging a Bluetooth BIP device to a UPnP MediaRenderer device, a reasonable translation would be that it "makes sense" conceptually for images from the BIP device to serve as a source for display on the MediaRenderer. The translation mechanism is the conceptual basis for accommodating such operations.

One approach is a **direct** translation (Figure 1-a), in which the semantics of a device on one platform is directly translated to that of another platform. In the BIP-MediaRenderer case, a direct translator would be a mechanism that (1) can convert between the Bluetooth and UPnP communication protocols (transport-level), (2) can exchange device registrations between their registry services (service-level), and (3) can map the concepts and operations in BIP into the UPnP MediaServer device type (device-level). This approach benefits from minimized semantic loss in translations, because there is a specific translator for every device type pair that can accommodate all the nuances of the types it is bridging. However, it does not scale well. Any new device type requires a new translator for each existing device type (`n(n-1)` translators for `n` total device types). As supported device types increase, the

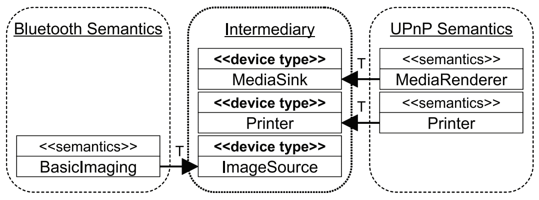


(2-a) Scattered Proxies

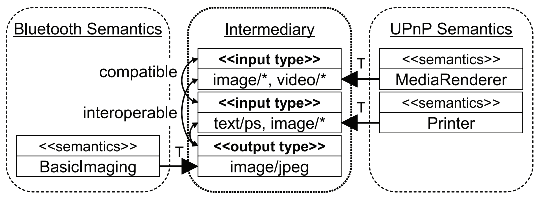


(2-b) Aggregated Proxies

Figure 2. Semantic Distributions



(3-a) Coarse-grained Representation



(3-b) Fine-grained Representation

Figure 3. Semantics Granularity

number of required translators becomes very large.

An alternative approach is a **mediated** translation (Figure 1-b). It first translates from the semantics of a device to common intermediary representations, and then, optionally, from those common representations out to different native semantics (the next section discusses why the second translation is optional). A mediated translator in the BIP-MediaRenderer case would be a mechanism that (1) can translate between the native and common protocols, (2) can merge device registrations in both Bluetooth and UPnP into an intermediary registry service, which makes them discoverable in the intermediary semantic space, and (3) can map the concepts and operations of both BIP and MediaRenderer into the intermediary representation. This approach may cause certain original device semantics to be lost in translation, since the common representation must be platform neutral, and may not be able to cope with the subtleties of a given new platform. The advantage of this approach is that it is scalable requiring at most one translator per device type (to translate between that type and the common format).

2.2.2 Semantic Distribution

The second key architectural dimension concerns whether devices in a given platform are *visible* and usable by applications built on top of a different platform (applications that are native to a particular platform).

One approach is to **scatter** (Figure 2-a) proxy representations of a device from one platform to other platforms, thereby making the devices visible and usable from any native platform. For example, a Bluetooth BIP device may be represented as a MediaServer device to UPnP devices. Scattered visibility is implied by the direct translation approach (1-a), since every direct translation creates a native representation on another platform after translating the original device. It may be combined with mediated translation as well. In this case, the BIP device is first translated to the common semantic space, and then out to the UPnP platform. The advantage of this approach is that it allows native applications to use devices from different platforms without modification. A native UPnP application can compose a MediaRenderer device with a BIP device (which would be represented as a proxy UPnP MediaServer device).

An alternative approach is to **aggregate** (Figure 2-b) the proxy representations of devices on different platforms in the intermediary semantic space. In this approach, translation is done solely to the intermediary semantic space. Since the device representations are aggregated and visible only in the intermediary semantic space, native applications (for example UPnP applications) cannot use the devices from the other peer platforms. However, applications built on top of the intermediary semantic space can use all the devices from the various other platforms. Further, this approach lends itself to portability across different smart spaces since applications built on top of the intermediary semantic space do not contain any platform dependencies.

