# An Infrastructure for Extending Applications' User Experiences Across Multiple Personal Devices

*Jeffrey S. Pierce and Jeffrey Nichols*
IBM Research
650 Harry Rd.
San Jose, CA 95120, USA
Tel: 1-408-927-1282
{jspierce, jwnichols}@us.ibm.com

## ABSTRACT

Users increasingly interact with a heterogeneous collection of computing devices. The applications that users employ on those devices, however, still largely provide user experiences that assume the use of a single computer. This failure is due in part to the difficulty of creating user experiences that span multiple devices, particularly the need to manage identifying, connecting to, and communicating with other devices. In this paper we present an infrastructure based on instant messaging that simplifies adding that additional functionality to applications. Our infrastructure elevates device ownership to a first class property, allowing developers to provide functionality that spans personal devices without writing code to manage users' devices or establish connections among them. It also provides simple mechanisms for applications to send information, events, or commands between a user's devices. We demonstrate the effectiveness of our infrastructure by presenting a set of sample applications built with it and a user study demonstrating that developers new to the infrastructure can implement all of the cross-device functionality for three applications in, on average, less than two and a half hours.

**ACM Classification:** H5.2 [Information interfaces and presentation]: User Interfaces. Graphical user interfaces.

**General terms:** Design, Human Factors

**Keywords:** multi-device services, multi-device user experiences, application development, infrastructure

## INTRODUCTION

Users are shifting from interacting with a single personal computer to interacting with a heterogeneous collection of computing devices (e.g., desktops, laptops, tablets, PDAs, mobile phones). A recent study of 27 people from academic and industrial research labs revealed that on average they employ more than five computing devices [5]. While those users may be at the leading edge of this trend, users with just a work computer, home computer, and mobile phone already employ three devices.

Users that work with multiple devices want a seamless experience when interacting across them [5, 15]. They want easy access to their files, but they also want access to meta-information such as the history of websites they visit and people they email. They want to be able to easily use one of their computers to control or access the others. In short, they want to employ their devices as a more integrated whole, rather than as a collection of independent devices. The current user experiences provided by devices and applications, however, are far from fulfilling those desires.

Part of the problem is that extending an application's user experience across multiple devices requires additional functionality. For example, a search application for a single device generally consists of three basic parts: an indexer, a query engine, and a user interface. An application for searching across multiple user devices is more complex. It also needs mechanisms for determining the addresses and availability of a user's devices, connecting to those devices, sending and receiving messages, and preventing unauthorized access. Implementing that additional functionality represents a significant barrier for developers.

We set out to reduce that barrier by creating an infrastructure that developers can use to allow their applications to easily exchange information, events, and commands across a user's devices. While there are other infrastructures that support the development of applications that span multiple devices in general, our infrastructure differs from them by explicitly focusing on supporting applications that span *personal* devices.

More concretely, our infrastructure elevates device ownership (or more broadly primary use, because users may employ devices owned by other entities, such as their companies) to a first class property and uses it to organize communication among devices. Ownership is a useful organizing relationship because, unlike other potential relationships such as physical proximity, it implicitly incorporates information about access permissions. Ownership is also stable over time, changing over months or years rather than hours or days.

In the next section we describe the basic design of our infrastructure. We developed over twenty applications in parallel with creating it in order to refine the functionality it provides. We describe a subset of those applications and discuss the refinements we made. We then walk through implementing a sample Search application to provide a sense of a developer's experience using the infrastructure, and we contrast our infrastructure with previous work. We close with an evaluation of our infrastructure, some additional lessons learned, and potential next steps.

## BASIC INFRASTRUCTURE DESIGN

We started with two design goals: elevate device ownership to a first class property, and allow applications on a user's personal devices to easily send information, events, and commands to each other. We also decided to accept rather than attempt to change some existing user practices, such as turning off devices and employing firewalls, dynamic IP addresses (DHCP), and network address translation (NAT) on them. That decision led us to impose several constraints on our infrastructure. Accepting that users turn off devices meant that the infrastructure needed to be able to tell applications of the availability as well as the existence of a user's devices in order to facilitate communication among them. It also meant that the infrastructure needed to help applications cope with devices that are only intermittently available. Finally, accepting that devices may be behind a firewall, that their IP addresses may change, and that they may use NAT meant that the infrastructure had to help applications communicate with devices that are difficult to contact (e.g., that have IP addresses that are unreachable outside of their local subnet).

Based on these goals and constraints, we chose to build our infrastructure using an instant messaging (IM) architecture. Current IM architectures address many of our needs. They provide synchronous communication between entities (traditionally users), facilities for describing relationships and controlling communication among entities, and presence updates to communicate availability. In addition, clients open persistent, outgoing network connections to a central server that routes messages between them, allowing clients to receive incoming messages without needing to accept incoming network connections. IM architectures also assign fixed addresses to entities, allowing other entities to route messages to them regardless of their actual IP address. As an added bonus, most developers (and users) are familiar with instant messaging.

We chose an IM architecture based on the IETF standard Extensible Messaging and Presence Protocol (XMPP) [6], also known as Jabber, for three reasons. First, several open source libraries exist for XMPP, simplifying the development of new XMPP clients. Second, XMPP messages are extensible; they are just XML fragments, and the protocol itself defines how to add custom elements to them. Finally, most open source XMPP servers are themselves extensible through plug-ins, allowing us to incorporate new server functionality without building a server from scratch. We took advantage of XMPP's extensibility to make three key changes to better fit our needs. We describe the result as an infrastructure for creating *personal information environments* (PIEs).

