uMiddle: A Universal Framework for Bridging Diverse Middleware Platforms

Authors

Institutes

Abstract. We present uMiddle, a universal framework for smart space applications designed to enable seamless device interaction over diverse middleware platforms. The rapidly expanding variety of specialized computing devices has been enabling useful ubiquitous computing applications. Unfortunately, there has been a proliferation of middleware platforms that cater to specific devices thus creating islands of devices with no uniform protocol for interoperability and extensibility across these islands. This void makes it difficult to rapidly prototype ubiquitous computing applications spanning a wide variety of devices and dynamically composed services. uMiddle addresses this void by bridging the growing number of middleware platforms in existence, and by allowing interoperability and use across the platforms. Key features of uMiddle are a multi-semantics communication library for flexible data transmission, an extensible system for incorporating entities using existing middleware platforms, and a configurable protocol translator. We discuss the design and implementation of the framework, and the ease of developing applications using this framework. Benefits of the framework include its ability to integrate devices from different middleware families with diverse communication technologies, and its ability to support rapid construction of applications that use a wide range of devices and media types.

1 Introduction

Recent years have seen a rapidly increasing array of specialized computing devices. These range from small network-aware gadgets such as network-connected GPS receivers and camera units to traditional general-purpose desktop, laptop, or tablet computers. Paralleling the proliferation of these devices has been the development of a range of middleware communication platforms, from both industry and academia. Each platform supports the ability of devices to interoperate with and use other devices built using the same platform. Examples of these platforms include Universal Plug'n'Play (UPnP) [17], Bluetooth [2], Jini [25], and others. In general, all of these middleware platforms provide a range of services, including naming/lookup, and data transport.

From the perspective of builders of ubiquitous, mobile, and "smart space" applications, one great problem with this range of middleware solutions is that they are virtually incompatible with each other. This means that technologies based on one platform cannot easily be used with the technologies built on another platform. For example, two devices built on Bluetooth and UPnP would be

unable to interact with each other, although 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, despite the fact that these devices are based on platforms designed to provide interoperability.

Many ubicomp and smart space applications, from various research groups (see, for example, [9] [15] [13] [3] [14]) share a property in common: all of them require a facility on which devices from differing middleware platforms can seamlessly use those based on other platforms. With such a facility, end users can achieve a richer variety of device compositions. Further, researchers investigating novel ubiquitous computing applications are more free to develop and deploy their results using the most practical and powerful range of devices available, without being constrained by underlying technology choices. Therefore, removing the barriers to accessing devices and services acrsoss a range of disparate and incompatible middleware platforms is a major technical challenge.

To address this challenge, we propose uMiddle, a universal middleware framework. This framework copes with the growing number of middleware platforms allowing interoperability and sharing across these platforms. It does so without any semantic loss despite differences in namespace, protocol, and abstractions in these diverse platforms. Further, the framework is flexible allowing extensibility to future evolutions of middleware platforms. The contributions of this paper are three-fold: First, we present the design of the uMiddle framework itself, and its mechanisms for service mapping, protocol translation, and modular extensibility. Second, we describe an implementation of the uMiddle framework showing the ease of adapting diverse middleware platforms to this framework. Third, we show the capabilities and limitations of our framework via current and envisioned applications.

The rest of the paper is organized as follows. Section 2 discusses the inherent requirements of ubiquitous computing applications, and a survey of related work. The design of the framework is presented in Section 3 and its implementation is discussed in Section 4. We sketch applications built on top of uMiddle in Section 5, and present our concluding remarks in Section 6.

2 Middleware Bridging in Smart Spaces

We start by discussing concrete applications to motivate the problem of middleware bridging and derive requirements thereof. We follow this with a discussion of the state-of-the-art in middleware bridging and lay the groundwork for our contributions.

2.1 Applications

Consider a Photo Viewing application, in which a user has a Bluetooth mobile phone, and wishes to view images stored on the phone using nearby display devices. While there is a *Bluetooth Basic Imaging Profile* (BIP), this profile can only

Fig. 1. Smart Shelf with four u-Textures (left) and Smart Living Room (right)

enable photo viewing if the phone implements BIP, if the peer devices use Bluetooth, and if they also implement that specific profile. UPnP, on the other hand, defines a MediaRenderer profile that would be implemented by devices such as televisions. While their semantics are similar in terms of the ability to handle images, they are unable to communicate with each other since their communication protocols are different. Of course, simply bridging these two technologies is insufficient, since new devices using totally new middleware technologies will be introduced in the future. For example, our research group has created a Smart Living Room based on a new type of device called u-Texture, shown in Figure 1. uTexture provides a set of building blocks for digital smart furniture, containing a computing engine, sensors, and network interfaces. Our Smart Living Room is a physical environment where applications such as Photo Viewing operate. It is equipped with u-Texture-based furniture and arrays of sensors, appliances, cameras, microphones, video projectors, user interface devices, and so on. In this application, the sensor data acquisition, device control, and u-Texture communication are all built on different middleware technologies, that currently must be integrated by the developer in order to realize this application.