2.2.3 Intermediary Semantics Granularity

The next architectural dimension, which is specific to mediated translation, is how native devices are represented in the intermediary semantic space. The representation determines the compatibility of any two devices. One approach is **coarse-grained** (Figure 3-a) representation that uses device types encapsulating all the operations and semantics of devices. Any two given devices are compatible if their coarse-grained representation types are the same. This is similar to the device profiles of UPnP or Bluetooth, which

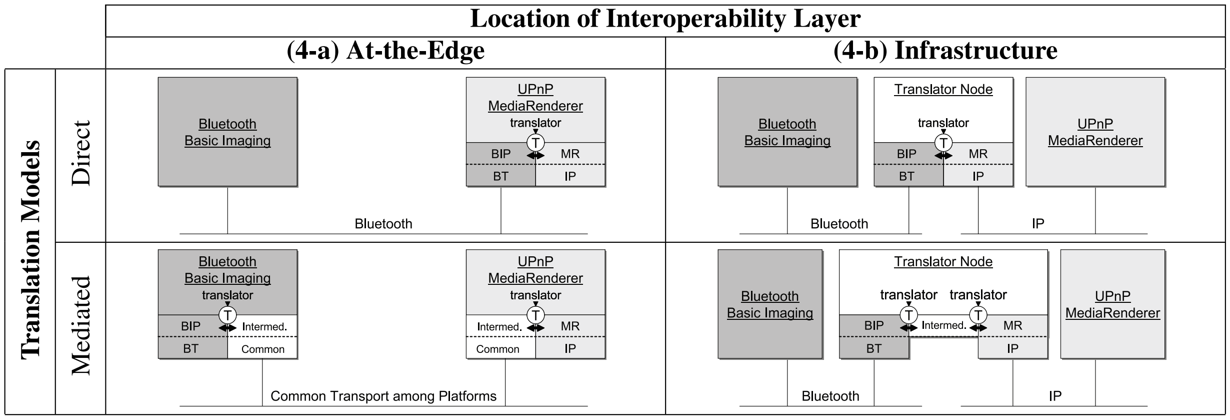


Figure 4. Location of Interoperability Layer

encapsulate high-level aggregations of functionality. Its advantage is that applications can simply match devices with requests of users via predefined type names. If a user requests an application to print a document, the application uses a device of "printer" type.

This approach requires an ontology of possible device types in the common representation space. This requirement imposes a number of disadvantages. First, applications can only use currently defined device types. Since the common semantic space is required to represent devices from various platforms each defining new device types occasionally, the common type set could expand rapidly. It is impractical to expect applications to be updated whenever new device types are defined. Second, it narrows the range of devices with which the applications can interact. In this approach, any two devices are incompatible when their types are different even if they might be "partially compatible" conceptually. For example, the *MediaRenderer* and a *Printer* types in UPnP are completely different, requiring applications to be implemented specifically to communicate with each type, even though both accept and render content.

An alternative approach is **fine-grained** representation (Figure 3-b) that narrows the granularity of compatibility. It breaks down the original complex device semantics into a set of communication endpoints and associates a data type with each endpoint. Any two devices are compatible if they contain either input or output endpoints where the same data types are associated. Since new data types are less-frequently defined than device types, this approach can allow applications to cope with greater range of devices without frequent modification. However, because the device interfaces themselves no longer encode the specific role of the device (that it is a printer, for example), this approach requires some augmentation to enable an application to specify the role of a target device with which it wants to interact.

2.2.4 Location of Interoperability Layer

The final dimension concerns the location of translators. This dimension only concerns *where* translation happens at runtime, and is distinct from previously described issues of whether intermediary representations are used, and so forth. Figure 4 shows possible locations in relation to the translation models discussed in Section 2.2.1.

One approach is **at-the-edge** translations (Figure 4-a), in which each device somehow acquires the ability to translate their own semantics for other platforms. A given device would be augmented with specialized knowledge of each peer, allowing it to communicate with peers using their native protocols. This choice allows for direct communication without the need for an intermediary; however, it is impractical in many situations. For example, although Speakeasy[4] supports at-the-edge mediated topology through mobile code, this capability necessitates extra facilities on each participating device (special protocols to exchange mobile code, and a runtime environment for executing it). Even with such facilities, however, an at-the-edge choice cannot support communication between devices over different physical transports, since it is impractical to expect devices in a platform like a UPnP MediaRenderer TV to have physical transports specific to other platforms such as Bluetooth, or vice versa.