First, we added support for devices as well as users. We considered simply treating devices as users (giving each its own IM account), but to reduce the administrative overhead (for both users and infrastructure) we chose to instead affiliate devices with users. Users create a single account on a server and affiliate their devices with it. Each device authenticates to the server as the user and has an address of the form userid@server/device. The server maintains a persistent list of personal devices for each user that the user's applications can access and that the infrastructure uses to determine which messages an application receives; by default an application only receives messages from the user's own devices (avoiding the need for developers to prevent access by devices owned by other users).

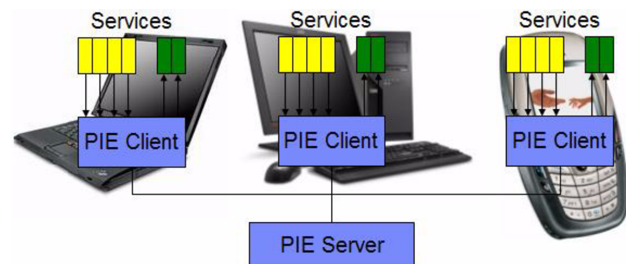


Figure 1: The server, clients, and services in our infrastructure.

Second, we extended traditional IM to include support for applications, which we call *services* in our infrastructure. Users run a single client on each of their devices that connects to the server. Individual services connect to that client, and it handles routing messages to and from the server for them (see Figure 1). We considered having each service connect directly to the server, but that would increase both the complexity of the services and the load on the server. Our approach allows us to hide the details of communication with the server from service developers. It also allows us to provide functionality that is common across services in the client itself. For example, our client monitors the availability of the user's other devices by handling presence announcements from the server, allowing service developers to ignore device availability unless their service actively requires it.

Third, we improved support for asynchronous communication. Some IM architectures queue messages for later delivery to off-line entities, but queuing is problematic for infrequently connected devices because of the potential volume of (often outdated) information. We go beyond such "store and forward" approaches by allowing services to replace an outdated queued message with a new one (or cancel the message altogether). We also extended our server with a data repository that provides "store and retrieve" functionality to services. Services can store arbitrary XML fragments on the server by simply sending it a message

containing the XML data, an identifying tag, and the identity of the associated service. To retrieve data, services send a message containing just the tag and service identity. Services can use the repository to, for example, store the most recent version of shared state information so that services on intermittently connected devices can quickly determine the current state. The repository keeps each user's data separate, so that by default one user's services cannot access another user's data.

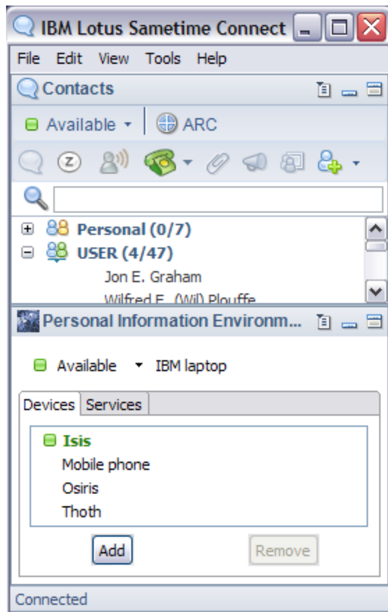


Figure 2: The Java client running in Lotus Sametime.

Our current implementation consists of a server and two versions of our client. Our server is an open source XMPP server that we extended via a plug-in to support persistent device collections and to improve asynchronous communication. One version of our client is a Java Eclipse extension that runs on Windows, Mac OS X, and Linux devices and integrates directly into our corporate environment's standard IM (see Figure 2) or email (Lotus Notes 8) applications, avoiding the need for users to run a separate application. The other version is a .NET Compact Framework client for Windows Mobile devices.

We also provide three libraries to simplify the creation of new services. Our infrastructure provides two ways for services to communicate with a client. First, services can integrate directly with the Java client via an Eclipse extension point. One of our libraries simplifies the development of such services in Java. Second, services can run as independent applications and communicate with a client over a local socket (exchanging simplified XMPP messages). Because these services run independently, developers can write them in any language; a service just needs to be able to read and write XML. These "socket" services can leverage arbitrary native libraries to provide functionality specific to particular device types or even add multi-device functionality to extensible legacy applications. We provide two libraries to simplify the development of these "socket" services: a Java library and a library for building services that are XPCOM extensions to Mozilla applications, such as Firefox and Thunderbird.

## SAMPLE MULTI-DEVICE SERVICES

We developed over twenty services in parallel with implementing the infrastructure both to test the flexibility of the infrastructure and to identify potential refinements to it that might improve the developer experience. We initially concentrated on services that address needs identified by previous research, such as light-weight information transfers and the ability to easily control one device from another [5, 15]. However, we also built services that vary widely in structure and functionality to verify that our infrastructure offers breadth. For example, some of the services store as much of their information on the server as they can, while others adopt a peer-to-peer model and only use the server to route messages. In this section we describe a subset of the services we built.

