ICrafter : A Service Framework for Ubiquitous Computing Environments

Shankar R. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd

Stanford University

Abstract. In this paper, we propose ICrafter, a framework for services and their user interfaces in a class of ubiquitous computing environments. The chief objective of ICrafter is to let users flexibly interact with the services in their environment using a variety of modalities and input devices. We extend existing service frameworks in three ways. First, to offload services and user input devices, ICrafter provides infrastructure support for UI selection, generation, and adaptation. Second, ICrafter allows UIs to be associated with service patterns for on-the-fly aggregation of services. Finally, ICrafter facilitates the design of service UIs that are portable but still reflect the context of the local environment. In addition, we also focus on the system properties such as incremental deployability and robustness that are critical for ubiquitous computing environments. We describe the goals and architecture of ICrafter, a prototype implementation that validates its design, and the key lessons learnt from our experiences.

1 Introduction

In this paper, we propose ICrafter: a service framework for a class of ubiquitous computing environments known as *interactive workspaces* [6]. An interactive workspace is a physically co-located, technology-rich space consisting of interconnected computers (desktops, laptops, handhelds, etc), utility devices (scanners, printers, etc) and I/O devices (large wall-mounted and table-top displays, microphones, speakers, etc), where people gather (with their own laptops, handhelds, etc) to do naturally collaborative activities such as design reviews, brainstorming, etc. Example interactive workspaces include conference/meeting rooms and lecture halls.

The main objective of ICrafter is to allow users of interactive workspaces to flexibly interact with the *services* in the workspace. By *service*, we refer to a device (such as a light, projector, or a scanner) or an application (such as a web browser or Microsoft PowerPoint running on a large display) that provides useful functions to end-users. Users interact with the services using a variety of access/input devices (such as laptops, handhelds, etc). We use the term *appliance* to refer to such an access/input device. (In other words, service UIs run on appliances.) ICrafter is a framework that allows developers to deploy services and to create user interfaces to these services for various user appliances.

The design goals of ICrafter stem from our experiences with users in *iRoom*, a prototype interactive workspace at Stanford, which serves as the experimental testbed for our research. Many of these goals arise from fundamental characteristics of ubiquitous computing environments, such as heterogeneity, presence of legacy components, and incremental evolution. We have implemented a prototype of the ICrafter framework in iRoom and have developed several services (and their UIs) using the framework.

Recent years have seen a spate of interest in service and UI frameworks for ubiquitous computing environments. Several industry and research projects have proposed various frameworks such as Jini [9], UPnP [17], Hodes et al. [10, 11], and Roman et al. [12]. ICrafter extends the existing work in three important ways:

1. ICrafter places intelligence in the infrastructure to select, generate, and/or adapt service UIs. This helps offload services and appliances and has several advantages such as extensibility, and better handling of legacy services and resource-limited appliances.
2. ICrafter provides a novel scheme for "on-the-fly" aggregation of services.
3. ICrafter facilitates creation of UIs that are portable across workspaces but still reflect the context of the current workspace.

The rest of the paper presents the goals, architecture, and implementation of ICrafter. We also present several examples that illustrate its use and the lessons we have learned from our experiences.

2 Design Goals

The objectives of the ICrafter framework may be separated into three overarching goals: *adaptability, deployability,* and *aggregation.* We believe that the first two goals are not specific to this framework alone but are central to ubiquitous computing in general, because heterogeneity, legacy components, and incremental evolution are the norm rather than the exception in these environments.

2.1 Adaptability

Interactive workspaces are characterized by heterogeneity at both the appliance-level and the workspace-level. (The workspaces themselves maybe widely different depending on their physical geometries and which sets of devices they contain.) Thus, the framework should facilitate adaptation to these two types of heterogeneity.

Appliance adaptation. The framework should not only support several modalities (e.g. a gesture-based UI or a voice-based UI), but also different appliances with the same modality (e.g. a handheld computer vs. a pen-and-tablet form factor computer vs. the screen of a user's laptop). Also, appliances can vary widely in resources.

Workspace adaptation. The framework should allow the generated UIs to include *contextual information relevant to that workspace*, which helps the user identify the association between the UI elements and the environment. This association may be established by human readable descriptions (e.g., a projector control interface that says "projector for middle screen"), layout of interface elements (e.g., a light interface that positions widgets to indicate the actual physical positions of corresponding lights in that space), or by some other means. The challenge is to facilitate the design of UIs that can be reused across workspaces. It is impractical to hand-design UIs for common services at each installation for every appliance.

2.2 Deployability

For the framework to be easily deployable and evolvable, it must satisfy the following requirements.

Flexibility in language/ OS/ UI toolkit support. The system should not force the use of particular programming languages, UI languages, or operating systems, since it is unlikely that a single programming language and/or UI language will work for all devices in the near future. Also, supporting off-the-shelf UI renderers such as web browsers is essential.

Spectrum of UIs. It is impractical to manually design UIs for all services for each appliance, especially considering the incremental evolution inherent in this environment. When a new service is added, ideally it should be accessible without the necessity of having to manually design UIs for every appliance. Similarly, adding a new appliance must not force writing UIs for that appliance for all existing services. Thus, support for (possibly partial) automatic UI generation is desirable. On the other hand, because effective UI design is an art, the framework should allow the presentation of custom-designed UIs. The framework should therefore support a spectrum of UIs ranging from fully hand-designed custom UIs to automatically generated UIs.

Robustness. A complex system with many components must handle partial failures gracefully for deployment to be practical. Partial failures may be software-related (a service fails, or some component of the UI-generating infrastructure fails) or hardware-related (a physical device fails or stops responding).

2.3 Aggregation

The framework must facilitate the creation of user interfaces for combinations of services. Users often need to control several services simultaneously to perform a task, such as a presentation or a meeting; for example, during a slide presentation, it is convenient to aggregate the lighting controls with the slide show controls (start the slide show, etc.)

To the best of our knowledge, all existing frameworks attach UIs to individual services, which makes it difficult to control groups of services. Of course, an ad-hoc UI can be generated by just naively grouping together all the individual service UIs, but there is more to controlling a group of services than simply controlling each one separately. For example, suppose that a user wishing to take a picture from a camera and print it requests the UI for the camera and the printer. If an ad-hoc union of the camera UI and printer UI were returned, the user would still have to request the camera to take a picture, save it in a temporary location, upload the saved picture to the printer, and request for printing.

