Composable Ad-hoc Mobile Services for Universal Interaction

Todd D. Hodes, Randy H. Katz, Edouard Servan-Schreiber, Lawrence Rowe
Computer Science Division
University of California at Berkeley
(hodes,randy,edss,larry)@cs.berkeley.edu

August 2, 1997

Abstract

This paper introduces the notion of "universal interaction," allowing a device to adapt its functionality to exploit services it discovers as it moves into a new environment.

Users wish to invoke services — such as controlling the lights, printing locally, or reconfiguring the location of DNS servers — from their mobile devices. But *a priori* standardization of interfaces and methods for service invocation is infeasible. Thus, the challenge is to develop a new service architecture that supports heterogeneity in client devices and controlled objects, and which makes minimal assumptions about standard interfaces and control protocols.

There are five components to a comprehensive solution to this problem: 1) allowing device mobility, 2) augmenting controllable objects to make them network-accessible, 3) building an underlying discovery architecture, 4) mapping between exported object interfaces and client device controls, and 5) building complex behaviors from underlying composable objects.

We motivate the need for these components by using an example scenario to derive the design requirements for our mobile services architecture. We then present a prototype implementation of elements of the architecture and some example services using it, including controls to audio/visual equipment, extensible mapping, server autoconfiguration, location tracking, and local printer access.

1 Introduction

Researchers have predicted that wireless access coupled with user mobility will soon be the norm rather than the exception, allowing users to roam in a wide variety of geographically distributed environments with seamless connectivity [37].

This *ubiquitous computing* environment is characterized by a number of challenges, each illustrating the need for adaptation: continuously available but varying network connectivity, with high handoff rates exacerbated by the demands of spectrum reuse; variability in end clients, making it necessary to push computation into the local infrastructure; and variability in available services as the environment changes around the client.

MOBICOM 97 Budapest Hungary

This paper investigates novel uses of a ubiquitous network, focusing on variable network services in the face of changing connectivity and heterogeneous devices. We propose that providing an "IP dialtone" isn't enough; we must add additional service infrastructure to augment basic IP connectivity. Specifically, we describe an architecture for *adaptive* network services allowing users and their devices to control their environment.

The challenge in the design is developing an *open* service architecture that allows heterogeneous client devices to discover what they can do in a new environment, and yet which makes minimal assumptions about standard interfaces and control protocols.

The key elements of the architecture we have developed include: 1) augmented mobility beacons providing location information and security features, 2) an interface definition language allowing exported object interfaces to be mapped to client device control interfaces, and 3) client interfaces that maintain a layer of indirection, allowing elements to be remapped as server locations change and object interactions to be composed into complex behaviors.

Additionally, we have designed, implemented, and deployed in our Computer Science building the following example services:

- untethered interaction with lights, video and slide projectors, a VCR, an audio receiver, and an A/V routing switcher from a wirelessly connected laptop computer;
- automatic "on-the-move" reconfiguration for use of local DNS, NTP, and SMTP servers; HTTP proxies; and RTP and multicast-to-unicast gateways;
- audited local printer access;
- interactive floor maps with a standardized interface for advertising object locations;
- tracking of users and other mobile objects with privacy control.

The testbed for our experiments [18] includes Intel-based laptop computers with access to a multi-tier overlay network including room-sized infrared cells (IBM IR), floor-sized wireless LAN cells (AT&T WaveLAN), and a wide-area RF packet radio network (Metricom Richocet). We also leverage facilities in a seminar room augmented with devices that can be accessed and controlled through a workstation. The physical components of the testbed are illustrated in Figure 1.