The ability to more easily share information across devices is a capability commonly requested by users [5, 15]; we therefore built several services that send information between devices. Our Notebooks service allow users to create, access, and edit multiple text notebooks on any of their devices. For example, a user could keep a log of research ideas using the service and access or modify it from any of his devices. The service shares changes to a notebook on any particular device with the user's other available devices in near real-time and ensures that devices that connect later can get the most recent versions of the notebooks by storing them on the server. We also built a Shared Lists service (Figure 3) that offers the same functionality for lists: users can create lists (e.g., a grocery list) and access or modify them from any of their devices. That service drove the addition of new functionality to allow users to share lists with each other. For example, a family could share their grocery list.

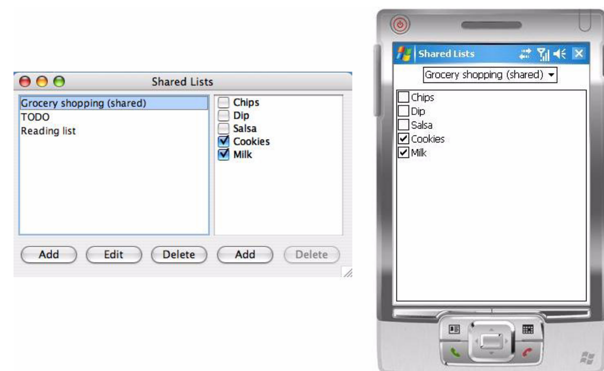


Figure 3: Desktop and mobile interfaces to the Shared Lists service.

In addition to sharing small pieces of information, services can also share entire files. We built a simple File Synchronization service (Figure 4) that mimics services such as FolderShare [8]. It allows users to synchronize sets

of files across their personal devices. Users can choose which of their devices participate for each set of files.

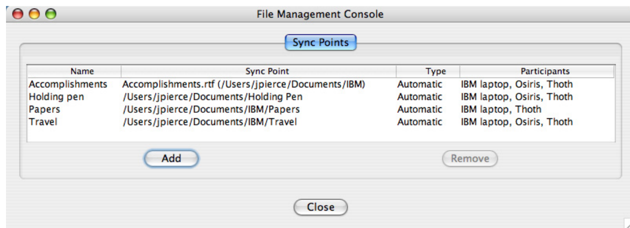


Figure 4: An interface for managing the File Synchronization service.

Of course, sometimes users only want to transfer information between their devices once instead of continually synchronizing it. We therefore built an Information Transfer service that allows users to drag and drop (or cut and paste) text, URLs, or files onto one of their devices listed in the client's interface to transfer them to the target device. This service combines sending information with sending commands. The receiving device will open transferred text in a window, it will open a transferred URL in its default browser, and it will store a received file in a pre-specified directory and then open a file browser showing that directory. We also built a Send a URL service that mimics multi-browsing [11]. Built as a Firefox plug-in, the service allows users to right-click on a URL and choose a device to open the URL on (Figure 5).

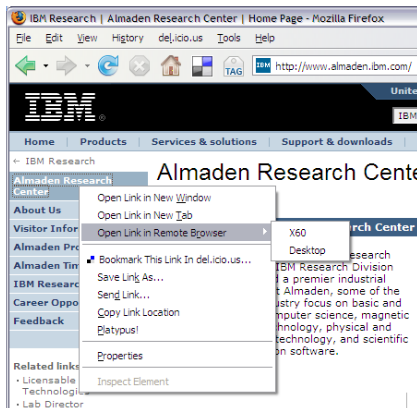


Figure 5: The Firefox Send a URL service.

In addition to pushing information to other devices, services can send commands to other devices to pull information from them. For example, we built a Browse service that allows users to visually traverse a remote device's file system and choose files to transfer to the local device. We also built a multi-device search capability, composed of a Search Console service and specialized Search services, to allow users to search across their available devices. A user initiates a search by typing keywords into a Search Console service's interface. That service sends those keywords to Search services on the user's other available devices. Taking advantage of the fact that developers can implement services differently for different devices, a receiving Search service may search for the keywords using Google Desktop Search (Windows), Spotlight (Mac OS X), or Beagle [3] (Linux) and will return the results to the requesting device. The Search Console service aggregates and presents the results to the user (Figure 6). The user can choose a particular result and tell the service to retrieve that file (or URL, for search tools that index URLs) and open it.

Previous research suggests that users would like to share their interaction histories across devices [5]. We created a Recent Shortcuts service [25] that shares the contact information of people that the user has recently emailed or instant-messaged, the identity of attachments in email messages that the user has recently viewed, and the identity of files that the user has recently accessed. In the service's interface (Figure 7) the user can click on a presented individual's contact information to initiate an email or IM to that person or double-click on a listed attachment or recent file to retrieve and open it.

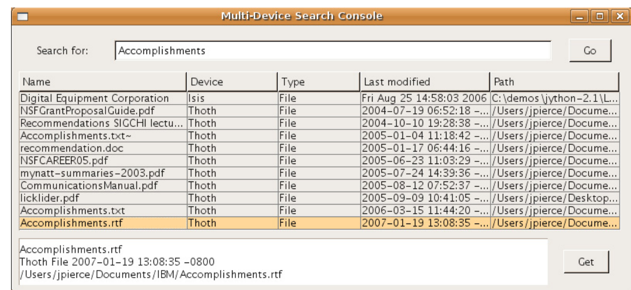


Figure 6: The Search Console service.