Ideally, a UI for the camera-printer combination should “recognize” that the two can be composed in a useful manner (that is, the output of the camera can be sent to the printer) and allow the user to do this as easily as possible. Of course, all existing frameworks allow creating a new specific printer-camera service, which in turn accesses the printer and the camera. However, creating services for every combination of services is impractical (the number of combinations explodes combinatorially). Thus, another goal of ICrafter is service aggregation without necessarily requiring the creation of composite services.

3 Architecture

In this section, we present the ICrafter architecture and explain how it achieves the goals enumerated in the previous section. In the ICrafter framework, user appliances request UIs for services from an infrastructure-centric component called the *interface manager (IM)* as shown in figure 1. When the IM receives a request for UI for one or more services, it first selects one or more *generators* based on the requesting appliance and the service(s) for which the UI was requested. (A generator is a software entity that can generate a UI for one or more services for a particular appliance). Next, the IM “executes” the generators and returns the generated UI to the requesting appliance. To generate the UIs, generators need access to information about the services, appliance, and the workspace context. This information is provided as follows:

- Services beacon [1] their presence, and the beacons include the *service descriptions* (information about the service, such as the operations supported by the service).
- When an appliance requests a UI from the IM, it supplies an *appliance description* that provides information about the appliance (such as number of pixels).

[1] A beacon is a periodic announcement to a broadcast medium which any other entity on the medium can “listen” to.

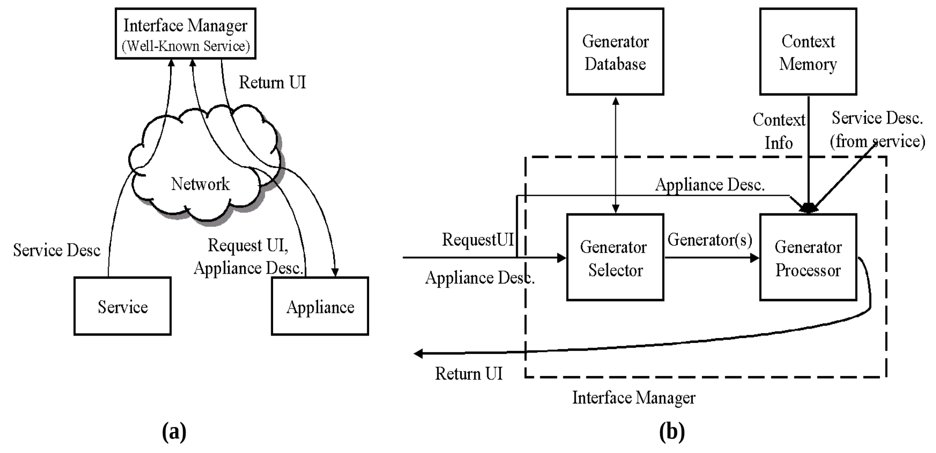


Fig. 1. ICrafter Architecture. Appliances request UIs from the Interface Manager while supplying an appliance description. The Interface Manager first selects appropriate UI generators based on the requesting appliance and the services for which the UI was requested. Next, it executes the generators with access to the service descriptions, appliance description, and the context to generate the UI.

– Information about the workspace context is contained in a central datastore called the *context memory.*

Thus, when the generators are executed in the IM, they are provided access to the service descriptions, appliance description, and the context memory. Next, we explain how we address the design goals laid out in section 2 using the above framework.

3.1 Designing for adaptability

Among existing approaches, the Hodes approach [10, 11] best handles appliance heterogeneity. In this approach, if the service does not supply a predefined UI suitable for the appliance, the appliance-side generates a suitable UI for itself from the service description. However, if the appliance is resource-limited, there is a need for alternatives, such as appliance-side proxies. We apply ideas from previous research in related domains [7] to generalize this approach by allowing "intelligence" to exist in the IM (i.e., a third party other than the service or the appliance) to handle UI selection, generation, or adaptation. This lets the resource-rich, infrastructure-based IM select, adapt, or generate a suitable UI based on the requesting appliance.

We also rely on this level of indirection to handle workspace heterogeneity. We cleanly separate workspace context information from the UIs and store the workspace context information in the context memory. To avoid hard-coding workspace information in the UIs, we stipulate that UI generators (and not UIs) be hand-designed. The generators can access the workspace context information from the context memory and generate the actual UIs. Examples of workspace context information include physical locations and dimensions of various devices (such as lights and displays), descriptive information about the devices (e.g., "Display1" is the "front display", etc), and device relationship information (e.g., "projector2" projects onto display "screen1").

An alternative to a centralized context memory would be to allow each generator to read the workspace context information from its own configuration file. However, this approach requires keeping the configuration files for all generators for the same service synchronized. Furthermore, changes in multiple files may be required if the room geometry changes. These kinds of administration are simplified when such information is centralized. Other advantages of centralizing context are explained in [20].

3.2 Designing for deployability

We accommodate incremental evolution by allowing for a range of generators, ranging from fully custom to fully generic. Fully generic generators are service-independent (but appliance-specific) and can generate a UI for any service for a particular appliance, and they enable rapid integration of new services and appliances. However, generators specific to a particular service class (such as InFocus projectors) can also be written and in the extreme case, they can also be written for a particular service instance. Since the generators are run in the infrastructure by default, they can be resource-intensive if necessary even if the resulting UIs may actually be simple.

Service descriptions in ICrafter contain only the service operations and their parameters. They do not contain machine names, port numbers, URLs, or UI elements unlike previous approaches [11, 12, 17]. This avoids the need for modifying the descriptions on a per-instance basis.

ICrafter is also designed to allow UIs for widely-deployed "legacy" renderers such as web browsers. Generic protocol gateways convert from the renderers' transport protocols (such as HTTP) to ICrafter protocols.

3.3 Designing for Aggregation

As mentioned in section 2, there is a need for creating UIs for combinations of services without necessarily having to create composite services for all combinations of services. Unlike existing frameworks, we allow UIs to be attached to groups of services rather than individual services. However, while useful, this only provides a partial solution, because this still requires creation of UIs (if not composite services) for every combination of services.

We propose a novel approach that exploits *service interfaces* to address this challenge. Here the term interface refers to programmatic interface (as in the interfaces defined by the keyword `interface` in Java), not user interface. It is general practice in object-oriented languages such as Java to have classes implement generic interfaces that represent particular behaviors. Similarly, services can also implement generic interfaces representing particular behaviors. Most devices (such as lights, projectors, etc) can implement a PowerSwitchInterface that contains the methods `poweron` and `poweroff`. Similarly, the printer can implement a DataConsumer interface while the camera can implement a DataProducer interface.[2] We then allow generators to be written for *patterns* of interfaces. For example, a generator can be written for a data consumer-data producer pattern. Such a generator can not only be used by camera-printer combination, but also by a scanner-display combination, a scanner-printer combination, etc.