2.2 Requirements

The requirements for the uMiddle framework follow directly from the aforementioned applications and the environments in which they operate.

Interoperability: Applications such as the Photo Viewer require interoperability between devices based on different middleware platforms, through a single unifying infrastructure. Each native middleware system (UPnP, Bluetooth, and so on) has its own specifications for communication. They usually define communication protocols, data types, and control parameters, and these are not universal across middleware platforms. In the case of the Photo Viewing application,

the mobile phone and the TV speak different protocols. Also, while the MediaRenderer specification allows devices to transmit arbitrary data types, phones supporting the BIP generally support only JPEG. Finally, each of them defines its specific commands and their semantics such as those for camera control in BIP. Therefore, the infrastructure must understand these native specifications, and allow both devices to interoperate with each other without requiring that they necessarily understand each others' specifications.

Extensible Adaptability: The diversity of devices in these applications, coupled with their degree of mobility, requires that the infrastructure be able to adapt at runtime to the presence of new devices, and new *types* of devices built on different native middleware platforms. For example, in the Photo Viewer application, the infrastructure must be able to discover and accommodate UPnP devices, while in the Smart Living Room example the infrastructure must be able to use u-Texture devices. The bridging framework must be able to accommodate applications that can be run in different environments such as these, without requiring modification of the application, nor of the middleware, nor of the devices in these environments.

Additionally, entirely new middleware platforms will arise in the future. Therefore, any bridging framework cannot a priori settle on a set of supported device types and middleware platforms; instead, the framework must be extensible to both new device types in existing middleware platforms, as well as entirely new forms of middleware.

Representational Power: Finally, the framework must provide a uniform set of abstractions for representing the services of native middleware platforms, while retaining sufficient representational power to allow the construction of realistic applications – including applications that are indistinguishable from those built directly atop the native middleware itself. There are many aspects of the representations that a bridging middleware must provide.

First, abstractions for naming and lookup must provide a uniform scheme sufficient to handle a wide range of devices. For example, in the Photo Viewer application the infrastructure must be able to accommodate both Bluetooth mobile phones and UPnP televisions into this uniform scheme, and provide applications with suitable means of discovery, despite the fact that device capabilities are represented in different ways in Bluetooth and UPnP.

Second, the infrastructure must provide transport abstractions that can accommodate the various forms of communication found in current (and future) middleware platforms. These transport abstractions must be able to handle not only transmission of small amounts of latency-sensitive control commands, but also bulk data, as well as high-bandwidth audio and video streams. The abstractions present in the infrastructure must represent a common language that enables definition of any device capability, while shielding applications from the particulars of the underlying infrastructure.

2.3 Related Work

Although a variety of ubiquitous computing communications infrastructures have been developed, in general most of these do not provide for extensible communication abstractions. They rely on specific communication protocols, which are assumed to be known by all parties a priori. For example, ADS [12] adopts Ninja [8] for communication; Hodes and Katz' Document-Based Framework [11] uses a specific remote method invocation mechanism, and UPnP is based atop SOAP. These tight bindings with specific underlying mechanisms limit user-level applications. Although UPnP can allow multimedia devices to negotiate and use arbitrary out-of-band communication protocols such as RTP and RTSP, it still requires that these protocols be known to all parties involved in an interaction.

Some of these communications middleware infrastructures provide support for runtime configuration with an emphasis on Quality-of-Service (QoS) management. For example, previous work has used CORBA as an underlying communication mechanism, and added support for timing constraints through pluggable protocol modules [23], or fragmented objects [16] [10]. Systems such as these motivated us to work towards an abstraction of transport mechanisms that could accommodate the multiple semantics provided by implementations of the abstraction; this goal is included in the representational power requirement.

Some directory mechanisms do not provide notifications, which are required to adapt to the dynamically changing configurations of smart spaces. Intentional Naming System INS [1] uses the notion of intentional names, which are a set of attributes and values announced to a network periodically. This period is fifteen seconds or more, and the notification mechanism is not included in its implementation. The Lightweight Directory Access Protocol (LDAP) [26] is a scalable directory mechanism which is also a reactive system.

Systems such as Universally Interoperable Core (UIC) [22] and Speakeasy [5] share many of the high-level goals of our system. UIC provides an abstract distributed component-based middleware, which can bridge multiple communication mechanisms such as Java RMI and CORBA. This approach, however, requires that individual components must be specialized, in order to work with the various communication mechanisms in use; we cannot expect such specialization to happen on a component-by-component basis each time a new communication mechanism appears. Speakeasy allows arbitrary computational entities to find, control, and interact with one another by exploiting mobile code. It shares a design principle with us, in that we cannot require that components have extensive knowledge about the programmatic types, protocols, and media formats of their peers in order to interoperate. Speakeasy has a weakness, however, in that it requires a common platform for mobile code execution to eliminate the need to modify device-side software. Our approach does not require the use of mobile code in order to achieve interoperability.