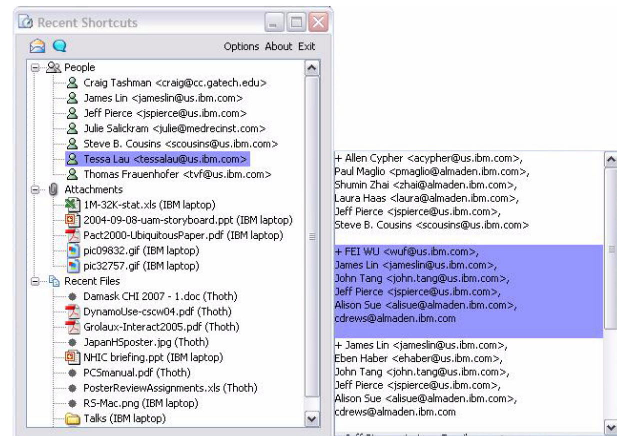


Figure 7: The Recent Shortcuts service.

Services can also send events to other services to help a user's devices coordinate their activities. We built a Phone Events service that sends events from mobile and voice-over-IP (VOIP) phones to a user's other devices when the user makes or receives a call, when a call connects, and when a call ends. A Phone Context service on each receiving device monitors phone events and provides possible context by employing the local Search service to find content containing the other party's phone number.

We also built a Context service that monitors keyboard and mouse activity on the local device and uses that information to add or remove the device from two server-side aliases: InUseDevices that the user has interacted with in the last

minute and RecentlyUsedDevices that the user has interacted with in the last five minutes. Other services can use those aliases to route pertinent messages to appropriate devices. For example, we built a Thunderbird extension that sends alerts about new email messages to InUseDevices and a Notification service that pops up a small window to display such alerts.

## REFINING THE INFRASTRUCTURE

Building sample services in parallel with implementing the infrastructure helped us identify opportunities to simplify service development. In this section we describe refinements that we made to the infrastructure.

In order to send information, commands, or events to a service on another device, the sending service needs to specify both the receiving device and the receiving service. We initially required that clients specify the full address (userid@server/device) of a receiving device. However, we quickly found this cumbersome; in many cases a service would not otherwise need to know the userid or server. We therefore also allow services to specify a receiving device using just its device name. The client handles expanding device names into full addresses before forwarding messages to the server.

Sending a message to multiple devices initially required sending separate messages to each one. Not only is this cumbersome, it also increases network traffic between the clients and the server. To simplify matters, we introduced server-side aliases. The server provides default aliases for each user, including one for all of the user's devices and one for just the user's available devices. The server also allows users and services to create, query, edit, and delete custom user-specific aliases. Such custom aliases are useful for communicating with subsets of devices that share a common property (e.g., WorkDevices, HomeDevices, InUseDevices, RecentlyUsedDevices, Laptops, Desktops, Tables). Services can address messages to just an alias (e.g., allAvailableDevices) or to a full address (e.g., userid@server/allAvailableDevices). The server examines each incoming message for an alias. If it finds one, it routes a copy of the message to each of the alias' members.

Services also need to specify the receiving service on a device. Our initial design required that services include the receiving service's name (e.g., "Search") in a message. Services register a name with the client when they start to allow it to route messages correctly. However, this approach made sending a message to multiple services cumbersome, and it required that sending services know the exact names for receiving services. We therefore added the ability for services to indirectly specify receiving services using the XML namespaces on the extensions (custom XML fragments) that they embed within a message. Services now tell the client what XML namespaces they are interested in (e.g., "ibm:pies:services:search") as well as their name when they start. Multiple services can register for the same namespace, and a service can register interest in multiple namespaces. When a client receives an incoming message, it first checks for a named recipient service and forwards the message appropriately if one exists. Otherwise it forwards a copy of the message to every service expressing interest in any of the XML namespaces on the message's extensions. The XML namespaces thus function like types: services tag sections of a message as having a particular type, and clients route the message to services that handle that type. This approach provides more flexibility for composing services, allowing multiple services to handle a message and services to send messages without knowing the exact recipients.

We found that sending and receiving files is a common component of services. We therefore added functionality to the client that allows services to request a file transfer to or from another device. Services requesting a transfer can specify a remote service to notify when the transfer completes (or fails), and they can request that the client also notify them. Clients provide a default behavior (opening a file browser on the receiving device showing the directory containing the transferred file) that services can request in lieu of notifying a service on the receiving device.

Our client's user interface lists active services in a separate tab from the user's devices and allows users to select a service and start, stop, or configure it. However, we found that in some cases it made more sense to allow services to extend the client's interface by adding context menu items to listed devices. We therefore allow services to specify whether they provide device context menu items and, if so, the item labels. Services can notify the client if the enabled/disabled status of a menu item changes, and the client notifies a service if the user selects a menu item it provides.

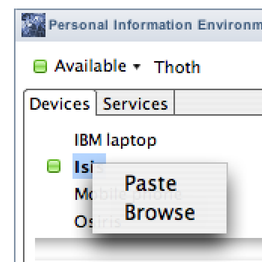


Figure 8: A device context menu with items from two services

While we initially assumed that each device would run its own client, we realized that some devices might not be sufficiently capable. We encountered the latter case for our Phone Events service, where we wanted to allow users to include their office VOIP phones in their device collections in order to broadcast events from them to their other devices. We therefore now allow proxies (typically one of the user's more capable devices or a trusted third party device) to run a client on behalf of another device.