When a user requests an interface for a camera and a printer, the IM searches for matching generators and finds the consumer-producer generator. Of course, the UI produced by this generator does not allow the user to perform operations specific to the camera or the printer (such as adjusting the zoom). However, these operations can be performed using the individual camera and printer UIs. Thus, the IM can (for example) return a simple aggregate of the producer-consumer UI, the printer UI, and the camera UI.

As another example, a simple generator can be written for one or more PowerSwitchInterface implementing services, that allows all these services to be powered on or off. Suppose the user requests a UI for all the lights in the room and a projector. The UI produced by PowerSwitchInterface generator can allow all the lights and the projector to be turned on with a single action.

4 Implementation

In this section, we describe the prototype implementation of ICrafter framework in the iRoom. The block diagram of the prototype is depicted in figure 2. The EventHeap [5] is a flexible event-based communication system used by all iRoom applications. The EventHeap is conceptually based on the tuplespace/blackboard model espoused by LINDA [1] and is currently implemented using IBM TSpaces [18]. Processes post events to the shared EventHeap and can subscribe to events matching a specified pattern. While the prototype ICrafter implementation uses the EventHeap, it can also be implemented using other communication abstractions such as RPC/RMI or message passing. Similar to the EventHeap, the context memory is also a generic workspace software component that is used by all iRoom applications (not just ICrafter). The context memory is implemented [20] as an XML-based lightweight datastore which allows storing and retrieving workspace context information using an XML-based query language. We do not discuss the implementation details of the EventHeap and the context memory any further in this paper.

As shown in figure 2, when a user requests a UI for one or more services (we explain how this process is bootstrapped later), the user appliance sends a request to the IM (step 1). The IM responds with the appropriate UI (step 2),

[2] Of course, this assumes that the data types are inter-operable. However, such assumptions are not required by the mechanism itself, but by the service interfaces chosen here.

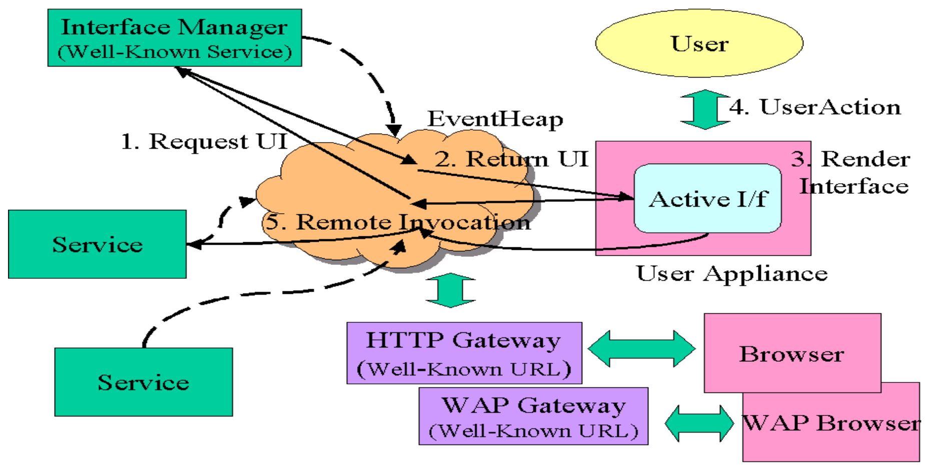


Fig. 2. ICrafter prototype implementation

which is rendered on the appliance by a renderer (step 3). The renderer itself is not part of ICrafter, and can be any native renderer, such as a web browser. User actions on the UI (step 4) result in remote invocations on the target services (step 5). Note that the remote invocations themselves are not mediated by the IM, so there is no slowdown in the user interaction with the UIs.

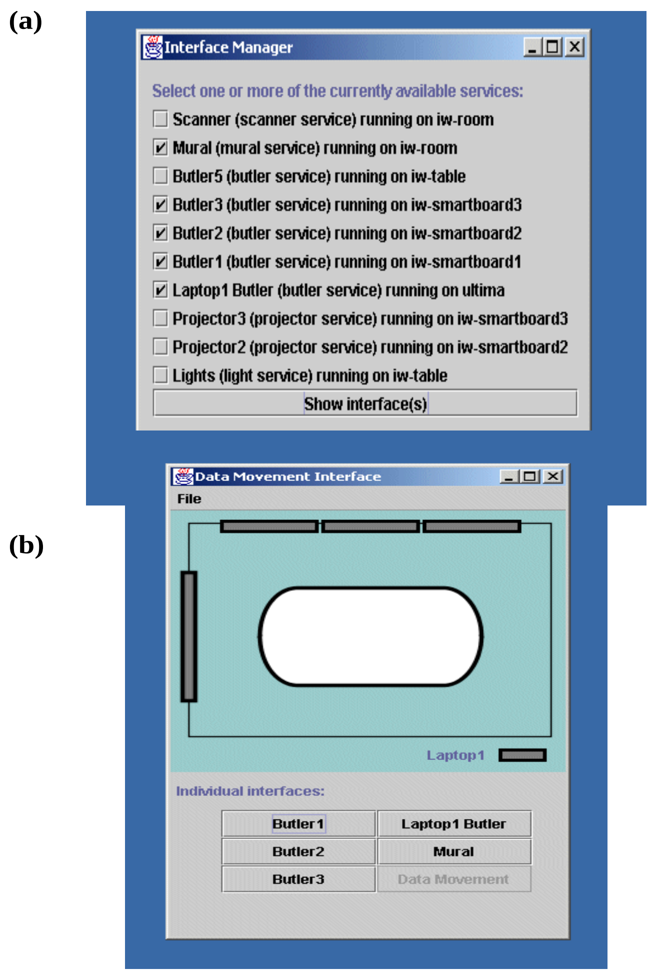
As seen by the application developers for the ICrafter framework, ICrafter is a set of specifications/APIs for developing workspace services and appropriate appliance UIs for them. Thus, to deploy a new service (such as a camera), an app developer needs to write a camera service using the ICrafter service APIs. Similarly, ICrafter provides specifications for an app developer to create UI generators. For example, to create a HTML UI for a service, the app developer needs to create a *HTML template* (which is explained below).