An alternative approach is translation in **infrastructure** (Figure 4-b), in which an intermediary node on the network performs the translation. It requires no changes to the devices themselves, or the presence of any special facilities. It can also allow the bridging of different physical transports as long as the intermediary can use the necessary transports.

2.3 Mutual Compatibility

Certain design choices from the previous subsections are mutually dependent on one another. Table 1 shows com-

Table 1. Mutual Compatibility Chart

		1		2		3		4	
		a	b	a	b	a	b	a	b
1	a	-	-	O	-	-	-	O	O
	b	-	-	O	O	O	O	O	O
2	a	O	O	-	-	O	O	O	O
	b	-	O	-	-	O	O	O	O
3	a	-	O	O	O	-	-	O	O
	b	-	O	O	O	-	-	O	O
4	a	O	O	O	O	O	O	-	-
	b	O	O	O	O	O	O	-	-

patibility between the approaches, where O means given two approaches can coexist, while − means they cannot. In particular, the approaches of aggregated visibility (2-b), coarse-grained representation (3-a), and fine-grained representation (3-b) are specific to the mediated translation (1-b); hence they cannot coexist with the direct translation (1-a) in one design. In other words, when taking the direct translation approach (the first row), the only design choice is between at-the-edge (4-a) and in the infrastructure (4-b). The aggregated visibility (2-b) approach is incompatible with the direct translation (1-a). The coarse-grained representation (3-a) and fined-grained representation (3-b) is specific to the intermediary semantic space, hence is incompatible with the direct translation (1-a).

3 uMiddle: A System for Universal Interoperability

In this section we describe uMiddle, a bridging framework that enables seamless device interaction over diverse middleware platforms. Our goal for uMiddle is to provide a universal and extensible bridging middleware that also enables platform-independent application development without requiring changes to existing devices.

3.1 uMiddle Design Approach

Based on its goals of universality and extensibility, uMiddle embodies the following design choices with respect to the architectural trade-offs presented in Section 2.

Mediated translation (1-b): This choice lowers the barrier to enabling support for new device types and new platforms and thus achieves extensibility. With mediated translation, each new device requires at most one translator to support it, rather than `n-1` direct translations.

Aggregated visibility (2-b): This choice allows applications to be used in arbitrary smart spaces taking advantage of any and all platforms that exist in the environment. Applications built atop the intermediary semantic space are platform-independent, thereby achieving such universality.

Fine-grained representation (3-b): In terms of universality, applications should also be able to take advantage of a wide range of devices including the definition of new device types. The choice of fine-grained representation ensures this ability without relying on the existence of an overarching device ontology.

In-the-Infrastructure (4-b): Universality also requires uMiddle to bridge over different physical transports. The choice of locating the interoperability layer in the infrastructure enables bridging across different physical transports without the need to change the native devices.

Each point in the design space entails not only certain advantages, but also a set of challenges that must be met. In our case, such challenges are (1) minimizing the semantic loss caused by mediated translation and (2) enabling applications to specify device roles despite a fined-grained device representation. We address these challenges by three major features. We first provide an overview of the uMiddle system followed by a description of these features.

3.2 System Overview

uMiddle is a universal interoperability system implemented in Java. Figure 5 shows its architecture with a Bluetooth digital camera and a UPnP MediaRenderer TV as example devices that are bridged. We call such devices bridged from communication platforms "native devices."

uMiddle realizes interoperability via two abstractions called *mappers* and *translators*. A mapper establishes *service-level* and *transport-level* bridges discussed in Section 2.1. Figure 5 contains mappers for Bluetooth and UPnP. It discovers a native device via a platform-specific discovery protocol, and imports it into the intermediary semantic space by instantiating the device-specific translator. It also contains a base-protocol support for the target platform, such as the base Bluetooth protocols or SOAP in the case of UPnP.