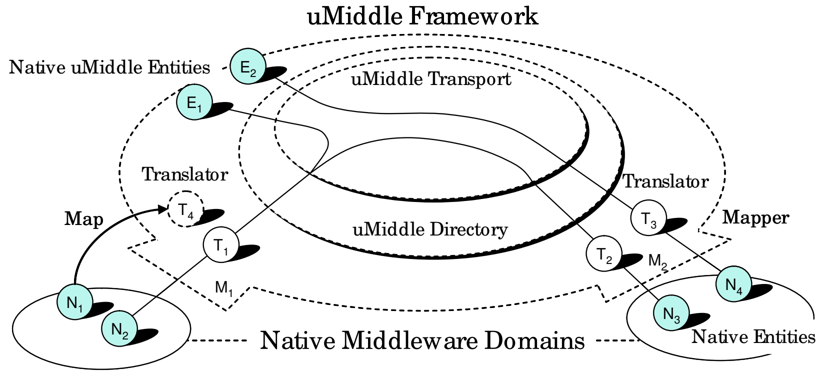


Fig. 2. Graphical Overview of uMiddle Framework

3 uMiddle Framework

3.1 Overview

The uMiddle framework is an infrastructure that addresses the aforementioned requirements. Its principal components are *entities*, *mappers*, *translators*, a *directory*, and a *transport*. The term entities describes a set of distributed software components that abstract devices, their functions, and their communication endpoints. The directory and transport are mechansims through which uMiddle entities interact with one another. On top of these exist applications that compose entities together based on users' preferences and desires. uMiddle is neutral as to the form of such applications – for example, they may be based on tangible UIs, graphical UIs, and so forth.

Figure 2 shows these components of the system, and the basic data flows between them. Shapes depicted with dotted lines represent the components of the uMiddle framework itself. In the diagram, there are two different communications middleware systems being bridged, which are labelled as native middleware domains. Entities N_{1-4} represent "native" devices or services running in these domains; these are *mapped* into the uMiddle framework through mappers M_1 and M_2. Mapping simply allows these devices or services to be represented in uMiddle as abstractions called *translators*. In the diagram, T_{1-4} are translators that represent the underlying native devices N_{1-4}. In addition to supporting devices built using existing middleware platforms, uMiddle also allows the creation of services that are "native" to uMiddle itself. In other words, these services are programmed directly against the uMiddle APIs. In the diagram, E_1 and E_2 denote such "native uMiddle" entities, which would typically run on desktop or wireless mobile computers, and provide services such as transmitting media stored on the machine, rendering media streams to an on-screen window, and controlling devices connected to a computer's peripheral ports. The uMiddle translators

and mappers are abstract components, designed to be entirely independent of particular implementations of devices or middleware platforms.

The diagram above represents the software and hardware components that might be present on a single uMiddle node – typically a desktop or mobile computer that hosts the uMiddle framework, along with one or more entities, and perhaps uMiddle-based applications. uMiddle allows interconnections to be made between entities on a single node; when multiple nodes are connected to the network, they can discover one another, allowing applications to form connections across multiple devices and multiple machines, as we shall explain shortly.

3.2 Design Principle

Before proceeding to the details of each component, let us describe an important principle of the design of the whole framework. It is that uMiddle entities are not programmatically typed. This is an important difference between uMiddle and various middleware communication platforms, many of which define a range of device-specific types. Although such types allow reasoning about device compatibility, they limit dynamic device composition. Such typing means that the operations and semantics of a component are often represented by a single complex type; in order for any peer to communicate with this component, it must have detailed knowledge of the specifics of the type. This means that the Bluetooth phone in the Photo Viewer application must know the complete specification of the Bluetooth BIP in order to communicate with a BIP-enabled printer or photo frame. The prerequisite binding between these two entities makes it difficult to compose the phone with other devices, without the presence of the detailed programming needed to allow it to communicate with devices that implement other types. A similar situation holds in UPnP also. For example, UPnP defines a MediaRenderer type and a Printer type. One could imagine that the semantics of such types are similar enough that it should be possible to send images to either type of device, through similar programming. The actual situation, however, is that these types are completely different, requiring components that use them to be implemented specifically to communicate with each type. Such differences result in a gap between users' intentions, the semantics they ascribe to certain devices, and the actual semantics defined by programmatic types.

If there existed an elegrant set of interface types that represented *aspects* of device functionality – providing crosscutting operations over different types – this problem would be partially solved. However, there will always remain operations highly specific to a given type of devices (a "next page" operation appropriate for Scanners but not to MediaRenderers, for example), that cannot be represented by a universal abstraction layer, and will never be invoked through dynamic device recompositions. The Speakeasy system attempted to provide such a base level of widely useful, aspect interfaces through a mobile code technique. In their mechanism, remaining device-specific operations – which cannot be easily mapped onto the common aspects – are exposed directly to users through

mobile code-based fixed GUIs, downloaded to users' clients to allow them access to device-specific functionality. Our framework solves this interoperability conundrum using a different approach which will be described shortly.

