User Interfaces for Network Services: What, from Where, and How

Shankar R. Ponnekanti, Luis Alberto Robles and Armando Fox
Stanford University
Computer Science Dept
353 Serra Mall, Stanford, CA 94305, USA
{pshankar,lrobles,fox}@cs.stanford.edu

Abstract

An important problem in the context of network services in ubiquitous computing is the support of ad-hoc interaction. Ad-hoc interaction allows a user entering an environment to discover, request, and interact with user interfaces for the locally available network services, even if she has done minimal or no installation in advance. We observe that most recently-proposed ad-hoc interaction frameworks lack two important mechanisms: distribution and personalization. A distribution mechanism would make it easy to add third-party UI's and to centrally administer UI's across multiple independent workspaces forming an administrative or logical unit, such as all workspaces on a campus. A personalization mechanism would enable a user to see familiar UI's as she roams to different workspaces. We propose extensions to an existing ad-hoc interaction system, ICrafter, that enable these two independent behaviors. The mechanisms raise important policy questions; although we have not studied optimal policies, we outline the policy space and the policies we have adopted.

1 Introduction

As originally envisioned by Mark Weiser [15], future conference rooms, lecture halls, and office environments are expected to consist of numerous hardware and software "network services". These services could be software applications running on public devices (such as a viewer running on a large display) or "smart" utility devices. These utility devices will not be peripherals controlled by device drivers but first-class network citizens capable of speaking standard Internet protocols. We use the term *workspace* to refer to such a physically co-located space consisting of various network services. We begin by describing typical scenarios in future workspaces, loosely based on our laboratory *iRoom*, which is shown in figure 1. Recently proposed service frameworks such as Sun's Jini [2], Microsoft's UPnP [5], ICrafter [10], Hodes et al [9] promise to make such scenarios real in the near future.

Figure 1. The iRoom. Modeled after a future office workspace, it contains several software-controlled utility devices and large displays.

1.1 Motivating scenarios

Jane walks into a workspace with her laptop and double-clicks the "show services" icon. As shown in figure 2(A), a Java Swing [1] UI showing the list of services currently running in the workspace is shown on her screen. She selects the four display services, and clicks the button "Show interface(s)". Using the returned UI (shown in figure 2(B)), Jane simply drags a URL onto one of the displays, causing the web page to be opened on that display. Similarly, she drags two other URL's on to two other displays respectively. She then selects the lights service, and uses the returned UI (shown in figure 2(C)) to turn some of the lights on.

Observation 1 *A user can control the smart-device services without ever needing to connect these devices directly*

[1] Swing is a Java-based GUI toolkit.