A *translator* works as a *device-level* bridge for a native device. First, it projects the device-specific semantics into the intermediary semantic space. For example, the *BIP Translator* in Figure 5 translates BIP-specific semantics into the common representation. Second, it acts as a proxy for that device, causing any connections to the translator to trigger actual connections to the native device. Therefore, the BIP camera device transmits images through its translator to destination devices. Finally, a translator embodies any protocol and semantics that are native to the particular device that is associated with it. Thus with reference to Figure 5, the *BIP Translator* implements the OBEX protocol using the base-protocol support provided by the Bluetooth mapper. In other words, the platform-specific knowledge of a

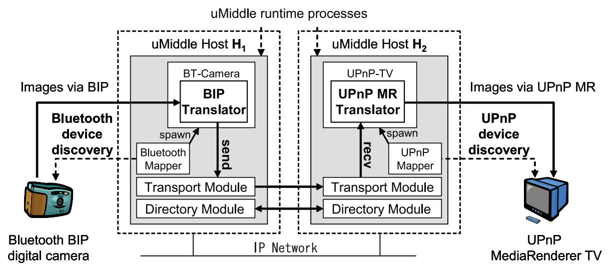


Figure 5. System Architecture

device is concealed by its translator and the mapper, and the rest of the system is platform-independent.

Multiple hosts on a network may run the uMiddle runtime. Devices directly connected to these hosts, or discoverable by them, can be freely used with devices known to other uMiddle runtimes. To support this functionality, the uMiddle *directory module* handles the exchange of device advertisements among hosts. This provides a discovery mechanism that allows notification about the presence of devices, across uMiddle runtimes, independent of the actual discovery protocols used by particular devices. Further, the uMiddle *transport module* serves to allow communication among translators situated in different nodes. In Figure 5, two translators for a Bluetooth BIP digital camera and a UPnP MediaRenderer TV, that are respectively hosted by intermediary translator nodes H_1 and H_2, communicate through the transport modules on their respective hosts.

Currently, uMiddle can bridge a range of platforms, including UPnP and Bluetooth devices, the Berkeley Motes platform, MediaBroker[13], Java RMI, and various web services. uMiddle is extensible along two dimensions to accommodate new standards and technologies. First, a new device type in a known platform can be incorporated into uMiddle by simply writing a translator for that device. Second, a new communication platform can be incorporated into uMiddle by writing a mapper specific to that platform along with a set of associated translators.

3.3 Service Shaping

The challenge with a fine-grained representation is to provide a facility that empowers applications to specify the role of devices with which they want to interact. We address this challenge through a technique that we call *Service Shaping*[14]. In the service shaping technique, the semantics of a native device are represented as a set of communication endpoints, called *ports*. uMiddle defines two types of ports *digital* and *physical* that define the capabilities of a device. Digital ports are used for communication between devices, while physical ports describe the user-perceptible

(1) *Collection lookup(Query query)* — Gets profiles of translators that match the given query.

(2) *void addDirectoryListener(DirectoryListener l)* — Registers a component to receive a notification when a new native device is mapped to uMiddle.

Figure 6. Directory API

effects of a device in the physical world, and are used to generically define the roles various devices may play.

A digital port is an endpoint owned by a translator, which transmits digital information to and from the network. For example, the BIP translator in Figure 5 would contain a digital port for image output. Each digital port is tagged with a MIME-type, and uMiddle applications can check interoperability of any two translators simply by comparing MIME-types of digital ports owned by those translators. A physical port is a conceptual entity that causes or senses some perceptible change in the physical world, and it is tagged with a combination of a *perception type* and a *media type*. The former represents how users can perceive the change, and can be one of *visible*, *audible*, and *tangible*. The latter represents the physical media that carries the information. We call this information associated with ports of a given translator the "shape," since this information represents the affordances of the device with which the translator is attached.

Consider, for example, a translator for a PostScript printer. It would contain a "text/ps" digital input port and a "visible/paper" physical output port. In this case, "visible" is a perception type, and "paper" is a media type. From an application's point of view, the physical data type can be used to specify the affordance of devices needed by the application. If a user wishes to *view* a document in one way or another, the application can select a device with an input port of the document's MIME-type and physical output port of "visible/*". If the user wants to print it, the application specifies "visible/paper". Programmers of uMiddle applications can discover the translators that match a given shape, using APIs listed in Figure 6.

3.4 Universal Service Description

We created a new XML-based language, called **Universal Service Description Language (USDL)** that is used to support the representation of semantics of native devices in uMiddle's intermediary semantic space for both humans and machines. A mapper creates a translator (and the shape) of a native device based on a USDL definition for that device. For example, a USDL document for UPnP light devices describes how to represent UPnP-specific actions, such as *SetPower*, in uMiddle. The *SetPower* action is specified to switch on a light when it gets "1" as a parameter. Therefore, the USDL document defines two digital input

(1) *void connect(OutputPort src, InputPort dst)* — Establishes a communication path between specific input and output ports.