Although our primary focus is communication between a user's devices, we realized that developers might also want services to send messages to other users' devices (e.g., to share changes to a grocery list among family members). While our basic infrastructure allowed services to send

messages to other users' devices or aliases, the receiving client would not deliver them for security reasons. We added the ability for services to tell the local client that they want to receive all the messages addressed to them, not just those from the user's own devices. This addition affords developers more flexibility, but requires that they implement filtering for inappropriate messages. We also extended the server so that services can set read and write access permissions for other users on their server-stored information. That extension improves asynchronous communication between users by allowing services to share information that they store on the server on a per-user basis.

## IMPLEMENTING AN EXAMPLE SERVICE

Our infrastructure thus provides a variety of capabilities to simplify implementing multi-device services. In this section we describe implementing a sample service in order to convey a sense of the developer experience, focusing on how the developer employs the infrastructure's capabilities. We will use one variant of our Search service as our example. This service will take incoming search requests, run the search using Spotlight (Mac OS X's built-in search mechanism), and return a list of candidate matches.

Creating a new service requires specifying three things: the service configuration, how the service handles changes in the state of the service and the client, and how the service handles incoming messages. Specifying the configuration is the easiest. Assuming that we want an Eclipse extension service (the process is similar for services that connect to the client via a socket), we first create a new class SearchService that extends a provided Service base class. We then implement a constructor that sets the service name and builds a list of the namespaces of interest. This service will use Mac OS X's Spotlight to search, so we assign it a name that differentiates it from services using other search mechanisms. We want this service to respond to search requests in general, not just Spotlight searches, so we assign it interest in a general namespace that a variety of services can use to communicate search requests and results.

```
public SearchService() {
   serviceName = "Spotlight Search";
   namespaces = new ArrayList();
   namespaces.add("ibm:pies:services:search");
}
```

Finally, we choose whether to override the default methods that configure whether the client should wait to start the service after instantiating it, whether the service provides a user interface for configuring it, whether the client interface should not show the service in its list of active services, whether the service wants updates in the availability or membership of the user's personal devices, whether the service provides context menu items for the client interface's list of devices, and whether the service accepts messages from other users' devices. All default to no; this service will keep those defaults.

Next a service has to handle changes in its own state and in the client's state. The client will notify a service when it should *start* (passing the client's connection status as a parameter) or *stop* running. The client will also notify a service when it needs to *suspend* sending messages because the client is no longer connected to the server or when it can *resume* sending because the client regained its connection. Our search service will simply use those methods to keep track of its own state and the client's state.

The last step is to specify how a service responds to incoming messages. XMPP refers to these as "packets" and distinguishes three types: presence, IQ (for information-query), and message. Presence packets indicate a change in the availability of a device, while IQ packets are primarily for setting data on or retrieving it from the server. Our service does not need to handle either of those types, so it will ignore them. Message packets are for general communication between entities, and as such are what services primarily use to communicate. The XML for a simple outgoing message packet with a single custom service extension might look like this (the server later adds the sender's ID to the message):

```
<message to="bob@pieserver/availableUserDevices"
         id="Initiate Search">
   <x xmlns="ibm:pies:services:search">
         <keywords>multi-device service</keywords>
         <numberOfResults>15</numberOfResults>
   </x>
</message>
```

While message packets can contain multiple custom extensions, each consisting of an XML fragment, in most cases a message between services only requires a single extension. A notable exception is an infrastructure-specific extension that services can include to cause the server to store the data from another extension; this allows messages to simultaneously transmit data to other devices and update stored data on the server. Our service will handle incoming message packets in order to process search requests.

```
public void handleMessagePacket(MessagePacket
               mPacket) {
```

Inside this method our service first determines what this message is about. The service shares interest in its XML namespace with other services that are initiating searches or returning results, so it needs to confirm that this message is initiating a search. It does that by examining the packet's ID field, which services can use to indicate the purpose of a message. We will assume the convention that services using the "ibm:pies:services:search" namespace will start their packet ID with the phrase "Initiate" when sending requests.

```
String packetID = mPacket.getID();
if (!packetID.startsWith("Initiate")) { return; }
```

If it is a search request, the service will determine what keywords to search for by locating the extension with the search namespace and retrieving the child XML elements within the extension's XML fragment. The infrastructure parses the fragment into a hash of key:value pairs, where keys are XML tags and values may be strings, nested XML fragments, or a list containing either of those two (when a

tag repeats within a fragment). Our service retrieves the contents of the tag "keywords". It also checks if the fragment optionally specifies the number of desired results.

```
String namespace = (String) namespaces.get(0);
PacketExtension extension =
   mPacket.getExtension(namespace);
PacketFragment fragment =
   extension.getChildElements();
String keywords = (String)
   fragment.get("keywords");
int numResults = 10;
if (fragment.containsKey("numberOfResults")) {
   numResults = Integer.parseInt((String)
         fragment.get("numberOfResults"));
}
```

The service then executes the local search for those keywords; we elide those details as they are independent of how the service interacts with our infrastructure. Once the service has completed the search it formats the results as an XML fragment. Our service will do that by creating one fragment to contain all of the search results and then creating additional nested fragments for each individual search result.

```
SearchResult[] searchResults = search(keywords,
   numResults);
PacketFragment results = new PacketFragment();
for (int i = 0; i < searchResults.length; i++) {
   SearchResult searchResult = searchResults[i];
   PacketFragment result = new PacketFragment();
   result.put("category",
         searchResult.category());
   result.put("title", searchResult.title());
   result.put("time", searchResult.time());
   result.put("url", searchResult.url());
   result.put("snippet", searchResult.snippet());
   results.put("result", result);
}
```