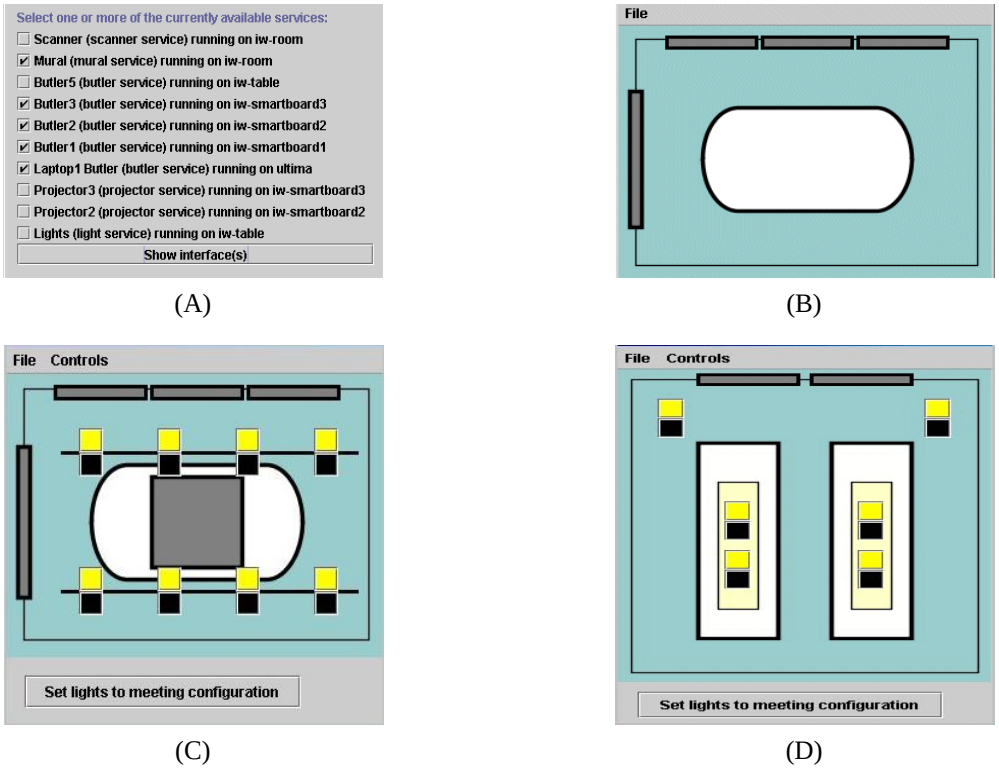


Figure 2. Ad-Hoc interaction illustrated. (A) A Swing UI showing all available services in a workspace. (B) A Swing UI for four displays in the workspace. The UI shows an overhead view of the workspace with each dark grey panel representing a display. The positions of the grey panels reflect the physical positions of the corresponding displays. Dragging a URL onto a display causes the web page to be opened on that display. (C) A simplified Swing UI for the lights in the workspace. The eight yellow/black button pairs are for turning on/off the lights respectively. The position of the buttons reflects the physical locations of the lights in an overhead view of the workspace. (D) The "same" lights UI as in (C) customized to a different workspace.

to her laptop or installing device drivers for them. The user simply requests UI's for services and appropriate UI's are automatically returned. We use the term "ad-hoc interaction" to refer to this mode of interaction.

Jane then requests a UI for three observation cameras (say c1,c2,c3) and three video player services (say v1,v2,v3). Using the returned UI, she sets up a desired video routing (say, c1→v2, c2→v3, and c3→v1).

Observation 2 *A user can request a multi-service UI (i.e., a UI for more than one service) and such a UI is automatically returned. Recall that even in the earlier step, Jane used a combined UI for the four displays.*

Later, Jack walks into the workspace with a device that doesn't support Java Swing but has a regular web browser. (Perhaps he carries a laptop without Java installed or a PDA.) Jack also performs all the actions performed by Jane above, using just the web browser. However, the plain HTML-based UI's he sees are inferior both aesthetically and from ease of use standpoint, since plain HTML lacks drag-and-drop or spatial positioning of widgets.

Observation 3 *No one UI fits all users, since different users carry different appliances, with different UI toolkits installed.*

1.2 Problem description and contributions

A natural question that emerges from the above scenarios concerns what happens when the user requests a UI. The interesting issues concern what UI is returned, from where and how it is obtained. For most existing frameworks, the UI's are expected to be provided by the services themselves. For example, in the UPnP (Universal Plug and Play) framework, every smart device is expected to provide a HTML UI served by an embedded web server. However, service-provided UI's are not enough, because every service can hardly be expected to cater to all (current and future) user appliances (laptop, PDA, next generation handheld), modalities (GUI, pen and tablet, voice), and UI languages (Java

Swing, HTML, WML, C#). Also, multi-service UI's (observation 2 above) cannot be expected to be provided by any individual service.

In "more advanced" frameworks such as Jini [2] and ICrafter [10], the UI's are obtained not directly from the services themselves but from a separate well-known "UI broker" service (called lookup service in Jini and interface manager in ICrafter). Consequently, these frameworks allow "third party UI's" (provided by a party other than the service itself) to be registered with the UI broker service. Services also register the vendor-provided UI's with the UI broker service. When a UI is requested, the UI broker service can return either a service-provided UI or a registered third party UI, whichever is available and appropriate. Thus, with these frameworks, the ad-hoc interaction behavior seen in a workspace can exploit both the UI's provided by the services and the third party UI's registered in the workspace.

However, even these more advanced systems lack mechanisms for flexible control of the ad-hoc interaction behavior either as seen by different users in a single workspace or as seen by a specific user in different workspaces. In particular:

- An administrator currently needs to manually locate and register the needed third party UI's. Also, there is no easy means for centrally administering the third party UI's in a group of workspaces that can together be considered a "logical unit". For example, to add a third party Swing light UI in all the workspaces in a building, manual administration effort is needed at every workspace.