Our infrastructure builds on the substantial work in mobility support provided by the networking research community. The Mobile-IP working group of the IETF [24] has made great strides in the routing aspects of the problem. Overlay networking [31] has demonstrated the feasibility of seamless handoff between Internet service providers and interfaces. The developing Service Location Protocol

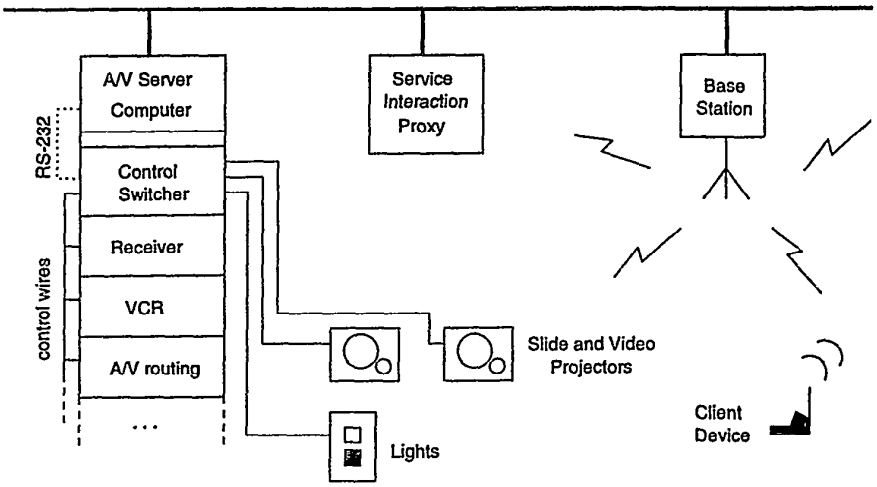

Figure 1: Project operating environment

[34] addresses resource discovery and management. Such efforts have been instrumental in motivating this work.

The rest of this paper is structured as follows. In Section 2, we discuss the key problem characteristics and provide a framework for a service provision architecture's core functionality. This is motivated by a scenario with a set of high-level functional requirements to be achieved. In Section 3, we detail our architecture's prototype implementation and the protocols that allow mobile clients to access the infrastructure. In Section 4, we describe the suite of example ad-hoc mobile services incorporated into it. In Section 5, we unify the design and implementation with some discussion of interrelationships, dependencies, and a layered view of the components. In Section 6, we discuss the relevant related work. In Section 7, we summarize future and continuing work, and finally in Section 8, we present our conclusions..

2 Designing a Service Interaction Architecture

We motivate our architecture for mobile services with the following scenario:

> You are on your way to give an invited lecture.
>
> After parking on campus, you take out your PDA with wireless connectivity, checking the list of local services available to you. You click on the map icon, and are presented with a campus-wide map that includes a rough indication of where you are. You select "Computer Science Division" from a list, and a building near you is highlighted. You walk toward it.
>
> As you enter the building, you glance down at your client device and the list of available services changes. Additionally, the campus map is replaced with the building floorplan.
>
> Using the new map, you find and enter the lecture hall. In preparation for your talk, you select "audio/visual equipment" and "lights" from the list of services, causing a user interface for each to appear. Your selections also cause the rooms' equipment to be located on the floorplan. You walk to the VCR and insert a tape.
>
> The lecture begins. As you finish the introduction, you dim the lights and start the VCR with taps on your PDA. At that moment, you realize you've forgotten your lecture notes. Using your personalized printer user interface, you retrieve a file from your home machine and instruct the closest printer to print the file. The printer's location appears on the floorplan.
>
> A minute later, you are notified the print job has completed, retrieve your printout, and return to finish the lecture as the videotape completes.

From this scenario, we can derive a set of required or desirable functions, presented in the next subsection.

2.1 Requirements Analysis

The problem is that users wish to invoke services — such as controlling the lights, printing locally, or reconfiguring the location of DNS servers — from their mobile devices. But it is difficult to obtain wide-spread agreement on "standard" interfaces and methods for such service invocation. The challenge is to develop an open service architecture that allows heterogeneous client devices to discover what they can do in a new environment, making minimal assumptions about standard interfaces and control protocols.

Implementing such a service architecture makes it possible to turn client devices into "universal interactors." An *interactor* is, broadly, a device that allows a user to interact with and modify his or her environment. Examples include electronic equipment remote controls and thermostats. A *universal interactor* is a device

that adapts itself to control many devices — if it can discover their control interface. A universal interactor thus exploits what it finds in the environment, and varies its abilities as a function of location. It is not a particular hardware component, but instead a way of *using* an existing device.

Realizing such a capability requires (at least) five technical components: 1) device mobility, 2) network-accessible controllable objects, 3) an underlying discovery architecture, 4) mapping between exported object interfaces and client device control interfaces, and 5) composing complex behaviors from underlying primitive objects. These are now described in detail in the following subsections.