3.3 Entities

Entities in uMiddle are categorized into three groups: devices, services, and ports. A device is a software component that represents a hardware device, and acts as a proxy for it. There need not be a precise one-to-one mapping between device entities and actual hardware devices; for example, integrated devices such as home servers and desktop computers may result in the instantiation of a number of device entities. In turn, each device may contain one or more services that represent certain aspects of its functionality; any device may contain one or more services. The third type of uMiddle entity is the port. Ports provide an abstraction of communication endpoints in our framework. uMiddle allows both input ports (meaning that they can receive data) and output ports (meaning that they can send it); ports are used for the communication of both control and data information.

These abstractions are reflective of the structures found in numerous existing middleware platforms. For instance, UPnP shares a similar notion of devices, each of which may contain multiple UPnP services. Bluetooth devices, likewise, may support multiple service profiles. Jini nodes may instantiate multiple services on the same node. This affinity to existing middleware systems facilitates mapping native devices onto uMiddle's abstractions smoothly.

Service Shaping Our framework solves the interoperability conundrum using an approach called *Service Shaping* [19], instead of programmatic typing. This approach has a number of tenets:

- First, and as already noted, uMiddle entities do not use programmatic typing as a way to represent their operations and semantics. Instead, in uMiddle, ports declare acceptable data types through the use of MIME type strings. This ensures at least a minimum level of interoperability between ports.
- Second, each entity holds a *profile* object that contains a semantic description of its functionality, expressed in human-readable language. This facility is used to allow users to assess semantic-level interoperability between entities.
- Third, we propose the notion of a *physical port*, which allows services to declare their roles within a physical space. For example, a physical port may declare the user-perceivable effects it has within a space, such as rendering data visible in the space. Such declarations are encoded as data types, and are used by applications to verify semantic-level interoperation among entities. For example, if a user wished to view a document, the application can select a service with input port of the document's MIME type, and physical output port with "visible/*" as the data type.

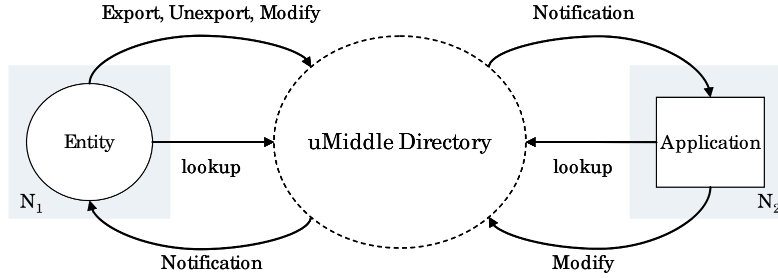


Fig. 3. uMiddle Directory Operations

– Finally, and most importantly, we introduce the notion of unparameterized event and control ports into our framework. In this model, control commands do not include parameters. Instead, services must contain control ports representing each discrete parameter value. For example, in our model, a service may have ports for *on* and *off*, but not a *setPower* port that takes a Boolean parameter. Likewise, events do not include any paramters. So, for example, a service may present ports corresponding to *turnedOn* and *turnedOff*, rather than *powerStateChanged* with a parameter. The advantage of making these types as "narrow" as possible is that arbitrary control and event ports are interoperable with each other, in the sense that any event port can be connected to any control port. Any device-specific operations are represented as control ports. Note that the semantics of ports are disclosed to users through their human-readable profiles. Thus, every operation on an entity can be invoked in response to any event, through user-defined port connections.

3.4 Directory and Mappers

The uMiddle directory service provides an implementation-neutral mechanism for rendezvous among uMiddle entities and applications. Figure 3 shows a high-level view of a uMiddle directory service. In the diagram, gray colored rectangles represent host boundaries. Node N_1 and N_2 respectively host an entity (perhaps representing a device connected to N_1) and a uMiddle-aware application. The directory service provides a means of monitoring entities' profiles, which contain their state along with static attributes such as name, identifier, and so forth. When an entity joins uMiddle, it exports its own profile to the directory. The directory service stores the exported profile in it, and notifies other entities and applications of the event through an implementation specific procedure. uMiddle allows the directory service to be extended to work with underlying native notification, directory, and discovery services. For instance, in case of a directory service based on the INS, the profile is encoded into an intentional name, and the notification procedure is realized by polling. Likewise, when an

entity wishes to change its status, or disconnect from the network, it can modify and unexport its profile.