The service next determines the return address (in this case, the sender of the request) and assembles a new packet ID by removing "Initiate" from the front of the ID and adding "Results" to the end of it. The service creates a new message packet and adds the results fragment as an extension with the service's namespace of interest. If the client is currently connected to the server the service asks it to send the packet; otherwise the service assumes the search results will not be timely and discards the packet.

```
String to = mPacket.getFrom();
String newPacketID =
   packetID.replaceAll("Initiate ", "");
newPacketID += " Results";
MessagePacket newPacket = new MessagePacket(to,
   packetID);
String namespace = (String) namespaces.get(0);
newPacket.addExtension(namespace, results);
if (clientIsConnected) client.sendPacket(mPacket);
```

Those are the basic steps to implement our service. Two of the other sample services we built can interact with it. Our Search Console service sends search request messages using the search namespace to each of the user's available devices and processes the results. It uses an ID of "Initiate Search" in its requests and looks for replies with the ID "Search Results". Our Phone Context service also leverages search services. The Phone Context service on a particular device looks for incoming phone events, and when it receives one it sends a search request with the phone number of the other party to the local device. It uses an ID of "Initiate Phone Context Search" in its requests and looks for replies with the ID "Phone Context Search Results".

## RELATED WORK

This example service implementation highlights three key aspects of our infrastructure. Previous products and research projects share some of these aspects, but our infrastructure is the first to combine all three:

1. It supports developing services that enhance interaction across personal devices.
2. It makes device ownership a first class property, allowing service developers to leverage a list of a user's personal devices and enabling the infrastructure to provide services with additional functionality such as default aliases and access control.
3. It provides simple mechanisms for sending information, commands, or events to other devices.

A variety of research projects and commercial products share our third aspect: they provide simple mechanisms to exchange data between devices. Shared folders, file transfer programs, and USB flash drives all allow users to transfer files between arbitrary devices. Groove [12] is a somewhat special case that sends more fine-grained information between devices; while primarily designed for multi-user collaboration, users can synchronize information across their devices by logging into their Groove account on each one. Other programs send events and commands as well as information to allow users to remotely access [18] or control [23] another computer.

Other work allows users to explicitly build collections of devices that persist beyond a single interaction session. Much of this work has focused on synchronizing or transferring files across devices [1, 8, 20], but researchers have also explored transferring meta-data about activities [2, 26]. That body of work, however, consists of a set of independent services, rather than infrastructures or tools to help developers create their own services.

There has been research on infrastructures that facilitate the development of services that span devices, but that research has not focused on using device ownership to organize device collections. Interest in ubiquitous computing has driven research on "roomware" or "meta-operating system" infrastructures that assemble devices in a shared physical environment into a persistent collection [9, 19, 24]. Users' personal devices are second-class citizens in these collections, however; the infrastructures focus on allowing users to temporarily add their personal devices to the environment to enhance its capabilities, rather than on allowing them to organize and employ their devices independent of it. These infrastructures also provide little support for creating multiple collections of devices (e.g.,

one for each user) or for communicating between collections.

Researchers have also explored infrastructures that span ad hoc collections of devices [13, 14, 16, 17, 22]. Most of this work focused on allowing users to improve interaction with their personal mobile devices by temporarily annexing devices in the local environment, but some also explored how to allow users to collaborate by temporarily combining the resources of their physically proximate devices.

Our infrastructure differs from both infrastructure types by focusing on users and the devices they employ rather than on physical proximity (whether ad hoc or based on the semantics of local environments). We believe that this approach is more likely to aid the development of a variety of services that improve interaction across personal devices.

There is a class of related work that has explored an alternate solution to improving interaction across multiple devices. Rather than making it easier to send information, commands, and events between personal devices, that work has explored how to turn different physical computers into the same logical computer. The data required to provide the same user experience across devices may transfer over the network [7, 21], or it might reside on physical memory that users carry between devices [4]. However, this work makes two big assumptions: that users want the same interaction experience across all of their devices, and that their devices are physically similar. Both assumptions are problematic. Some users employ multiple devices to get different interaction experiences, while some devices that users employ (e.g., smart phones vs. laptops) have drastically different form factors and capabilities. We therefore believe that focusing on improving coordination among personal devices is a more promising approach.

## EVALUATION

Evaluating a developer toolkit, framework, or infrastructure typically requires examining both utility and usability. An infrastructure's utility depends on whether developers can use it to build the types of applications they want. In general, the broader the set of applications that developers can build with it, the more utility it provides. We argue that the set of services we presented demonstrates by example that our infrastructure supports the development of many interesting applications.

An infrastructure's usability, on the other hand, is a measure of how easily developers can build applications with it. Rigorous evaluation of the usability of a developer infrastructure is challenging for a variety of reasons: a lack of standard tasks, variation between developers, time-intensive tasks, etc. However, we wanted to explore whether our infrastructure does indeed make it easy to build services that span personal devices. We also wanted to identify areas for improvement. We therefore recruited five programmers at our company to participate in a user study.

All participants had more than 5 years of general programming experience. All also had more than 3 years of Java experience (the development language for the study), but only participant 1 still actively used it (and only on a monthly basis). Participant 4 had no experience with Eclipse (the development environment for the study), while the others had at least 3 years of experience and employed it on a daily or weekly basis. Participants 2, 3, and 5 had previous experience building client-server or distributed computing systems. No participants had previous experience using our infrastructure.