Before describing the implementation specifics, we first illustrate the end-user experience by presenting a walk-through for an example iRoom scenario.

4.1 Walk Through

A user walks into the iRoom with her laptop and starts the SUIML [3] interpreter. The SUIML interpreter is setup to first request the IM (which is a well-known service) for its own (i.e., the IM's) UI. The IM returns its UI which is shown in figure 3(a). The UI of the IM allows the user to select one or more services and request the UI for them. Note that the IM is just another service, and its UI is generated just as for any other service. That is, the IM locates a suitable SUIML generator for itself and executes it to produce the desired SUIML UI.

[3] SUIML (Swing UI Markup Language) is a declarative markup language developed in iRoom for describing GUIs produced by the Java Swing toolkit.



Interface Manager

Select one or more of the currently available services:

- [ ] Scanner (scanner service) running on iw-room
- [✓] Mural (mural service) running on iw-room
- [ ] Butler5 (butler service) running on iw-table
- [✓] Butler3 (butler service) running on iw-smartboard3
- [✓] Butler2 (butler service) running on iw-smartboard2
- [✓] Butler1 (butler service) running on iw-smartboard1
- [✓] Laptop1 Butler (butler service) running on ultima
- [ ] Projector3 (projector service) running on iw-smartboard3
- [ ] Projector2 (projector service) running on iw-smartboard2
- [ ] Lights (light service) running on iw-table

Show interface(s)

Fig. 3. Walk Through Example. The user first sees the IM's UI (part (a)), which allows her to choose one or more services. The returned UI is an aggregate of the UI for each service chosen and the data movement UI. The data movement UI (part (b)) is generated by the generator for "multiple consumers-multiple producers" pattern that has no specific knowledge of the services involved except that they implement the DataConsumer and DataProducer interfaces. The UI shows a top-down view of the room. (The white region is a table which is a fixed landmark in iRoom.) The dark grey rectangles identify the service locations (the three wall-mounted SmartBoards at the top, the mural on the left, and the laptop at the bottom). The user can drag and drop data from any of these services onto another. For example, dragging from the laptop to the middle SmartBoard results in the URL currently displayed on the laptop to be displayed on the middle SmartBoard. The user can also drag a URL from a browser on her laptop onto any of the consumer services. (As expected, dragging from the mural has no effect because it is only a consumer).

While the UI shown here presents a list of services, a different generator could result in (for example) a spatial map of all available services. The user selects the instances of the *butler service* [4] running on her laptop and the three wall-mounted SmartBoards (all the SmartBoards are attached to Windows desktops) and the *mural service* [5].

The UI returned for this request is shown in figure 3(b). Apart from the individual service generators, the generator for the *multiple consumers-multiple producers* pattern also matches at the IM. Notice that the consumer-producer generator has *no* specific knowledge of the butler or the mural, but just uses the information that they implement the DataProducer and DataConsumer interfaces to generate a "data movement UI" that allows data to be moved between these services. This illustrates how aggregation works in ICrafter.

We now proceed to describe the internal workings of the ICrafter implementation in some detail. Throughout this description, we will refer to Figure 4 which shows a very simple example that illustrates the details of the interface generation process.

4.2 Service Descriptions

Our language for describing services is an XML-based language called SDL (Service Description Language). SDL is similar to ISL (Interface Specification Language) in Hodes et al. [11] and the UPnP [17] service descriptions. Figure 4(a) shows part of the SDL for the projector service. Just as ISL, it lists the operations supported by the service. However, unlike previous approaches, SDL does *not* contain addresses/ URLs. For services written in Java, SDL is generated automatically at runtime using Java reflection and included in the service beacons. This avoids the problem of maintaining consistency between services and their descriptions as services evolve. We also provide API calls to service developers that can *optionally* be used to refine the generated SDL.