- The end user has little control over which UI is returned. As she moves across multiple workspaces, she is likely to see differing UI's, even for the same service, because different service vendors could provide different UI's and/or because different third party UI's are registered in these workspaces. (For example, the UI's shown in figure 2 obviously represent only one of the essentially unlimited number of possible Swing UI designs independent UI designers can create for these services. A different designer could use a different set of colors, widgets, and metaphors such as drag-and-drop or spatial widget positioning.) This can be quite disconcerting, particularly for the more complicated services.

In this paper, we describe an architecture for ad-hoc interaction that provides mechanisms to address the above issues. The notable contributions of this architecture may be described as follows:

1. Distribution/administration mechanism: A workspace can be setup such that the third party UI's appropriate for the services in that workspace are automatically procured without manual intervention. Further, this mechanism allows a set of workspaces to be setup such the default ad-hoc interaction behavior seen in them is similar.

2. Personalization mechanism: If a user likes a UI returned in a workspace, she can "bookmark" this UI. From then on, if she requests a UI for a service of the same type (even in a *different* workspace), she is returned the familiar UI. Of course, where applicable, the UI must customize itself to the local workspace. For example, if the user bookmarks the light UI shown in figure 2(C), and requests a light UI in a different workspace, the returned UI looks similar but reflects the number and positions of lights in that workspace, perhaps as shown in figure 2(D). Effectively, once a user bookmarks a UI, she "carries" this favorite familiar UI with her, wherever she goes.

These two mechanisms are independent (i.e., neither is a substitute for the other). Also, either mechanism is useful by itself and can be provided independent of the other.

The rest of the paper is organized as follows. In section 2, we present the architecture. We raise several significant policy issues in section 3 and also present our current solution approaches to these issues. The prototype implementation is discussed in section 4, while section 5 discusses some alternative approaches to the problem. Section 6 describes related research.

2 Architecture

We extend an existing ad-hoc interaction system – specifically, the ICrafter system we proposed in [10] – to provide mechanisms for distribution and personalization. We first review the basic ICrafter architecture, which is shown in figure 3(A).

2.1 Background

In ICrafter, a user can use one of many supported UI language *renderers* (such as web browser, Swing renderer etc) for ad-hoc interaction. Consider a user wishing to control lights using Java Swing (as shown in figure 2):

1. When the user requests a UI for lights, the Swing renderer back-end requests a well known service called the interface manager (IM) for a "Swing **LightService** UI", assuming that the lights service exports the (programmatic) interface **LightService**.

2. The IM consults a local *generator repository* for a **LightService** Swing UI generator.

3. The located Swing **LightService** generator is executed (at the IM) to produce Swing UI (which is described in a home-grown markup language called SUIML).

4. The resulting Swing UI is returned to the Swing renderer and is rendered by the latter.

It helps to clarify the distinction between a "generator" and a "UI" in our terminology. In general, a generator is written for a service type (such as **LightService**) and *when executed at the IM* for a specific instance (such as the lights in room 104) generates a UI for that instance. Any user action on the resulting UI causes the corresponding operation being performed on that instance. As discussed in [10], where applicable, a generator may also customize the generated UI to the target workspace by dynamically accessing the workspace context information (such as physical geometry). For example, figures 2(C) and 2(D) represent two *different UI's* produced by the *same generator*. As an important special case, a generator can also be specific to a particular service instance, in which case it is essentially a UI.

Finally, as shown in the figure, the repository contains two types of generators: generators registered by the services (service-registered generators) and generators registered by a local admin (local generators).

2.2 Extended architecture

We extend the basic ICrafter architecture in two ways. (The extended architecture is shown in figure 3(B).) First, we allow the IM to be configured to automatically look for suitable generators in a *remote generator repository* in addition to the local repository. The remote repository is essentially a web service operated (for example) by the vendor shipping the IM software. (Ideally, there should be a hierarchy of such repositories to ensure scalability, availability and efficiency, although this is not the case with our current implementation.) The IM automatically downloads various UI language generators (HTML, Swing, etc) for the local services from the remote repository and caches them. Thus, the generator repository now also contains generators downloaded from the remote repository in addition to the service-registered and local generators. Only generators applicable to the locally running services are ever downloaded. For example, if no lights service is running locally, no lights generator will ever be downloaded.

Second, after making a request for the UI of a service, the user can "bookmark" the returned UI. When a UI is bookmarked, the *corresponding generator* (that generated the UI) gets added to the set of "favorite generators" on the user appliance. (What gets added is only a description of the generator and the URL of the generator, not the generator itself.)