(2) *void connect(Port src, Query dst)* — Establishes a dynamic message path between a specific port owned by a translator and the ports matching a given query.

Figure 7. Transport API

ports to the translator corresponding to the light device; one is to switch on passing "1" to the native UPnP light, and the other is to switch off passing "0" to it.

USDL documents describe how mappers configure translators for specific devices given a generic translator implementation. For example, since notions such as states, actions, and events are common abstractions in every UPnP service, it is possible to create a generic translator for the UPnP platform which is then mechanically parameterized for any given UPnP device by a USDL document describing that device. Similarly, any Bluetooth BIP device defines image transmission capability, but its role (such as camera or printer) can be determined at runtime. Thus, we can provide a generic Bluetooth BIP translator implementation which is parameterized for these different specific types of devices based on different USDL documents. Therefore the implementation of translators can be generic, assuming such a document-based runtime configuration.

3.5 Dynamic Device Binding

This mechanism enables applications to establish connections between translators based on their shapes rather than their specific identity. uMiddle applications are allowed to establish a communication path between translators using the transport APIs listed in Figure 7. Such a path exists in Figure 5 between the BIP and UPnP MR translators. The applications can connect a digital port with peers by specifying either a specific port instance (Figure 7-(1)) or a generic *template shape* as a query object. Since native devices are mapped and unmapped in uMiddle dynamically, such a template-based connection is important in our framework. When an application creates a dynamic message path, the uMiddle runtime hosting the source port evaluates the given template shape adaptively to the presence of translators in a network. If a translator matching the template appears in the network, a new message path to the translator is established being aware of the data type; it is bound to the port owned by the target translator, whose data type is equivalent to the source port.

The dynamic template-based type-aware connection enables a fine-grained device polymorphism in the common semantic space. A device can be connected to others with different roles and implementations through their translators. For instance, the *BIP Translator* in Figure 5 can be connected to a player device, a storage device, and others if their MIME-types match by issuing only one template-based connection request. Once a connection is made, the sender can interact with the receiver being unaware of its behavior. On the other hand, if programmatic types were to be used for interoperability, such dynamic compositions are possible only if the devices are fully statically defined to the others. This situation is not practical to expect as described in Section 2.2.3. In uMiddle, since we narrowed the granularity of compatibility to data types, a device can directly interact with wider range of other devices.

3.6 System Characteristics

Our design choices result in the following system characteristics. First, uMiddle does not allow applications built on native platforms to access devices on other platforms. Instead, based on our primary goal to enable platform-independent application development, uMiddle allows applications built on the common semantic space to have universal interoperability. Second, the choice of infrastructural translation enables flexible deployment, because mappers in our framework can be distributed over a network. If the framework is used in a small area, such as in a room, the user can co-locate mappers for different platforms in one computer. If it is used to cover a larger area, such as a house or a university campus, mappers can be located in different rooms based on the specifics of the environment. In a room where only Bluetooth devices are used, an intermediary translation node would be configured with the Bluetooth mapper. In another room where Bluetooth, UPnP, and other devices are used, an intermediary node would host mappers for those various platforms. These intermediary nodes communicate with one another through the directory and transport modules in our framework to form the common intermediary semantic space.

4 Experiences with uMiddle

This section presents our experience with uMiddle by describing two applications: one is event and control-oriented, and the other is multimedia-oriented.

4.1 uMiddle Pads

Pads is a GUI-based application generator that allows device composition across different platforms. It provides cross-platform "virtual cabling," in which the user need not care whether the devices being interconnected are based on Bluetooth, UPnP, or other platforms. Figure 8 shows a screen shot of the Pads application containing translators for twenty-two devices, including one Bluetooth and three UPnP devices. The other eighteen are native uMiddle