### Tasks

We asked participants to complete three services: a Search Console service that issues search queries to other devices and displays returned results, a Search service that runs requested searches and returns results, and a Contacts service that provides a consistent address book across a user's devices. We did not have participants create the services from scratch because we were uninterested in whether users could implement the local functionality for them (e.g., creating user interfaces, running searches on a local device). We instead chose to provide participants with pre-built skeletons for the services: partial implementations of the service classes that participants had to complete by adding methods to send information to other devices and methods to process information coming from them.

We divided the completion of the services into four tasks:

1. Add functionality to the Search Console service to send out search requests to the user's other devices and to process the returned search results. Participants also had to add functionality to the Search service to take an incoming search request and return the relevant search results (generated by our skeleton code).
2. Add functionality to the Contacts service to send notifications to the user's other devices when the user adds, deletes, or contacts a person (the service keeps track of when and how a user contacts other people). Participants also had to add functionality to process incoming notifications.
3. Extend the Contacts service to add queueing of messages sent to offline devices for later delivery.
4. Extend the Contacts service again to make it keep the most recent version of its information on the server. Participants also had to make the service ask the server for the most recent information when it starts and process the information returned by the server.

### Method

We provided participants with a laptop computer and a desktop computer, each running Windows XP and a PIE client. Participants developed on the laptop, which had Eclipse and the service skeletons installed, and tested on both devices. We also provided participants with written study instructions, a copy of the developer's guide that we wrote for the infrastructure, and a Java reference manual. We instructed participants to work at their own pace and to ask clarification questions when necessary (to help us identify where they experienced difficulties).

## Results

Table 1 shows the task completion times in minutes for each participant. Participants completed all tasks in an average of 147.4 minutes. While the first two tasks are roughly equivalent, participants completed task 2 in one-third of the time that they completed task 1. This substantial decrease in task completion time suggests that two-thirds of the time participants required to finish task 1 was spent learning our infrastructure's conceptual structure, programming model, and API. Our observations reinforce this possibility (e.g., participants typically spent the first 15 minutes skimming the developer guide and referred to it frequently during the first task), as does the faster completion time of participant 5 (who read the developer guide the day before the study).

The completion times for task 3 were minimal because requesting that the server queue messages for offline devices requires the addition of a single line of code; participants spent more time looking through the guide than coding. Task 4, by contrast, took participants almost an hour on average. We believe that participants spent much of this time learning the concepts and the part of the API required to make a service store information on and retrieve it from the server. We suspect that, similar to the decrease in task completion time between tasks 1 and 2, participants would complete a subsequent task similar to 4 much more quickly.

**Table 1: Task Completion Times (Minutes)**

| Participant | Task 1 | Task 2 | Task 3 | Task 4 |
|---|---|---|---|---|
| 1 | 71 | 23 | 3 | 66 |
| 2 | 96 | 20 | 6 | 49 |
| 3 | 77 | 22 | 2 | 63 |
| 4 | 80 | 17 | 5 | 55 |
| 5 | 33 | 19 | 3 | 27 |
| Average | 71.4 | 20.2 | 3.8 | 52.0 |
| Std. Dev. | 23.4 | 2.4 | 1.6 | 15.5 |

We believe that these results demonstrate that our infrastructure does indeed make it easy for developers to create services that span multiple personal devices. With no previous experience using the infrastructure, participants implemented the cross-device functionality for all of the three services in, on average, less than two and a half hours.

We did, however, identify areas for improvement. While the infrastructure made addressing and sending messages easy, participants reported that formatting outgoing messages and retrieving data from incoming messages were still somewhat time-consuming and error-prone. One potential solution would be to use reflection to automatically convert simple classes to and from messages. Improved error handling was another common request. Participants wanted more and better error messages, particularly to help them diagnose silent failures (e.g., why a service was not receiving a message). Creating easy-to-use tools to help developers debug functionality distributed across multiple computers is a potentially valuable area for future research.

## ADDITIONAL LESSONS LEARNED

While our user study helped us assess the usability of our infrastructure and identify areas that need additional improvement, we also learned a few additional lessons in the process of implementing our sample services:

- Our loosely-coupled approach to service interoperation (i.e., allowing services to exchange information by directing messages using XML namespaces rather than named services) made it easy for us to reuse services for purposes we had not originally intended. We believe that our infrastructure's support for such reusability will be particularly valuable as developers begin to build more multi-device services and identify additional functionality that the infrastructure can or should provide, because any developer can create a service that extends the infrastructure by providing higher-level functionality to other developers' services.
- Some services fit a peer-to-peer model where they only use the central server to route messages, while others are easier to build when they store most or all of their data on the central server. Our experience suggests that an infrastructure should support both models and let developers choose which to use.

## CONCLUSIONS AND FUTURE WORK

Users increasingly employ multiple computing devices, but the user experiences currently provided by applications when users interact across those devices leaves significant room for improvement. One of the barriers to improving those experiences is the amount of additional effort required to implement the functionality necessary for an application to communicate across a user's devices. We contribute an instant messaging-based infrastructure that makes it easy to add that functionality. Our infrastructure:

- Extends previous work by elevating device ownership to a first class property, allowing developers to provide functionality that spans personal devices without writing code to identify and manage users' devices or establish connections among them.
- Provides simple mechanisms for services to send information, events, or commands between a user's devices.
- Provides simple access control by blocking messages sent to services by other users' devices unless a service explicitly asks to receive them.
- Incorporates a number of refinements, such as default and customizable aliases for sending messages to multiple devices and support for extending the client's user interface, that we identified by building over twenty services with it.