With these two additions, the UI request life cycle of section 2.1 changes as follows:

1. When the user requests a UI for lights, the Swing renderer back-end requests the IM for a "Swing **LightService** UI". This request also contains the favorite generators of the user.

2. The IM selects a Swing **LightService** UI generator from among the favorite generators, generators in the local repository and the generators available at the remote repository. (If there are multiple Swing **LightService** generators, the "best" generator is chosen based on the prevailing policy.)

3. The located Swing **LightService** generator is executed (at the IM) to produce Swing UI.

4. The resulting Swing UI is returned to the Swing renderer and is rendered by the latter. If the user likes thus UI, she can bookmark it, causing the corresponding generator to be added to the user's favorites.

Not all UI's can be bookmarked as described above. In particular, a generator written specifically for a service instance, which hard-codes information specific to that instance, cannot be applied to a different service instance. Thus, only non instance-specific UI's should allow themselves to be bookmarked. For example, if the generator for the lights UI in figure 2(C) was written specifically for that workspace and hard-coded the number and locations of lights, it should not allow itself to be bookmarked. On the other hand, if the generator was written in a generic fashion, and uses room geometry information at runtime to customize the generated UI, the resulting UI can allow itself to be bookmarked.

Several contentious issues remain to be addressed in this architecture. However, before addressing them, we explain the workings of the architecture with a simple (hypothetical) example scenario.

2.3 Example scenario

Consider two buildings 1 and 2, each with several workspaces containing smart printers. Building 1 contains workspaces of two types: 1H and 1C. Workspaces of type 1H contain an HP printer while workspaces of type 1C contain a Canon printer. Building 2 also contains two types of workspaces: 2C and 2E. Workspaces of type 2C contain a Canon printer, while workspaces of type 2E contain an Epson printer. Finally, all workspaces in building 1 are configured to obtain generators from remote repository R1, while all workspaces in building 2 are configured to obtain generators from remote repository R2.

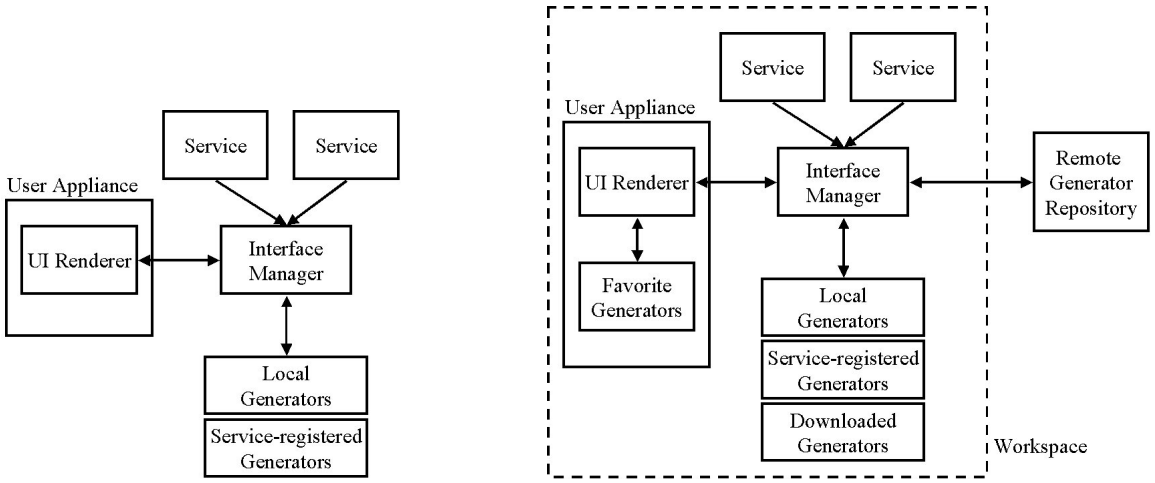


Figure 3. (A) Basic ICrafter architecture. (B) Extended ICrafter architecture. Notice the three additions in (B): remote generator repository, favorite generators on the user appliance, and the downloaded generators in the generator repository.

The HP and Canon printers export the (programmatic) interface **Printer**, while the Epson printer exports the (programmatic) interface **EpsonPrinter**. Also, **EpsonPrinter** is not a subinterface of **Printer**, nor are they derived from a common parent interface. The HP and Epson printers ship with Swing UI generators GH and GE respectively, while the Canon printer does not ship with any. Note that GH uses the interface **Printer** to invoke operations on the printer. On the other hand, GE uses the interface **EpsonPrinter**.