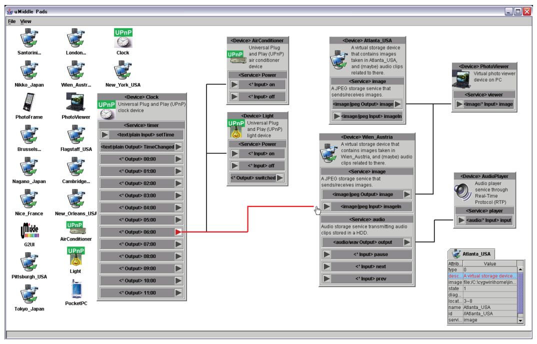


Figure 8. Screen Shot of uMiddle Pads

Figure 9. G^{2}UI Atlas Application

devices, by which we mean services built directly against uMiddle as their native middleware platform. Its key functionalities include: (1) a visual representation of the intermediary semantic space of a particular uMiddle environment by showing the translators as icons, (2) a simple hotwiring capability that allows an application developer to establish device connections by drawing lines between the translators shown in the GUI, and (3) a runtime environment on top of uMiddle backing the GUI causing end-to-end device communications to be established across the different platforms. While such a cross-platform application development usually requires intensive knowledge about the environment and the platforms therein, uMiddle's platform-independent semantic space coupled with this tool lowers the barrier for application development, and makes it as low as drawing lines on a GUI.

4.2 G^{2}UI: Geographical User Interface

G^{2}UI is a real-world user interface toolkit built with uMiddle to improvise intuitive and adaptive multimedia applications using pervasive off-the-shelf devices. The key ideas of G^{2}UI are that (1) gadgets, including media storage, player, and capture devices, can be registered to be "located" at certain regions in a geographical coordinate system; and (2) co-location of devices within the coordinate space triggers *geoplay*, which involves playback of media acquired from one or more co-located storage or capture devices, or *geostore*, where a storage device stores data received from a co-located capture device. Since this application is built on top of the common semantic space, the above operations work across diverse communication platforms. For example, if a user co-locates a Bluetooth digital camera and a UPnP MediaRenderer TV, the images in the camera would serve as the source for the TV via a uMiddle's dynamic message path. Universality and extensibility of uMiddle ensure that the co-located services will work with future evolutions of device types and platforms.

5 Benchmarks

This section evaluates the systems performance for translations. The benchmarks are conducted using (1) IBM ThinkPad T42p (Pentium M 2.0GHz, 2GB RAM, Windows XP Service Pack 2) (UPnP mapper), (2) IBM ThinkPad T42p (Pentium M 2.0GHz, 2GB RAM, Fedora Core 3 Linux) (Bluetooth Mapper), and (3) IBM ThinkPad T42p (Pentium M 2.0GHz, 2GB RAM, Fedora Core 3 Linux) connected by a 10Mbps Ethernet hub.

5.1 Service-level Bridging

This section evaluates performance of service-level bridging. In particular, we benchmark UPnP and Bluetooth mappers to show the lower-bound of the system's performance to cope with device presence in pervasive environments. The experiment illustrates the time needed by the uMiddle mapper to dynamically generate translators for devices after they are discovered in their native platforms. We use the Linux BlueZ library to create the Bluetooth mapper, and the CyberLink Java library for the UPnP mapper.

Figure 10 shows the performance of the mapping operation for UPnP and Bluetooth. The mapping overhead depends on the complexity of the translators. In this benchmark, the translator for a UPnP clock device contains fourteen ports and two more uMiddle entities for the UPnP service/device hierarchy, and takes more than 1.4 seconds. This means this particular configuration of the UPnP clock translator can be generated at approximately 0.7 instances per second. The instantiation of the translators for the other UPnP devices (air conditioner and light) is much faster yielding a performance of approximately four instances per second. Since UPnP devices are mostly immobile connected to an IP network, we believe that the performance of the current UPnP mapper implementation is adequate.

As can be seen from Figure 9, the instantiation of Bluetooth device translators is quite good (roughly 5 instantia-