Mapper The mapper is a component of the uMiddle directory service. Its role is to find entities in a native middleware domain, and then spawn translators that represent those entities. uMiddle defines an XML-based language, called Universal Service Description Language (USDL), for describing policies for mapping and translation. Figure 4 shows a USDL document that describes a UPnP light switch device. USDL consists of two classes of XML schemas: one is generic, and represents common abstractions across all devices; the other is middleware platform specific. Tags in the generic schema include `<device>`, `<service>`, and `<port>`, corresponding to the abstractions discussed earlier, that describe the attributes of uMiddle entities, to which native devices and services are mapped. The `<map>` tag is also an element in the generic schema, and contains any middleware platform-specific tags that might be present. These middleware specific tags are passed to the corresponding mapper to facilitate the mapping operation. For example, the UPnP mapper may use middleware-specific tags that refer to UPnP device or service types.

Adaptability and Extensibility These three different levels of uMiddle system components – a mapper, a USDL document, and a translator instance – respectively correspond to a native middleware platform, a native device type, and the instances of that device type. Our framework is adaptive at each level. There are three distinct scenarios in which adaptation and extension may occur. The first scenario is adding support for a totally new middleware platform. This scenario is accomplished by providing a new mapper implementation corresponding to the new platform, along with defining an XML schema for any middleware-specific attributes that may need to be declared. The second scenario is adding support for a new device type within an existing middleware specification. This scenario is supported by simply creating a new USDL document that defines how the device is incorporated into uMiddle. The third scenario is when new instances of known types of devices appear on the network. uMiddle instantiates new translators as necessary to adapt to device availability.

3.5 Transport and Translator

The uMiddle transport service abstracts multi-semantics data transmission ports, path creation between them, and a transport interface for talking to underlying communication mechanisms. Our framework defines three types of ports for data transmission: streaming media ports, bulk data transmission ports, and event and control ports. These port send and receive data through the uMiddle transport interface, and can operate over a variety of underlying communication mechanisms.

Figure 5 shows the establishment of a communications path, with host boundaries depicted by gray colored rectangles. Each node hosts a transport service

```
<?xml version="1.0"?>
<device xmlns="//umiddle.usdl" xmlns:upnp="//umiddle.upnp"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="...">
  <name>Light</name>
  <description>UPnP light device</description>
  <map>
    <upnp:device type="urn:schemas-upnp-org:device:light:1"/>
  </map>
  <services>
    <service>
      <name>Power</name>
      <description>Power control service</description>
      <map>
        <upnp:service type="urn:schemas-upnp-org:service:power:1"/>
      </map>
      <ports>
        <port portType="control" dataType="*">
          <name>on</name>
          <description>turns on this light</description>
          <map>
            <upnp:input action="SetPower">
              <upnp:argument name="Power">1</upnp:argument>
           </upnp:input>
          </map>
        </port>
        <port portType="event" dataType="*">
          <name>TurnedOn</name>
          <description>fires an event when power is switched on</description>
          <map>
            <upnp:output>
               <upnp:state name="Power">1</upnp:state>
            </upnp:output>
          </map>
        </port>
      </ports>
    </service>
  </services>
</device>
```

Fig. 4. Translation Policy Definition for UPnP Light Device

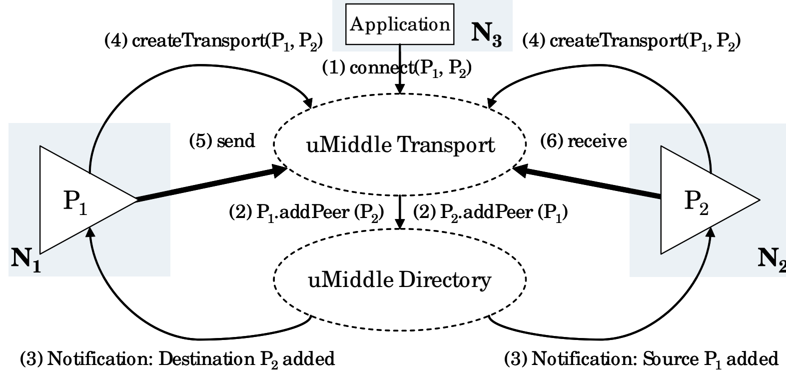


Fig. 5. Path Establishment in uMiddle Transport Service

and a directory service, which shares a single underlying communication mechanism and directory mechanism, respectively, with others. uMiddle nodes N_1 and N_2 host ports P_1 and P_2 respectively. The entities containing these ports are mutually discoverable through the uMiddle directory service; further, each of these ports uses the uMiddle transport service for data transmission. In this example, P_1 is an output port, which sends data to a list of destinations; P_2 is an input port that receives data from sources. An application, hosted on uMiddle node N_3, is requesting a communications path establishment to the transport service on node N_3. The transport service then adds the identifier of each port into the other's peer list attribute in their profiles through the directory service on that node. This information is propagated to the ports through a notification mechanism, and each opens a new underlying communication channel through the transport service. Once the path is established, these ports communicate directly with each other without mediation by the application. This end-to-end abstraction increases the efficiency of device interaction, saving a hop and also avoiding the application being a single point of failure.