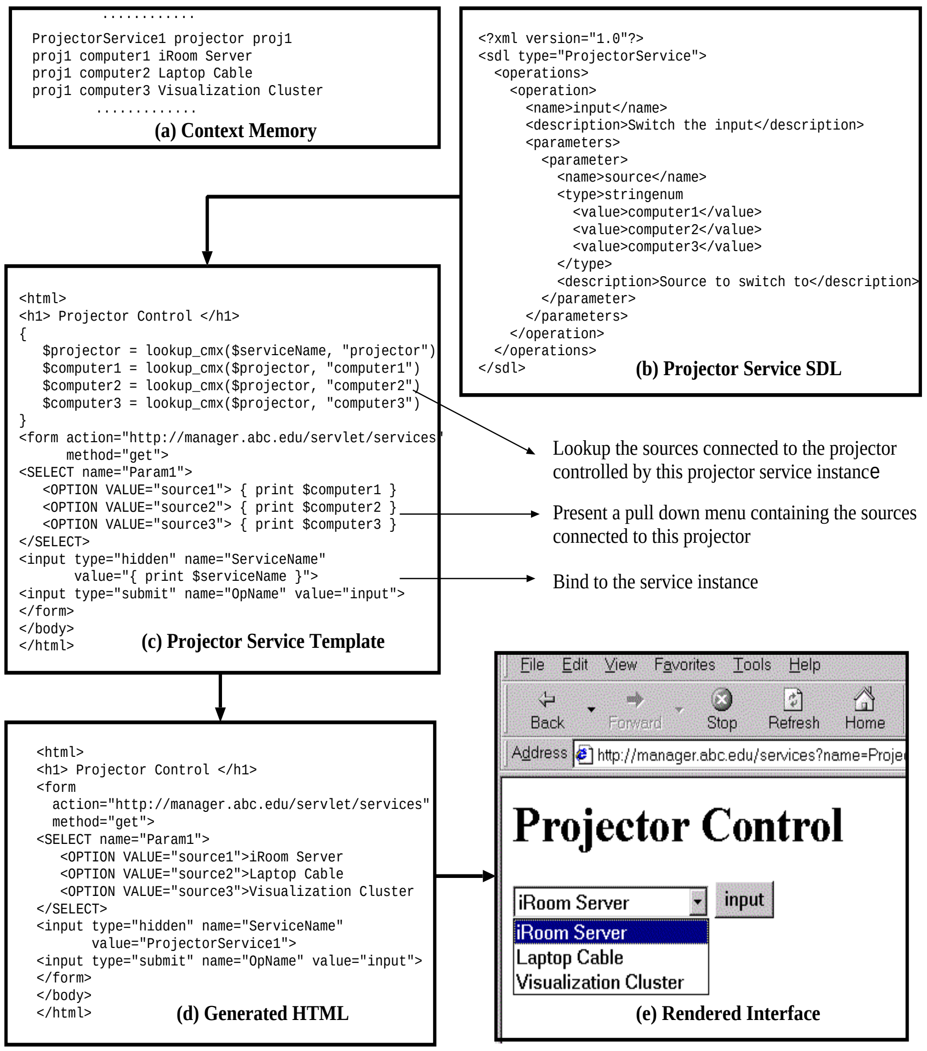
4.3 Discovery and Remote Invocation

Discovery is handled by service beacons. Services beacon their presence by posting short-lived events to the EventHeap, and the IM (or any other entity including the appliances) can query the EventHeap for all (unexpired) beacon events to discover services.[6] Service beacons contain the service descriptions and also a unique name for the service instance that is used for remote invocations on the

[4] The butler service can run on any Windows machine and provides two functions (among others). First, it allows a call event posted on the Event Heap to specify an arbitrary URL to be displayed in a browser running on its host machine. Second, when queried, it can return the URL currently displayed in the top-level browser window on its host machine. Thus, it implements both the DataConsumer and DataProducer interfaces.

[5] The mural service runs on a high-resolution display called the mural and displays data on the mural on-demand. Thus, it implements the DataConsumer interface.

[6] Our current discovery scheme is relatively primitive and is an area of future work.



```
        ............
  ProjectorService1 projector proj1
  proj1 computer1 iRoom Server
  proj1 computer2 Laptop Cable
  proj1 computer3 Visualization Cluster
         .............
```

```
<?xml version="1.0"?>
<sdl type="ProjectorService">
  <operations>
    <operation>
      <name>input</name>
      <description>Switch the input</description>
      <parameters>
        <parameter>
          <name>source</name>
          <type>stringenum
            <value>computer1</value>
            <value>computer2</value>
            <value>computer3</value>
          </type>
          <description>Source to switch to</description>
        </parameter>
      </parameters>
    </operation>
  </operations>
</sdl>
```

```
<html>
<h1> Projector Control </h1>
{
   $projector = lookup_cmx($serviceName, "projector")
   $computer1 = lookup_cmx($projector, "computer1")
   $computer2 = lookup_cmx($projector, "computer2")
   $computer3 = lookup_cmx($projector, "computer3")
}
<form action="http://manager.abc.edu/servlet/services"
      method="get">
<SELECT name="Param1">
   <OPTION VALUE="source1"> { print $computer1 }
   <OPTION VALUE="source2"> { print $computer2 }
   <OPTION VALUE="source3"> { print $computer3 }
</SELECT>
<input type="hidden" name="ServiceName"
       value="{ print $serviceName }">
<input type="submit" name="OpName" value="input">
</form>
</body>
</html>
```

(c) Projector Service Template
Lookup the sources connected to the projector controlled by this projector service instance
Present a pull down menu containing the sources connected to this projector
Bind to the service instance

```
<html>
<h1> Projector Control </h1>
<form
  action="http://manager.abc.edu/servlet/services"
  method="get">
<SELECT name="Param1">
   <OPTION VALUE="source1">iRoom Server
   <OPTION VALUE="source2">Laptop Cable
   <OPTION VALUE="source3">Visualization Cluster
</SELECT>
<input type="hidden" name="ServiceName"
       value="ProjectorService1">
<input type="submit" name="OpName" value="input">
</form>
</body>
</html>
```

Fig. 4. Interface Generation Process. Part (b) shows portion of the projector service SDL with just one operation "input", which allows the source of the projector to be switched among any of the connected computers. Part (a) shows the information relevant to a projector service instance ("ProjectorService1") in the context memory. The projector service template shown in (c) can generate the UI for any projector service instance. It is passed the unique identifier of the projector service instance (whose UI is to be generated) through the variable serviceName. The template looks up the sources connected to the projector controlled by the projector service instance with the given name, and generates a UI that allows the user to select one of them. (The procedure *lookup_cmx* looks up a property in the context memory.) The HTML generated when the template is processed for the service instance "ProjectorService1", and the generated interface in the browser are shown in (d) and (e) respectively.

service instance. Because beacons are marked as short-lived, the EventHeap's event expiry mechanism eventually removes beacon events from failed services. Remote invocations are marshaled into call events and return events (if there is a return). The use of loosely-coupled, semi-persistent event-based infrastructure for communication instead of RMI/RPC (as in Jini and other frameworks) has led to increased robustness. First, certain transient failures of services are easily masked. (A service restarting after a transient failure can still pick up call events directed to it if they have not expired yet). Second, certain components such as the IM can be replicated for fail-over/efficiency such that any event directed to a replicated component is picked by exactly one of the replicas.

4.4 Appliance Descriptions

In our current implementation, we assume that every appliance supports one or more UI languages and we rely on the corresponding UI language interpreter (e.g., web browser for HTML) to render the UI and handle user actions. Thus, appliance descriptions simply consist of the set of UI languages supported by the appliance and optional (name, value) pairs describing the other attributes of the appliance. We have used four different UI languages in ICrafter so far: HTML, VoiceXML, MoDAL [19] (a markup language from IBM Almaden for PalmOS-based devices), and SUIML.

4.5 Generators

Most generators have been implemented using a *template system*. A template is UI markup in one of the supported UI languages with embedded Tcl or Python scripts. (An example template for the projector service is shown in Figure 4(b)). The embedded scripts may use a set of library routines to access the context memory, service description, etc. When a template is executed, the embedded scripts are executed and they are replaced by the output they produce. Custom templates can be written for a service and UI language or a generic service-independent template can be written for a given UI language.

4.6 Generator Database

The generator database lists the language (currently HTML, VoiceXML, MoDAL, or SUIML), platform (currently Tcl, Python, or Java), suitable text description, location, and the associated services/patterns for all the generators. Currently, simplified regular expression-like syntax is used for representing patterns. A generator can be associated with a service instance or a service interface (such as the PowerSwitchInterface) or any pattern of service instances and interfaces. Example patterns include "*all* services that implement the PowerSwitchInterface" and "multiple services implementing DataConsumer and multiple services implementing DataConsumer". Generic generators are marked as service-independent, so that they can match any service. We have had relatively limited experience with pattern based matching for generators so far and we plan to explore it further in the near future.