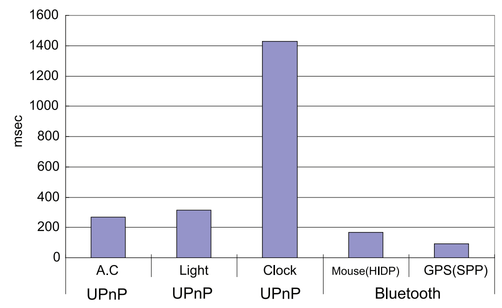


Figure 10. Service-Level Bridging

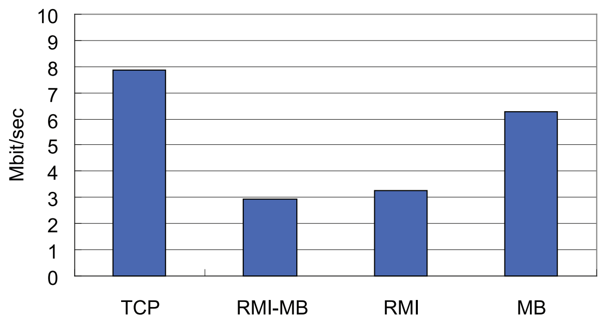


Figure 11. Transport-level Bridging

tions per second for a HIDP mouse). Considering the locality of a Bluetooth network (at most eight devices in one piconet covering a few tens of meters), the mapping performance is adequate to cope with the moderate device mobility expected in environments such as a classroom.

5.2 Device-level Bridging

This section evaluates performance of device-level bridging for a Bluetooth mouse device and an emulated UPnP light switch device included in the CyberLink library. Device-level translation is done on top of the base protocol to translate device-specific protocols and data representations. In the UPnP case, the benchmark involves the translator for the light switch device, receiving UPnP *actions* a hundred times from a uMiddle application. The Bluetooth case involves the translator for the mouse device receiving mouse click signals a hundred times from the mouse and then sending them out to another uMiddle device.

The average time to control the UPnP light switch is 160 milliseconds. This includes 150 milliseconds consumed in the UPnP domain (marshaling/unmarshaling XML messages and controlling the light switch), and the rest in uMiddle (translating a control request dispatched by the application to a UPnP *action* object). In the Bluetooth case, the average overhead is 23 milliseconds that involves translating the mouse signal to a Vector Markup Language document as the common representation, and passes it to the uMiddle's transport module. These results show that the infrastructure itself contributes little to the performance overhead.

5.3 Transport-level Bridging

This section evaluates performance of transport-level bridging using a Java RMI mapper and a MediaBroker (MB)[13] mapper to show the system's bottleneck. The MB system is a distributed media transformation infrastructure developed at Georgia Tech. The benchmark uses a Java RMI service on node (3) and an MB service on node (1). Node (2) runs a uMiddle runtime containing TCP/IP-based transport module. Node (1) and (3) host an MB server and a Java RMI registry program, respectively.

In this network, we conduct the following three tests. First, the MB service sends 1400-bytes messages to its translator on node (2), and the messages are echoed back to the same service (MB test). Second, similarly to the first, the Java RMI service sends 1400-bytes messages to itself through uMiddle (RMI test). These tests illustrate the baseline throughput of the Java RMI and MB service through uMiddle. Third, the same MB service sends the messages to the Java RMI service through uMiddle (RMI-MB test). This shows the overhead of the transport-level bridging.

Figure 11 shows the performance of these. With 7.9Mbps TCP baseline throughput, these test marked 2.9Mbps (RMI-MB test), 3.2Mbps (RMI test), and 6.2Mbps (MB test). This means that the process of transport-level bridging, requiring marshaling and unmarshaling of data encapsulated in platform-specific data packets, causes additional end-to-end communication delay. In addition, if one of the services uses narrower bandwidth network, such as Java RMI service in this case or Bluetooth services, the service would be a bottleneck that causes the data sent from other services to accumulate in the uMiddle's translation buffer. Therefore, the universal interoperability layer should provide some QoS control mechanism.

6 Related Work

Systems aimed at achieving platform interoperability include Universally Interoperable Core (UIC)[15] and Speakeasy [4]. UIC provides a component-based middleware framework, which can bridge multiple communication platforms. Speakeasy allows arbitrary computational entities to interact with one another. Interestingly, from our point of view, these two take the same design choices, which are mediated translation (1-b), aggregated visibility (2-b), coarse-grained (3-a), and at-the-edge (4-a). However, they require that devices be modified to enable an at-the-edge translation; for example in Speakeasy, by adding the common platform for mobile code execution. Our approach does not require the device modification, since we locate the interoperability layer in the infrastructure.

In UIC, devices, such as user-side handheld terminals, must have knowledge to communicate with devices on other platforms. Though UIC allows specialization of the handheld devices with such knowledge, we cannot expect such specialization to happen on a device-by-device basis each time a new communication mechanism appears. In uMiddle, such specialization is only required in the intermediary nodes in the infrastructure, and not in application nodes. By providing a generic object interface, UIC's implementation tries to narrow the intermediary semantics; however, since UIC allows devices to export arbitrary interfaces in addition to the generic one, which brings with it the downside of the coarse-grained representation.