We will now illustrate using this example scenario how the architecture works: As shown in figure 4, Jane walks into a 1C workspace with her Swing renderer enabled appliance and accesses the printer using the Swing renderer. She is returned UI(GP1), i.e., the UI generated by GP1. She bookmarks this UI and GP1 (or rather, a description of GP1) is added to the favorite generators on her appliance. Sometime later, she enters a 1H workspace and accesses the local printer. This time, she is returned UI(GP1) again even though the local printer (which is a HP printer) exports the generator GH. Similarly, she sees UI(GP1) in 2C workspaces also. (Note that had she not had GP1 bookmarked, she would have seen a generator downloaded from R2 in the 2C workspaces.)

Jane now walks into a 2E workspace (as shown in figure 5) and accesses the local printer. This time, she is returned UI(GE2) and not UI(GP1). This is because, GP1 expects the interface **Printer** while the local printer exports the **EpsonPrinter** interface. (Here we assume the downloaded generator GE2 is preferred by the IM over the service-supplied generator GE. We discuss precedence issues in section 3.) Suppose she bookmarks this UI too, causing GE2 also to be added to the favorites. Then, for all future printer accesses, she receives UI(GP1) whenever the local printer exports the **Printer** interface and UI(GE2) whenever the local printer exports the **EpsonPrinter** interface.

3 Policy Issues

Several non-trivial policy issues stem from the architecture presented in the previous section. In this section, we raise these issues and explain how our current implementation approach resolves them. Since we do not yet know which policies are optimal, we have initially chosen to implement simple ones.

Generator selection policy: In addition to the generator matching logic (explained in [10]), the IM must also implement a selection policy. This is because multiple generators could possibly match at the IM for a given UI request. In such cases, the IM needs a policy to select the "best" generator.

We currently implement a two-level policy. The first-level policy determines the precedence order among the four types of generators: favorite, service-registered, local, and downloaded. The precedence order we use is: favorite $>$ local $>$ downloaded $>$ service-registered. We also allow the user to specify whether favorite generators should

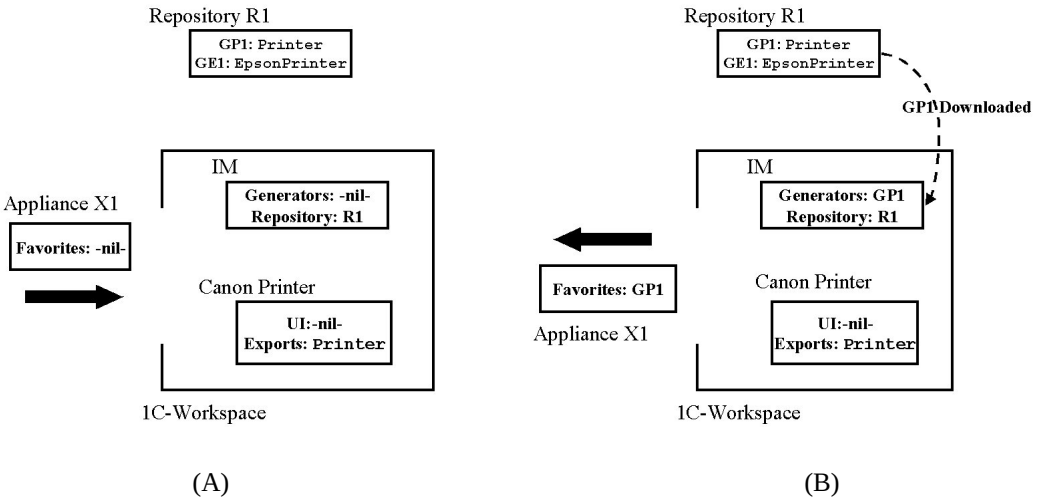


Figure 4. (A) Jane about to walk into a 1C workspace with her Swing renderer enabled appliance X1. The local repository initially does not contain any Swing generator suitable for the Canon printer, nor does X1 contain any favorites. (B) Jane accesses the printer. The IM downloads and caches GP1 from the remote repository and returns UI(GP1), which Jane bookmarks. As Jane walks out, GP1 is in both the local repository and X1's favorites.

be considered by the IM. Any UI request sent to the IM contains a boolean field termed "check favorites". If this field is not set, the IM does not consider the favorites during generator selection. The second-level policy determines the precedence order within each group. This policy randomly picks one of the matching generators.