All data sent in streaming and bulk transmission ports carry MIME types; those sent through event and control ports use the "wild card" MIME type, since events and commands in our framework are unparameterized. Therefore, there are the following potential communication paths. First, two streaming or bulk transmission ports can be connected to each other if their MIME types match. Second, an event port can always be connected to a control port, enabling any event port to send messages to control an entity. Our framework entails the ability to incorporate a context interpreter as a service which fires events through event ports upon updates of context properties. Users can connect these context events to control ports of uMiddle services to create context-aware device interactions. Finally, a streaming or bulk transmission output port can be connected to a control port; doing so simply "fires" the control port. Examples of this sce-

nario include a path from a camera's video output port to a TV's power control port, which automatically turns on the TV when a video stream is fed to it.

Translator uMiddle translators are instantiated by a mapper in response to the presence of a native device or service, as noted earlier. Translators receive data from the corresponding native device or service, translate the data into MIME-typed data, and then send the data to the transport service (or vice versa). Translators conceal middleware-specific semantics from the uMiddle framework, and provide the abstraction that allows interoperability among devices and services built atop different native middleware platforms. Let us revisit the USDL document that describes the semantics of UPnP light switches shown in Figure 4. In this case, the translator is configured by the mapper to contain a control port and an event port, whose names are `on` and `turnedOn`, respectively. When this translator receives a command through its control port, the translator dispatches a UPnP action to update the native device's `Power` parameter and set its value to `1`, as shown in line 23. Conversely, line 32 defines how the translator will monitor the UPnP `Power` state, and fire a uMiddle event when its value changes to `1`. The translator conceals the underlying UPnP parameter, its state, and the mechanisms for updating and reading it, from the rest of the framework. From the perspective of the rest of the framework, these entities are simple generators and receivers of unparameterized events and commands. The `<description>` tag in the USDL document contains a human-readable statement about the "type" of the device. Applications can use this to present the device to users, allowing them to make decisions about the semantics of these devices; the framework then allows users to interconnect them without needing to know about the details of the native middleware.

4 Implementation

The uMiddle framework consists of a core library implemented in Java, and extension modules including implementations of the directory service, the transport service, and the mapper/translator. As a set, they constitute the uMiddle runtime. We choose Java as an implementation language due to its effectiveness for rapid prototyping; the framework design itself does not entail any language dependency. We have plans to port the framework to C++; both implementations will be able to coexist in one single uMiddle domain. This section also characterizes the performance benchmark results of implemented modules.

4.1 Core Library

The core library includes abstract implementations of uMiddle entities and their profiles. The abstract entities can refer either local or remote devices or services. The profiles wrap actual naming records, such as intentional name strings in INS, provided by concrete directory modules. Applications can acquire entity

Table 1. Directory Services

	Lines	Period
JS	1020	1 week
INS	507	3 hours

Table 2. Transport Services

	Lines	Period
TCP	259	1 hour
UDP	265	1 hour
RMI	248	3 hours
MB	401	2 days

Table 3. Mappers

	Lines	Period
UPnP	627	2 days
BT	490	2 days
WWW	328	2 hours

Period: development period, JS: JavaSpaces, MB: MediaBroker, BT: Bluetooth

profiles from the directory service on any uMiddle runtime in a distributed system. Therefore, our framework supports applications on distributed nodes sharing essentially a single image of the state of the system. Entities are completely independent of the rest of the framework. Even if the underlying directory mechanism or communication mechanism changes, no modification or recompilation of individual entities is needed.

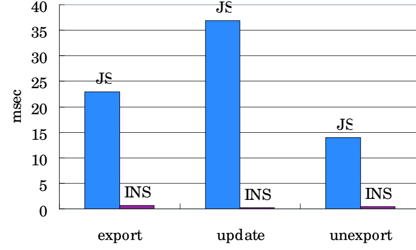
4.2 Directory Service

The core library includes an abstract directory service, which is subclassed by concrete directory modules. Two concrete implementations are currently provided. One is a JavaSpaces-based implementation, which is used during our development. This creates a directory with single-writer and multiple-reader semantics in a JavaSpace. The other is INS-based. Profiles are encoded into intentional name strings that are periodically announced to the network. Table 1 shows the number of lines and development period of these directory service implementations. We believe other implementations can be done with reasonably small amount of effort. We plan to implement uMiddle directory services atop mechanisms with built-in notification mechanisms such as Rendezvous [4] and Simple Service Discovery Protocol [7].

Figure 6 shows a benchmarking result of these modules in terms of exporting a new uMiddle profile to the underlying directory mechanism (the case of an entity joining to uMiddle), modifying an already existing profile (adding a new communication peer, for instance), and unexporting it (leaving from uMiddle). Node (1) and (3) host uMiddle runtime, and Node (2) hosts server programs required for INS and JavaSpaces during the corresponding tests. The INS performance do not include actual network I/O, since the INS library postpones it until the next period of announcement where all the names are written to the network at the same time. This result hence characterizes the common directory service overhead caused by the framework itself.