- Supports service composition, particularly through loose coupling between services, and allows developers to structure communication between services using a peer-to-peer, centralized, or hybrid approach.

We evaluated both the utility and usability of our infrastructure. We demonstrated its utility by presenting a subset of the services we built with it. We demonstrated its usability by presenting a study of five developers that used it to implement the cross-device functionality for three services. Although none of the developers had previous experience using our infrastructure, they were able to implement all of the required functionality in less than two and a half hours on average.

Our next steps are to study our infrastructure and sample services more broadly. We recently released our infrastructure and sample services to our internal early adopter and developer communities. We intend to study which services users employ, which they avoid, and why, with the goals of improving the general multi-device interaction experience and of identifying specific opportunities for new services. We also intend to examine how developers employ the infrastructure to identify additional opportunities to improve it.

## REFERENCES


1. Ahn, J. and Pierce, J.S. SEREFE: Serendipitous File Exchange Between Users and Devices. In *Proceedings of Mobile HCI 2005*, pp. 39-46.
2. Bardram, J. Bunde-Pedersen, J., and Soegaard, M. Support for activity-based computing in a personal computing operating system. In *Proceedings of CHI 2006*, pp. 211-220.
3. Beagle. http://beagle-project.org/Main_Page.
4. Caceres, R., Carter, C., Narayanaswami, C., and Raghunath, M.T. Reincarnating PCs with Portable SoulPads. In *Proceedings of ACM/USENIX MobiSys 2005*, pp. 65-78.
5. Dearman, D. and Pierce, J. "It's on my other computer!": Computing with Multiple Devices. In *Proceedings of CHI 2008*, pp. 1144-1153.
6. Extensible Messaging and Presence Protocol. Core, Instant Messaging and Presence. http://www.ietf.org/rfc/rfc3920.txt and http://www.ietf.org/rfc/rfc3921.txt.
7. eyeOS. http://eyeos.org.
8. FolderShare is a Windows Live Service. https://www.foldershare.com/
9. Fox, A., Johanson, B., Hanrahan, P., and Winograd, T. Integrating Information Appliances into an Interactive Space. *IEEE Computer Graphics and Applications*, 20, 3 (May/June 2000), pp. 54-65.
10. Hutchings, H. and Pierce, J. Understanding the whethers, hows, and whys of divisible interfaces. In *Proceedings of AVI 2006*, pp. 274-277.
11. Johanson, B., Pennekanti, S., Sengupta, C., and Fox, A. Multibrowsing: Moving Web Content across Multiple Displays. In *Proceedings of Ubicomp 2001*, pp. 346-353.
12. Microsoft Office Groove. http://office.microsoft.com/en-us/groove/default.aspx.
13. Newman, M., Izadi, S., Edwards, W.K., Sedivy, J., and Smith, T. User Interfaces When and Where They are Needed: An Infrastructure for Recombinant Computing. In *Proceedings of UIST 2002*, pp. 171-180.
14. Olsen, D.R., Nielsen, S.T., and Parslow, D. Join and capture: a model for nomadic interaction. In *Proceedings of UIST 2001*, pp. 131-140.
15. Oulasvirta, A. and Sumari, L. Mobile kits and laptop trays: managing multiple devices in mobile information work. In *Proceedings of CHI 2007*, pp. 1127-1136.
16. Pering, T., Ballagas, R., and Want, R. Spontaneous marriages of mobile devices and interactive spaces. *Communications of the ACM*, 48, 9 (2005), pp. 53-59.
17. Raghunath, M., Narayanaswami, C., and Pinhanez, C. Fostering a Symbiotic Handheld Environment. *IEEE Computer*, Sept. 2003, pp. 55-65.
18. Richardson, T., Stafford-Fraser, Q., Wood, K.R., and Hopper, A. Virtual Network Computing. *IEEE Internet Computing*, 1, 1-2, (Jan./Feb. 1998), pp. 33-38.
19. Roman, M., Hess, C., Cerquiera, R, Ranganathan, A, Campbell, R., Nahrstedt, K. A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing*, 1 (2002), pp. 74-83.
20. Satyanarayanan, M., Kistler, J., Kumar, P., Okasaki, M., Siegel, E., and Steere, D. Coda: A Highly-Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39, 4 (April 1990), pp. 447-459.
21. Satyanarayanan, M., Kozuch, M., Helfrich, C., and O'Hallaron, D.R. Towards Seamless Mobility on Pervasive Hardware. *Pervasive & Mobile Computing*, 1, 2 (July 2005), pp. 157-189.
22. Schilit, B. N. and Sengupta, U. Device Ensembles. *IEEE Computer*, 37, 12 (Dec. 2004), pp. 56-64.
23. Synergy. http://synergy2.sourceforge.net/.
24. Tandler, P. The BEACH application model and software framework for synchronous collaboration in ubiquitous computing environments. *Journal of Systems and Software*, 69, 3 (January 2004), pp. 267-296.
25. Tang, J.C., Lin, J., Pierce, J.S., Whittaker, S., and Drews, C. Recent Shortcuts: Using Recent Interactions to Support Shared Activities. In *Proceedings of CHI 2007*, pp. 1263–1272.
26. Yin, M. and Zhai, S. Dial and see: tackling the voice menu navigation problem with cross-device user experience integration. In *Proceedings of UIST 2005*.