Security: Since the generators include code, the system has to guard against malicious generators. Techniques such as code checking, sand-boxing, and/or signatures can be used to protect against malicious generators.

We use signatures to ensure safety. Thus, when the IM downloads a generator, it checks if the generator is signed by a trusted principal. Unsigned generators are rejected. The current trust policy is simple: the only trusted principal is the remote repository. With this policy, there is no need to configure each IM with a set of trusted principals. Rather, the trust administration is delegated to the remote repository. Of course, we do need a policy at the remote repository to determine the trusted principals. If the provider of a generator is trusted according to this policy, the remote repository signs the generator with its key.

Our current policy either fully trusts a generator (that is, allows access to all resources) or fully rejects it. A more flexible policy would allow the specification of the resources a generator may access in the policy. Much research has gone into the mechanisms needed for supporting such policies. Specifically, for Java (which is our implementation platform), Wallach et al. [14] present several mechanisms for mobile code security. The Java 2 platform from Sun [6] provides a stack introspection based mechanism. A natural extension to our implementation is to allow "full policy specifications" for generators at the remote repository, which can also be downloaded by the IM in addition to the generators themselves.

Favorites management: The favorites on the user appliance need to be jointly managed by the various renderers. This can be problematic, especially for "legacy" renderers such as web browsers, which we do not own the code for. Another issue is the transmission of favorites to the IM, for which there are at least two options. First, the favorite generators relevant to a request (if any) can be selected at the appliance side itself. Alternatively, the entire list of favorites can be included in the request to the IM.

Our current approach is to distribute the favorites management among the different renderers. For example, a web browser only manages the HTML generators among the favorites, while a Swing renderer manages the Swing generators among the favorites. We have so far implemented "favorites management" only for web browsers. Also, for simplicity, we include all the favorite generator descriptions in a request to the IM whenever the "check favorites" field is set to true. Note that a generator description includes the URL where the actual generator resides.

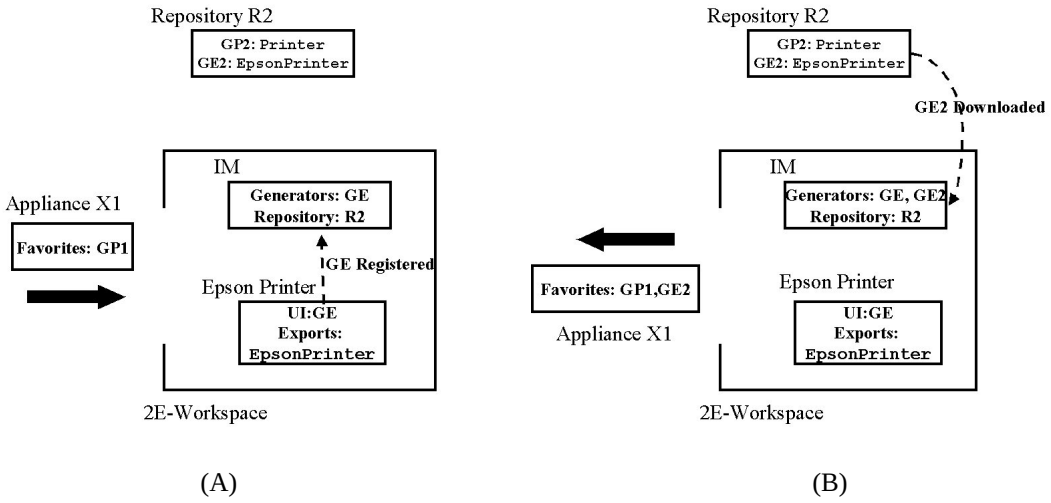


Figure 5. (A) Jane about to walk into a 2E workspace with her appliance X1. The local repository initially contains the Swing generator GE registered by the Canon printer, and X1 contains one favorite generator, GP1. (B) Jane accesses the printer. The IM downloads GE2 from the remote repository and returns UI(GE2), which Jane bookmarks. As Jane walks out, GE2 is in both the local repository and X1's favorites.

Active vs. passive remote repository: The remote repository can contain varying levels of "smartness". On the one hand, a passive remote repository simply serves as a listing of all available generators. On the other hand, an active remote repository supports queries of the form "return all **Printer** HTML generators" etc. As the remote repository logic becomes more complicated, the sustainable request rate diminishes.