4.3 Transport Service

The transport service is represented by a class that includes abstract operations for allocating network resources and establishing communication paths. The extension modules currently include the concrete transport service implementations listed in Table 2. The MediaBroker [18]-based transport service contains a



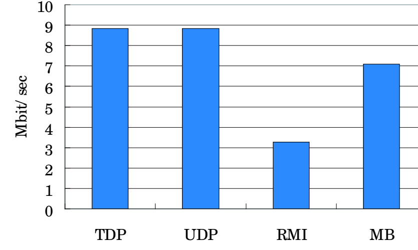


Fig. 6. Directory Service Benchmark

Fig. 7. Transport Service Benchmark

These results are benchmarked over a network of the following uMiddle nodes connected by a 10Mbps Ethernet hub: (1)IBM ThinkPad T23 (Mobile Pentium-III 1.2GHz, 512MB RAM, Fedora Core 3 Linux), (2) IBM ThinkPad T42p (Pentium M 2.0GHz, 2GB RAM, Windows XP Service Pack 2), and (3) IBM ThinkPad T42p (Pentium M 2.0GHz, 2GB RAM, Fedora Core 3 Linux).

media transformation mechanism. This capability enables ports of different data types to be connected with each other through a runtime transformation process, increasing interoperability between devices. Table 5 shows the number of lines and development period of these transport service implementations; again, given the simplicity of creating such implementations, we believe that others can be done with small amounts of effort. Figure 7 shows performance of these. These tests were done by sending 1400-bytes messages from (3) to (1). During MB transport test, node (1) hosted a MediaBroker server. Also, both of (1) and (3) hosted RMI registry program during the RMI transport test.

4.4 Mapper and Translator

The core library provides a skeleton mapper implementation, containing abstract operations used by concrete mappers to instantiate translators. The only convention required for translators, on the other hand, is that they must implement either the uMiddle device or service interface. Table 3 shows the currently available mappers and translators, and the number of lines and time required to support a given platform. We used the Linux BlueZ library with a Java wrapper to create the Bluetooth mapper, and the CyberLink Java library for the UPnP mapper. The end-to-end performance to control a UPnP light switch service via a Bluetooth mouse (crossing middleware platforms) is 160 milliseconds on average. This includes 150 milliseconds consumed in the Java UPnP library. The Web mapper downloads resources from a URL specified in USDL, and exports them to uMiddle through bulk data transmission output ports. Such resources include real world information such as severe weather warnings, TV programs, stock quotes, etc. We intend to utilize this information for a range of context interpreter services in the future.

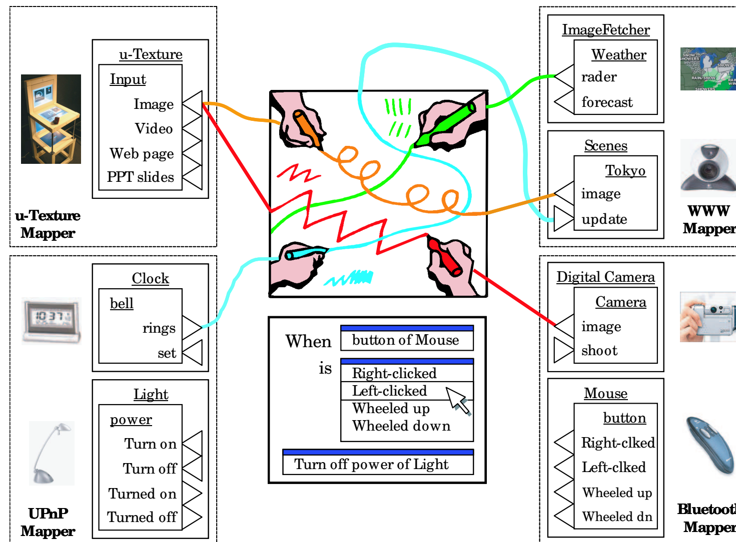


Fig. 8. Multimodal End-user Programming in Mixed Smart Space

5 uMiddle Framework Usage Examples

This section highlights several scenarios that take advantage of our framework; these have been implemented by our research group.