Several systems exploit user interface migration to allow users to access devices easily. Kangas *et al.* presents a system architecture [10] for controlling ubiquitous embedded devices with a migratory object approach. Munson *et al.* propose an architecture to control information appliances with an XML-based user interface markup language [5]. Hodes and Katz' Document-Based Framework [7] also proposes an XML-based language that enables user interface-based device compositions. With such systems, users are allowed to use devices without considering their platform difference; however, they cannot use them as a means to achieve device-to-device interoperability.

7 Conclusions

We have discussed design patterns and architectural trade-offs that arise when trying to achieve interoperability across communication middleware platforms, and described uMiddle: a system for universal interoperability. uMiddle enables platform-independent application development without requiring changes to existing devices. It sits at a particular point in the space of the architectural trade-offs, and we have addressed the challenges in our design choices with three unique features. They are the Service Shaping approach for common device representation to maximize device interoperability, an XML-based language called USDL for dynamic translator generation, and a dynamic device binding mechanism to provide a device-level polymorphism based on the device shapes. We have shown that the current implementation achieves platform interoperability with minimal overhead, and have also described how a number of applications can be implemented using it. The major future work is QoS control in the service-level bridge. Since different platforms entail different QoS semantics, the service-level bridge needs to translate between such different semantics.

References

[1] Bluetooth Consortium. Specification of the Bluetooth System, Version 1.2, Core, Nov. 2003.

[2] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (soap) 1.1, May 2000. W3C Note.

[3] B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer. Easyliving: Technologies for intelligent environments. In *Second International Symposium on Handheld and Ubiquitous Computing, HUC 2000*, pages 12–29, Sept. 2000.

[4] W. K. Edwards, M. W. Newman, J. Z. Sedivy, T. F. Smith, and S. Izadi. Challenge: Recombinant computing and the speakeasy approach. In *Proceedings of the Eighth ACM International Conference on Mobile Computing and Networking (MobiCom 2002)*, Sept. 2002.

[5] K. F. Eustice, T. J. Lehman, A. M. G., M. C. Muson, S. Edlund, and M. G. G. A universal information appliance. In *IBM Systems Journal, 38(4)*, pages 575–601, 1999.

[6] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The anatomy of a context-aware application. *Wireless Networks*, 8(2/3):187–197, 2002.

[7] T. D. Hodes and R. H. Katz. A document-based framework for internet application control. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems (USITS '99)*, pages 59–70, Oct. 1999.

[8] S. S. Instille. Designing a home of the future. In *IEEE Pervasive Computing Magazine 1(2)*, pages 80–86, Apr. 2002.

[9] B. Johanson, A. Fox, and T. Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. In *IEEE Pervasive Computing Magazine 1(2)*, pages 71–78, 2002.

[10] K. Kangas and J. Röning. Using Mobile Code for Service Integration in Ubiquitous Computing. In *Proceedings of the 5th Mobile Object Systems Workshop*, June 1999.

[11] C. D. Kidd, R. Orr, G. D. Abowd, C. G. Atkeson, I. A. Essa, B. MacIntyre, E. Mynatt, T. E. Starner, and W. Newsletter. The aware home: A living laboratory for ubiquitous computing research. In *Proceedings of the Second International Workshop on Cooperative Buildings - CoBuild'99*, Oct. 1999.

[12] Microsoft, Corp. Universal plug and play device architecture reference specification, 1999.

[13] M. Modahl, I. Bagrak, M. Wolenetz, P. Hutto, and U. Ramachandran. Mediabroker: An architecture for pervasive computing. In *IEEE PerCom 2004*, pages 253–276, Mar. 2004.

[14] J. Nakazawa, J. Yura, and H. Tokuda. A service shaping approach for task-based computing middleware. In *International Workshop on Computer Support for Human Tasks and Activities*, Apr. 2004.

[15] M. Roman, F. Kon, and R. H. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online, Special Issue on Reflective Middleware*, July 2001.

[16] Sun Microsystems, Inc. Jini Architecture Specification, Nov. 1998. http:// www.javasoft.com/ products/ jini/ specs/ jini-spec.pdf.