2.2 Device Mobility

A critical component of the scenario is device mobility. The client moves from a wide-area network to a local-area network, and between points in the local-area.

This functionality is available through Mobile-IP [24] and network overlays [17]. The former supplies IP-level transparency to changes in location, and the latter augments this functionality with a policy layer for managing connectivity to multiple available network interfaces and a mechanism for seamless (low-latency) hand-off. We build upon this network-layer functionality directly.

On top of this, we only require the ability to detect changes in connectivity with an event-delivery mechanism. Such a mechanism is required to implement automatic reconfiguration: when the client device discovers it has moved, it should check (or be notified) if a local instantiation of a remote service is available, and should autoconfigure to use the local service in this case. Concrete examples include DNS, NTP, and SMTP.

2.3 Controllable Objects

Most objects can be controlled. Doors and windows open; lights turn on; coffee-makers brew. Most physical objects provide only manual controls. A *controllable object,* on the other hand, is one that exposes the interface to which it responds to control requests or transmits status information. Additionally, it makes this interface accessible over a network.

To fit into our architecture, it is crucial that objects be augmented with an ability for network-based control. Open issues include addressability, naming, and aggregation of objects into a controllable unit. Individual controllable objects may be too numerous or the expense of individual control may be too high. For example, while it is possible to make every lightbulb its own controllable object, the sheer number of them in a typical building, the expense of assigning processing to each one, the difficulty of wiring each to the network, etc., would mitigate such a decision. Instead, control functionality could be assigned to a bank of lights, and what is augmented is the switch bank rather than all of the individual lightbulbs. In general, this means that the current infrastructure for naming – DNS – must be extended to include objects that do not have (or need) IP addresses. An alternative is to develop a separate infrastructure to match this need rather than overloading DNS. In the latter case, we can take advantage of the fact that instantiations of these name servers need only have a local, rather than global, scope.

Another approach for interacting with objects is to use video capture augmented with image processing ("computer vision") where applicable. Example uses of this approach include fine-grain object tracking, directionality sensing, and event triggers keyed to particular circumstances [22]. E.g., a camera can be used to detect the opening of a door or window. In this case, it is the camera that exports the control interface.

2.4 Resource Discovery

The function of a resource discovery protocol is to maintain dynamic repositories of service information and make this information available through scoped attribute queries. In contrast with DNS, the repositories' information is specifically local in nature.

We couch our discussion of resource discovery in the context of the Service Location Protocol [34], under development by the IETF Service Location working group. Although there are open issues in this domain, we avoid duplicating much of the relevant discussion here. Interested readers are pointed to the Internet draft and the Service Location working group mailing list.

From our local-area network perspective, the only mechanism we require is a function to allow mobiles to query the server for a mapping from strings to strings. We describe our own mechanisms for finding the correct local server and initializing the string mappings. Finding the a correct local server is similar to delivering the correct SCOPE attribute to the mobile host in SLP.

2.5 Transduction Protocols

A transduction protocol maps a discovered object interface to one that is expected by a given client device. It supports interoperability by adapting the client device's interface to match the controllable object's interface.

The issue with transduction protocols is how to map control functions into a UI supported by the portable device. As an example, assume a client device has a two-position switch widget for use with the local light controller. At a visited location, the light controller supports continuous dimming. In this case, the client may substitute a slider widget for the switch. If it cannot do this (or chooses not to), then the purpose of the transduction protocol is to map the on/off settings of the UI to one of the two extremes of the actual dimmer control.

Our solution is to transfer an entire GUI to the client in a language it understands, and when possible, augment the GUI with an interface description that starts with base data types and allows them to be extended hierarchically. A transducer that doesn't understand a level in the hierarchy can use elements below it. Alternatively, the interface description can be used directly to generate a rough GUI when no language implementation appropriate for the client is available.

The interface descriptions not only allow for data type transducers between client and server; they also provide a critical layer of indirection exactly where it is needed: underneath the user interface, allowing widgets to be transparently remapped to new servers in a new environment. This function is required to allow custom user interfaces for ad-hoc services, such as allowing a virtual "light switch" on the client device's control panel to always control the closest set of lights.