We have created a heterogeneous smart space environment, with the mappers described earlier; this space supports a range of different native middleware platforms, allowing them to coexist. Figure 8 shows the configuration of this environment, including our Photo Viewer implementation. The central two squares represent alterative end-user programming systems for controlling the environment; the top is connection-oriented, while the bottom is a programming-by-example system. The connection drawn by the bottom-right hand between u-Texture and the digital camera is essentially the Photo Viewer application described at the start of this paper. The u-Texture is mapped to uMiddle by the uMiddle u-Texture native platform mapper, as are the devices that utilize other native middleware platforms. This u-Texture device can then be connected to arbitrary other devices if their data types match. Therefore, in addition to displaying digital camera output, it can be used to view weather radar, or webcam images, when used with devices translated by the Web mapper, as shown in the figure. In this case, the u-Texture is updated with a webcam image everytime the clock is triggered. A similar configuration can be also used to instantiate an intercom-style application that allows users to initiate multimedia communication with each other throughout a home. The same clock service event used in this application can also be connected to other event input ports, such as the power service on the light, or the imaging service of the camera.

6 Conclusions

uMiddle is a universal framework for bridging diverse middleware platforms, which is adaptive to smart space environments and also configurable by researchers. We have motivated our framework by examining the requirements of common smart space applications. In particular, we have discussed the key features of this framework: the use of a service shaping technique that introduces high interoperability between entities, a multi-semantics communication library, an extensible system for incorporating entities based on existing middleware, and flexible protocol translator among them. We have presented our implementation of uMiddle, along with a set of extension modules that are ready for end-users and researchers to use to create applications that leverage a range of different middleware platforms. We have also described how a number of applications can be implemented atop uMiddle.

Our future work is concerned with resolving several functional limitations. One of them is QoS management in the framework, which provides a unified abstraction of QoS management mechanism over multiple different transports. The other is a clear abstraction of the context interpreter to make contexts exchangeable between different mechanisms such as those based on first order predicate logic [21], and Bayesian networks [6] [24] [20]. Finally, we plan to port the framework to C++, to better support wearable or mobile nodes with limited resources; this implementation will coexist with, and be compatible with, the current Java-based implementation.

References

1. W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Symposium on Operating Systems Principles*, pages 186–201, 1999.
2. Bluetooth Consortium. Specification of the Bluetooth System, Version 1.2, Core, Nov. 2003. http://www.bluetooth.org.
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. S. Cheshire and M. Krochmal. Dns-based service discovery, Aug. 2004. Internet Engineering Task Force, INTERNET DRAFT, draft-cheshire-dnsext-dns-sd-02.txt.
5. 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.
6. D. Fox, J. Hightower, L. Liao, D. Schulz, and G. Borriello. Bayesian filtering for location estimation. *IEEE Pervasive Computing*, 2(3):24–33, July-September 2003.
7. Y. Y. Goland, T. Cai, P. Leach, and Y. Gu. Simple service discovery protocol/1.0, operating without an arbiter, Apr. 1999. Internet Engineering Task Force, INTERNET DRAFT, draft-cai-ssdp-v1-03.txt.
8. S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proceedings of Fourth Symposium on Operating Systems Design and Implementation(OSDI2000)*, Oct. 2000.

9. A. Harter and A. Hopper. A distributed location system for the active office. *IEEE Network*, 8(1):62–70.
10. F. J. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier. Aspectix: A quality-aware, object-based middleware architecture. In *Proceedings of the 3rd IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, Sept. 2001.
11. T. D. Hodes and R. H. Kats. 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.
12. A. C. Huang, B. C. Ling, J. Barton, and A. Fox. Making computers disapper: Appliance data services. In *Proceedings of the International Conference on Mobile Computing and Networking (Mobicom)*, pages 108–121, 2001.
13. S. S. Instille. Designing a home of the future. In *IEEE Pervasive Computing Magazine 1(2)*, pages 80–86, Apr. 2002.
14. 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.
15. 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.
16. R. Koster and T. Kramp. Structuring qoS-supporting services with smart proxies. In *Proceedings of the IFIP/ACM Middleware Conference (Middleware 2000)*, 2000.
17. Microsoft, Corp. Universal plug and play device architecture reference specification, 1999.
18. 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.
19. 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.
20. D. J. Patterson, L. Liao, D. Fox, and H. Kauts. Inferring high-level behavior from low-level sensors. In *5th International Conference on Ubiquitous Computing (Ubicomp2003)*, pages 73–89, Oct. 2003.
21. M. Roman, C. Hess, R. Cerqueira, A. Renganat, R. H. Campbell, and K. Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. In *IEEE Pervasive Computing*, pages 74–83, Dec. 2002.
22. 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.
23. D. C. Schmidt, D. L. Levine, and S. Mugnee. The desing of the tao real-time object request broker. *Computer Communications*, 21(4), Apr. 1998.
24. F. Sparacino. Sto(ry)chastics: A bayesian network architecture for user modeling and computational storytelling for interactive spaces. In *5th International Conference on Ubiquitous Computing (Ubicomp2003)*, pages 54–72, Oct. 2003.
25. Sun Microsystems, Inc. Jini Architecture Specification, Nov. 1998. http:// www.javasoft.com/ products/ jini/ specs/ jini-spec.pdf.
26. W. Yeong, T. Howes, and S. Kille. Lightweight directory access protocol, Mar. 1995. RFC1777.