4.7 Interface Manager

When the IM receives a request for UI for a single service, it first searches for a generator for that service instance, then for that service interface, and finally for the service-independent generator. For a UI request for multiple services, the generator selector first searches for generators for each service (according to the algorithm mentioned earlier). Second, the generator selector searches for all the generators that match any subset of the given set of services. Finally, the selector returns a simple aggregate of all the generators that matched in the first or the second step. In the execution stage, depending on whether the generator is a Tcl/Python template or a Java class, an appropriate processor is chosen to execute it. The execution produces code in a UI language supported by the appliance. (For example, figure 4(c) shows the HTML produced by the execution of the template in (b)).

5 Examples

We present more example service UIs written for ICrafter in this section that highlight various aspects of the system. Our experience so far indicates that template-based UIs are easy to write, because no code needs to be written by the UI developer apart from the simple embedded Tcl/Python scripts. Since our infrastructure provides generic back-end code (such as the HTTP gateway) for converting user actions in any supported UI language to remote invocation events, the UI developer need not write any backend code (such as servlets/CGI for HTML etc). For example, no back-end code was written by the UI developer for the projector service UI shown in figure 4.

Appliance adaptation. Figure 5 shows a simplified illustration of the butler UI for a PalmOS-based device. It offers the same functionality as the SUIML laptop UI for the same service (not shown here but similar to figure 3(b)) but is less convenient because of lack of drag-and-drop in PalmOS. Creating this UI involved only writing the MoDAL markup with the simple embedded Tcl statements for accessing the context memory. The IM automatically picks a suitable UI based on the requesting appliance.

Workspace adaptation. Figure 6 shows the light control SUIML UIs for two different workspaces. Note that the UIs are very different but are *generated by the same template* accessing different context memories. Equipment locations and dimensions are detailed in the context memory; hence, the generated UIs reflect the context without user intervention or changes to code. This illustrates the ability to reuse templates while still creating UIs tailored to the workspace.

Custom and automatic UIs. Figure 7 illustrates the tradeoffs between UIs generated by custom and generic templates. The automatically generated UI doesn't require a UI developer, while the custom designed UI is functionally

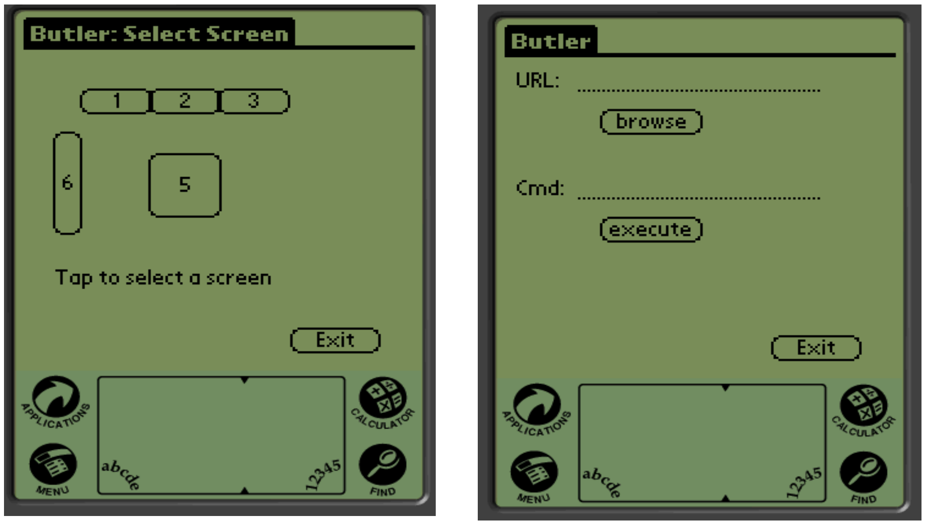


Fig. 5. Global butler control UI using MoDAL on a PalmOS device is shown above. This UI allows users to remotely display a URL on any screen in iRoom using their PalmOS device. Once the user selects a screen using the iRoom top-down view in the left form, the form on the right appears, which prompts the user to enter a URL.

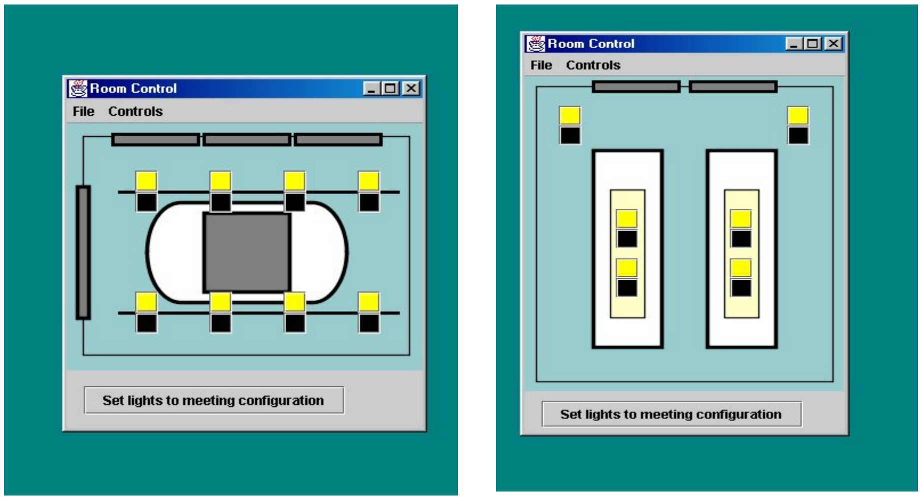
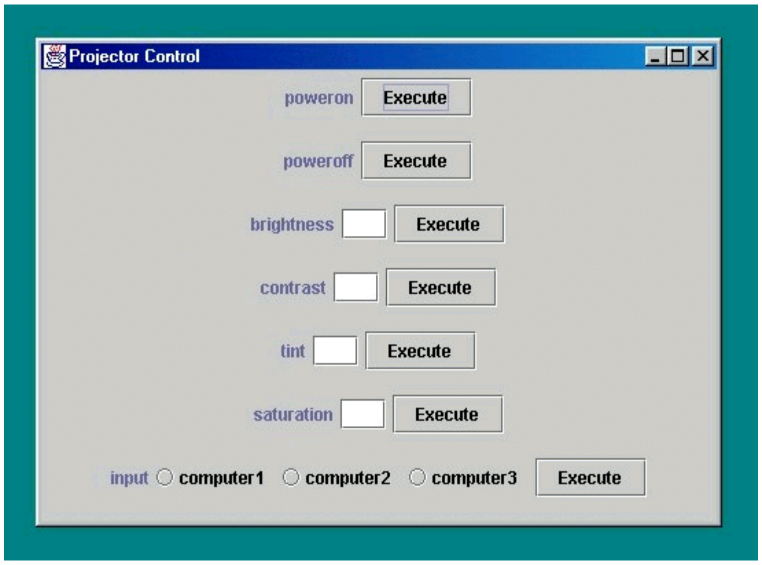