2.6 Complex Behaviors

Objects have individualized behaviors. We wish to couple and compose these individual behaviors to obtain more complex behaviors within the environment. For example, consider a scenario where

music follows you as you move around a building. One behavior of the sound system is to route music to specific speakers. A behavior of location tracking services is to identify where specific objects are located, such as the user. A "complex" behavior allows us to compose these more primitive behaviors of sound routing and location tracking to obtain the desired effect of "following" music.

A key problem is that there is no common control interface for individual components. Furthermore, some behaviors may require maintenance of state that is independent of both subcomponents. An example of the latter is instructing the coffee maker to brew only the first time each morning that the office door opens. Another issue is the policy-level difficulty implied by this scenario: resolution of incompatible behaviors. If another user considers music to be noise, the visiting user's music may or may not be turned off in their presence, depending on seniority, social convention, explicit heuristics, or otherwise. At a minimum, the system must guarantee that it will detect such incompatibilities and notify the user(s) involved in order to avoid instability (e.g., music pulsing on and off as each individual behavior is interpreted).

Our solution is to use *interface discovery,* (i.e. any method through which new objects' input/output data types are learned) paired with the aforementioned data type transducers to allow objects to be cascaded much like UNIX pipes to achieve the desired complex behaviors. Additionally, we allow intermediate entities ("proxies") to maintain state that is independent of the constituent subcomponents. This allows for the incorporation of such features as conditional statements and timing information.

In our prototype, complex behaviors are written as scripts invoked by the delivery of particular events. These events are generated (when necessary) by the data type transducers that translate between the client user interface invocations and the RPC commands sent to a service daemon.[1]

3 Implementing Service Interaction

This section describes implementation details of the service interaction proxy (SIP), the service interaction client (SIC), and beaconing daemon (beacond) programs. These prototypes implement selected components of our overall mobile services architecture.

The prototypes allows a mobile host to enter a cell, bootstrap the local resource discovery server location, and acquire and display a list of available services. They also allows users to maintain a database of scripts to be executed when particular services are discovered for use in autoconfiguration, local state updates, and to trigger location-dependent actions.

If a user wishes to use a service it does not understand, the client first automatically searches its local cache for an interface to that service; if it is not there, the infrastructure is automatically notified and it attempts to send an interface description and GUI to the client.

3.1 Setup

A single copy of the "service interaction client" (SIC) program runs at each client device. Copies of the "server interaction proxy" (SIP) program run at domain-specific granularities. For example, a set of base stations in geographic proximity could be associated with a single SIP. Beaconing daemons (beacond) run at each base station.

An example SIC screenshot is shown in Figure 2. SIP and beacond use configuration files and command-line switches, and thus user interfaces are not shown.

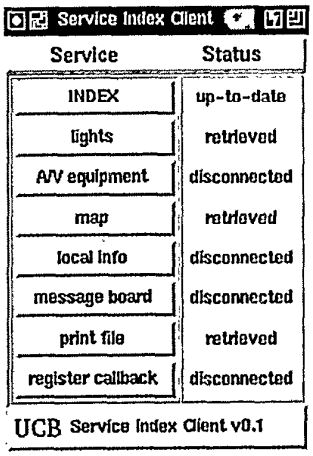

Figure 2: The SIC application GUI is currently a series of buttons that can be used to retrieve and invoke application interfaces.

Each SIP process maintains a database of the services and service elements that it provides to mobile hosts. An example startup file for such a database is listed in Figure 3. It contains three types of entries: SERVICES, VALUES, and PROPERTIES. VALUES are used for generic (key, value) lookups. These are useful for, e.g., detecting the need to update server addresses. SERVICES and PROPERTIES are used to specify what, where, and how services are available from that particular location. Each SERVICE has a unique name, and maintains PROPERTIES such as the version number, a pointer to an associated IDL file[2], pointers to particular language implementations of user interfaces for the service, and the geographic location (if any) for use with maps. VALUES and PROPERTIES may just be pointers to another SIP, allowing simple incremental deployment to subdomains and yielding a notion of topology.

3.2 Message-level Detail

The client enters a cell with a beaconing daemon. The daemon sends periodic broadcasts that contain the bootstrap address and port number of that cell's SIP. The client automatically registers with the base station to establish IP connectivity. It then requests the well-known meta-service INDEX, which returns a list of the services available. Based on the contents of the reply, the client renders labelled UI buttons for unknown services, remaps the location of running services, and executes scripts in a database to enable autoconnection

[1] Thus, even in the case where no translation is necessary, a null transducer must be interposed in order to allow detection of invocations. In other words, the transduction layer is the layer that provides the indirection.

[2] Use of the Interface Definition Language (IDL), a generic format for service interfaces similar in concept to a model-based UI, is described in Section 3.6

```
set NAME {
        Soda 405: High-Tech Seminar Room
}
set SERVICES {
        INDEX lights {A/V equipment} map printer {location tracking}
}
set VALUES {
        DNS {128.32.33.24 128.32.33.25}
        NTP {orodruin.cs.berkeley.edu barad-dur.cs.berkeley.edu}
        SMTP {mailspool.cs.berkeley.edu}
        ...
}
set PROPERTIES {
        lights {IDLfile ../helpers/lights.idl version 0.01 \
                location {132 210} appName-tk ../helpers/lights.tk \
                appArchive-tk ../helpers/405/lights405.tar.uue
                appName-tcl ../helpers/lights.tcl \
                appArchive-tcl ../helpers/405/lights405tcl.tar.uue}
        {A/V equipment} {IDLfile ../helpers/htsr.idl location {132 180} \
                version 0.01 appName-tk htsr.tcl \
                appArchive-tk "../helpers/405/HTSR.tar.uue"}
        ...
}
```

Figure 3: An abridged SIP services database example

and composed actions.[3] When a user requests a particular service, the client software checks its local cache of applications. If an interface supporting the requested application is not there, it asks the SIP for the service's "properties." This is a list of available interface descriptions and/or implementations. It also receives any service metadata (such as version numbers). It then chooses either to download a particular interface implementation (e.g., as a Java applet) or the generic interface description. The SIC then unpacks the received archives, transduces the interface description to match the device characteristics, and finally executes the GUI.

An example exchange of protocol messages for a client moving between SIP servers is illustrated in Figure 4.

3.3 Bootstrap

For a client to use services, it must first find the address of the local resource discovery server. In our architecture, this bootstrap above IP is minimal: there is an indirection embedded in the mobility beacons. This minimal bootstrap standardizes the interface for sending service advertisements without constraining the item to which it points. In general, it could point to any type of name server, thereby allowing variation in resource discovery protocols if this were desired.

3.4 Beaconing

Beaconing is required in a system to facilitate notification of mobility-based changes in the relative position of system components. Its use is motivated by inherent availability of physical-level hardware broadcast in many cellular wireless networks and the need to track mobiles to provide connectivity.

[3]The database currently resides on the client, but could additionally be retrieved from elsewhere by a proxy server to address client computational limitations.

Two issues arise once the decision to beacon has been made. The first is which direction to send them: uplink or downlink. The second is what information to put on the beacons, if any at all. (An empty beacon acts as a simple notification of the base station address, available in the packet header.) These are discussed in the following subsections.

3.4.1 Beaconing Direction

In terms of choosing whether to have client devices or infrastructure servers beacon, existing systems can be found which have made either choice. Client beaconing is used in both the Active Badge [13] and PARCTAB systems [26], while server beaconing was used in Columbia Mobile IP [14]. IETF Mobile IP utilizes both periodic advertisements *and* periodic solicitations.

One might expect the application-level framing [9] argument to hold here: different policies optimize for different applications' operating modes. This is indeed the case: there are trade-offs in such a decision, as it varies allowances for privacy, anonymity, particular protocols' performance, and scalability.

Specifically, some benefits of base station beaconing include:

- less power is consumed at the mobile by periodically listening than by periodically transmitting;
- finding a base station requires only a single message rather than a broadcast/response pair;
- mobiles need not transmit to detect when contact is lost;
- detection of multiple beacons can be used to assist handoff;
- anonymity of location is preserved for non-transmitting mobiles;
- allows possibility of "anonymous" access to some data known to the infrastructure (at a cost of management overhead and increased beacon size due to the piggybacking);

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.