Room Control
File Controls
Set lights to meeting configuration
Room Control
File Controls
Set lights to meeting configuration

Fig. 6. Light control UIs for iRoom (left), which has a table, five displays, and eight lights on ceiling tracks, and another workspace (right), which has two tables, two displays, four overhead lights, and two corner lamps. The user turns a light on using the yellow button corresponding to the light, and turns a light off using the black button. The UIs show top-down views of the workspaces generated from the information in the respective context memories.



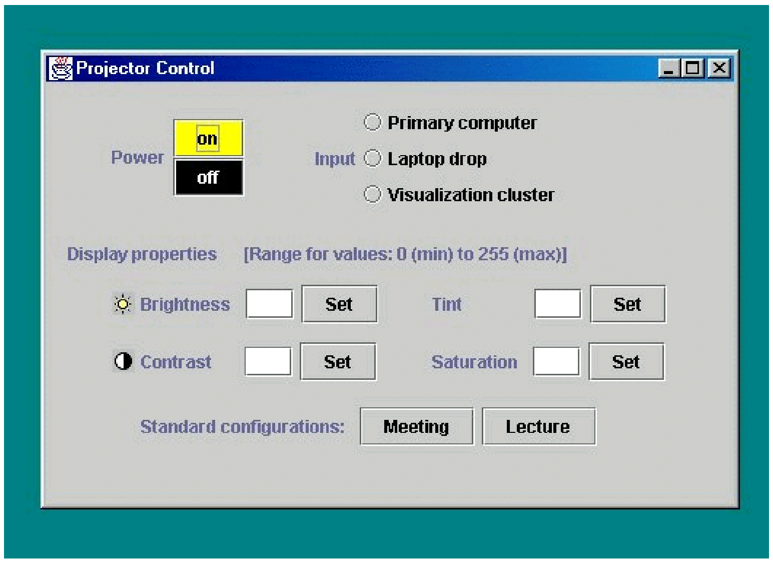

Projector Control
Power on off
Input Primary computer Laptop drop Visualization cluster
Display properties [Range for values: 0 (min) to 255 (max)]
Brightness Set
Tint Set
Contrast Set
Saturation Set
Standard configurations: Meeting Lecture

Fig. 7. Automatic and Custom Generation of UIs. The automatically generated SUIML UI (left) was generated solely based on the SDL using only knowledge of the operations, and the types of each parameter. Though not aesthetically pleasing, it is functional. The custom-designed SUIML UI (right) is superior in two respects. First, it is functionally more convenient. It has descriptive names for projector input sources (such as iRoom server, laptop cable, etc), and has commands for commonly-used settings (standard configuration buttons for meetings and lectures). Second, it is aesthetically pleasing, by using intuitive colors for power switches, easily-recognizable icons where appropriate, and logical grouping of commands by type (power, input, display).

and aesthetically better. Note that the UI developer could use the automatically generated UI as a starting point for the custom UI design.

Voice Interface. Part of an example VoiceXML UI (simplified for illustrative purposes) written for ICrafter is shown below.

```
<form id="projectorcontrol">
  <field name="operation">
    <prompt>
      Please choose a projector operation.
      Available functions are turn on, turn off, switch input.
    </prompt>
    <grammar>
      [turn] on [[the] projector] "turn on"
      | [turn] off [[the] projector]"turn off"
      | (switch | set) input [of [the] projector] "switch the input of"
    </grammar>
  </field>
  <filled>
    You chose to <value expr="operation"/> the projector.
    <submit next="http://server:8080/servlet/voiceservices"/>
  </filled>
</form>
```

The resulting dialog is shown below:

```
COMPUTER: Please choose a projector operation.  Available functions
are turn on, turn off, switch input.
USER (into microphone): Turn on.
COMPUTER: You chose to turn on the projector.
```

Though we can automatically generate UIs for some interaction styles, it is extremely difficult to model speech interactions (e.g., phraseology, for both humans and computers) in a generic fashion, short of full natural language processing. The above UI was thus hand-written by a UI designer, who customized it based on the projector service descriptions and her knowledge of speech interactions. Even though they are hand-designed, it is still easier to create these UIs using the ICrafter template framework since no back-end code needs be written and the Python/Tcl helper routines that provide access to context memory, service and appliance descriptions simplify appliance and workspace adaptation.

6 Related Work

We concur with other researchers' identification of many of our design goals, although we sometimes use different terminology. For example, PIMA [3], Portolano [2], and Borriello et al. [4] also identify many of our design goals as challenges to be addressed by researchers in ubiquitous computing.

Much work has gone into discovery [15–17]. We have not used existing discovery schemes since the simple mechanism based on EventHeap beacons has proven sufficient for us so far, given our smaller scale.

Recent work on generic UI modeling languages is orthogonal to our work. UIML [8] is an appliance-independent XML UI language, which uses style sheets to map the appliance-independent interface description to markup languages or programming languages. Eisenstein et al. [13] present model-based techniques for designing UIs in a platform-independent manner that also adapt to certain types of context . If widely deployed and available, such generic UI languages can be used in ICrafter for writing templates thus simplifying adaptation. We have leveraged some of the recent work in the design of context-aware applications in general and architectures for storing context information [21, 20] to achieve workspace-adaptation in ICrafter. Projects such as EasyLiving [22] and iLand [23] are also investigating software architectures for technology-rich spaces, but they do not explicitly focus on service and UI frameworks.

Among related research and industry efforts, ICrafter is most closely related to Jini [9], UPnP [17], Hodes et al. [10, 11] and Roman et al. [12]. The Jini ServiceUI [14] project allows Java UI code to be registered as a service attribute with the lookup service. Clients are responsible for selecting one of the registered UIs based on which toolkits (such as Swing) the UI code needs. In UPnP (which is a generalization of controlling devices via embedded Web servers), service descriptions include control and presentation URLs which can be accessed by the client using a HTTP-based protocol. In the document-based approach of Hodes et al., service advertisements include a URL where the service documents can be downloaded from. The service documents contain the machine addresses and URLs to download UI code from. If there is no suitable UI, the client can automatically generate its own UI from the service document. Roman et al. propose a "device-independent" representation of services that attempts to capture both service functionality and UI, but the UI elements they use are GUI-centric. As mentioned earlier, ICrafter extends all these four approaches in three significant directions: infrastructure support for UI generation/selection/adaptation, aggregation, and workspace adaptation. In addition, some of these existing approaches allow only certain programming languages and UI languages/ modalities to be used, some others restrict the clients to be web browsers, while yet others do not support web browsers at all.

7 Lessons and Conclusions

We identified the goals for a service framework for interactive workspaces and designed a framework that is compatible with these goals. Although the framework was specifically designed for interactive workspaces, many of the ideas carry over to service frameworks for many other ubiquitous computing environments also. Our framework makes three contributions: infrastructure support for UI selection/adaptation/generation, associating generators with service patterns for on-the-fly aggregation, and workspace adaptation. In addition, our design gives utmost importance to system properties such as incremental deployability, support for legacy components, and robustness which are critically important for ubiquitous computing environments. To validate the design, we implemented a prototype in our experimental workspace, the iRoom, and built several services and appliance UIs for them.

In conclusion, we attempt to abstract away the domain-specific details and examine in retrospect the key design techniques that were used in ICrafter to address the challenges that arise in ubiquitous computing environments. While these techniques are not new, we believe they will serve as useful reminders for ubiquitous computing researchers.

– ICrafter often leverages a "level of indirection" for dealing with heterogeneity. Whenever two sets of variable entities need to inter-operate with one another, a level of indirection is useful. For example, the key for the inter-operation of services and appliances in ICrafter is the existence of intelligence in a resource-rich third party to select and execute appropriate generator(s) from among a range of generators. Similarly, to deal with the inter-operation of UIs and workspaces, we introduce the context memory.
– To accommodate incremental evolution, we follow the guideline: "Provide a reasonable default but allow for fine tuning". For example, default UIs are automatically generated by the generic generators, but custom generators can always be written if desired.
– The use of loosely-coupled, semi-persistent event-based infrastructure for communication instead of RMI/RPC has led to increased robustness. As explained in section 4.3, certain transient failures are easily masked and some components can be easily replicated for fail-over/efficiency.

8 Acknowledgements

We are grateful to Brad Johanson, Susan Shepard, Emre Kiciman, Caesar Sengupta, Meenakshy Chakravorthy, Kathleen Liston, Kathy Richardson, George Candea and Petros Maniatis for their help with this work. We also thank the anonymous reviewers whose comments helped improve this paper. This work was supported in part by the US Department of Energy, Contract B504665.

References

1. David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
2. M. Esler, J. Hightower, T. Anderson, and G. Borriello. Next Century Challenges: Data-Centric Networking for Invisible Computing. The Portolano Project at the University of Washington. In *Proceedings of the Fifth ACM/IEEE International Conference on Mobile Networking and Computing*, pages 256-262, August 1999.
3. Guruduth Banavar, James Beck, Eugene Gluzberg, Jonathan Munson, Jeremy Sussman, and Deborra Zukowski. Challenges: An Application Model for Pervasive Computing. In *Proceedings of the sixth annual international conference on Mobile computing and networking*, pages 266-274, August 2000.

4. Gaetano Borriello and Roy Want. Embedded Computation Meets the World Wide Web. In *Communications of the ACM*, 43(5):59-66, May 2000.
5. Brad Johanson and Armando Fox. The EventHeap: A Coordination Infrastructure for Interactive Workspaces. 2001. *Unpublished draft.* `http://graphics.stanford.edu/~bjohanso/papers/ubicomp2001/eheap_ubicomp.pdf`
6. Stanford Interactive Workspaces Project. `http://graphics.stanford.edu/~iwork/`
7. Armando Fox, Steven D. Gribble, Yatin Chawathe and Eric A. Brewer. Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives. IEEE Personal Communications (invited submission), August 1998.
8. Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, Jonathan E. Shuster. UIML: An Appliance-Independent XML User Interface Language. *Eighth International World Wide Web Conference.* May 1999.
9. Jim Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, pages 76-82, July 1999.
10. T. D. Hodes, R. H. Katz, E. Servan-Schreiber, L. A. Rowe. Composable Ad-hoc Mobile Services for Universal Interaction. *Proceedings of The Third ACM/IEEE International Conference on Mobile Computing (MobiCom '97)*, pages 1-12. September 1997.
11. Todd D. Hodes and Randy H. Katz. A Document-based Framework for Internet Application Control. *2nd USENIX Symposium on Internet Technologies and Systems*, pages 59-70. October 1999.
12. Manuel Roman, James Beck, and Alain Gefflaut. A Device-Independent Representation for Services. *Third IEEE Workshop on Mobile Computing Systems and Applications*, pages 73-82. December 2000.
13. Jacob Eisenstein, Jean Vanderdoncki, and Angel Puerta. Adapting to Mobile Contexts with User-Interface Modeling. *Third IEEE Workshop on Mobile Computing Systems and Applications*, pages 83-92. December 2000.
14. The Jini ServiceUI Project. `http://www.artima.com/jini/serviceui/`
15. S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R Katz. An architecture for a secure service discovery service. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 24-35, August 1999.
16. J. Veizades, E. Guttman, C. Perkins, and S.Kaplan. *Service Location Protocol*, June 1997. RFC 2165. `http://www.ietf.org/rfc/rfc2165.txt`
17. Universal Plug and Play. `http://www. upnp.org/`.
18. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. TSpaces. *IBM Systems Journal*, 37(3):454–474, 1998. 6
19. MoDAL (Mobile Document Application Language). `http://www.almaden.ibm.com/cs/TSpaces/MoDAL/`
20. Terry Winograd. Architectures for Context. *Human-Computer Interaction, 16.* 2001.
21. A. K. Dey, D. Salber, and G. D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction, 16.* 2001.
22. B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer, EasyLiving: Technologies for Intelligent Environments, *Handheld and Ubiquitous Computing 2000 (HUC2K)*, September 2000.
23. Norbert Streitz, Jorg Geibler, and Torsten Holmer. Cooperative Buildings - Integrating Information, Organization, and Architecture. *First International Workshop on Cooperative Buildings (CoBuild 98)*, pages 4-21, February 1998.