We have currently adopted the passive remote repository approach. Thus, the remote repository is essentially just a HTTP-accessible listing of all the available generators. As shown symbolically below, this listing only contains the generator descriptions and locations (but not the generators themselves):

```
.....
G1: Interface: org.service.Printer
 Output Language: WML
 Location: http://repository.org/G1.jar
G2: Interface: com.epson.EpsonPrinter
 Output Language: HTML
 Location: http://repository.org/G2.jar
G3: Interface: org.service.LightService
 Output Language: Swing
 Location: http://repository.org/G3.jar
.....
```

To enable efficient searching, each IM downloads and caches this listing, and then searches for the "relevant" generators. For example, an IM in a workspace containing only one service – a printer, which exports the interface **org.service.Printer** – only downloads the generator G1 from the repository shown above. (The **com.epson.EpsonPrinter** generator G2 is *not* downloaded.)

Pro-active vs. on-demand downloading: In on-demand downloading, the IM checks the remote repository only when processing a UI request from an appliance and downloads any generators relevant to the request. In pro-active downloading, as soon as a new service is added to the workspace, the IM downloads all the generator(s) for that service. With either approach, the downloaded generators are cached for future use.

Our current implementation uses an on-demand downloading strategy. As mentioned earlier, the remote repository is passive. The overall strategy is as follows: For the first request, the IM downloads and caches the generator listing from the remote repository. The generator listing at the remote repository can change over time. Thus, while handling subsequent requests, the IM downloads the listing again if the cached listing becomes "stale". The listing is considered stale after a configurable "timeout period" has expired since the most recent download time.

4 Implementation status

While all the components of the system have been implemented, we have not deployed it in our testbed environment yet. Also, the distribution/administration architecture is fairly general, but the personalization part has only been implemented for web-based UI's.

Figure 6 illustrates how the prototype is intended to function. Jack walks into the workspace with an appliance containing a web browser. He then enters a well-known URL into the browser. This URL represents the "bootstrap" user interface, and lists all the services currently running in the space as shown in figure 6(A). Notice the "check favorites" checkbox, which allows him to indicate if his favorite generators should be considered. If this box is checked, the back-end servlet extracts the favorite generator descriptions from the "favorites cookie" and includes them in the UI request sent to the IM. Jack selects the lights service, and the returned UI is shown in figure 6(B). Notice the bookmark button at the bottom of the page. Jack clicks this button, and the back-end servlet extracts the generator description from the hidden fields and adds them to the favorites cookie.

5 Discussion

In this section, we discuss various related and alternative approaches to service interaction.

5.1 Planned interaction

Mobile users can also interact with a service in what we call "planned interaction mode", where users are themselves responsible for procuring the appropriate application/UI to interact with the service in advance of using the service. The means for procurement can vary. For example, to interact with a printer, a user can install a printer client application on her appliance, which discovers the locally available printers using a discovery protocol and enables printing to them. Another option for the user is to locate a "printer client applet" on the web that does the same. Although no installation is needed with an applet, the user becomes responsible for trust management.

The distinction between the ad-hoc and planned modes is important, but can be subtle. For example, if a user locates and loads a printer client applet herself, we label this planned interaction. However, if the same applet is uploaded to the user on-demand by the system when she requests the UI for a particular printer, we consider it ad-hoc interaction. [2]

[2] In reality, the same printer applet cannot typically be used for both modes of interaction. An applet used for planned interaction is usually designed as a "stand-alone" applet, while an applet used for ad-hoc interaction is designed as an "attachable" applet, so it can be bound to the particular printer (for which the UI was requested) before being returned to the user.

The main advantage of planned interaction is that the user experience is predictable, since the users themselves install/locate chosen UI's. However, the very fact that users need to install/locate UI's themselves renders this mode impractical in several situations. Also, locating a suitable UI gets further complicated for the end-user in the absence of universal agreement on service interfaces. For example, if the printer client applet expects the **Printer** interface, while the workspace contains only an Epson printer, then the applet wouldn't discover the printer and consequently, the user would be unable to print. Ad-hoc interaction systems fare better in this regard since the system would return an **EpsonPrinter**-compatible UI to the user.

Finally, planned and ad-hoc interaction are complementary. In other words, pre-installed applications or known applets can be used where available. For the other services, ad-hoc interaction can be used.

5.2 Smart appliances, smart middle, or smart services?

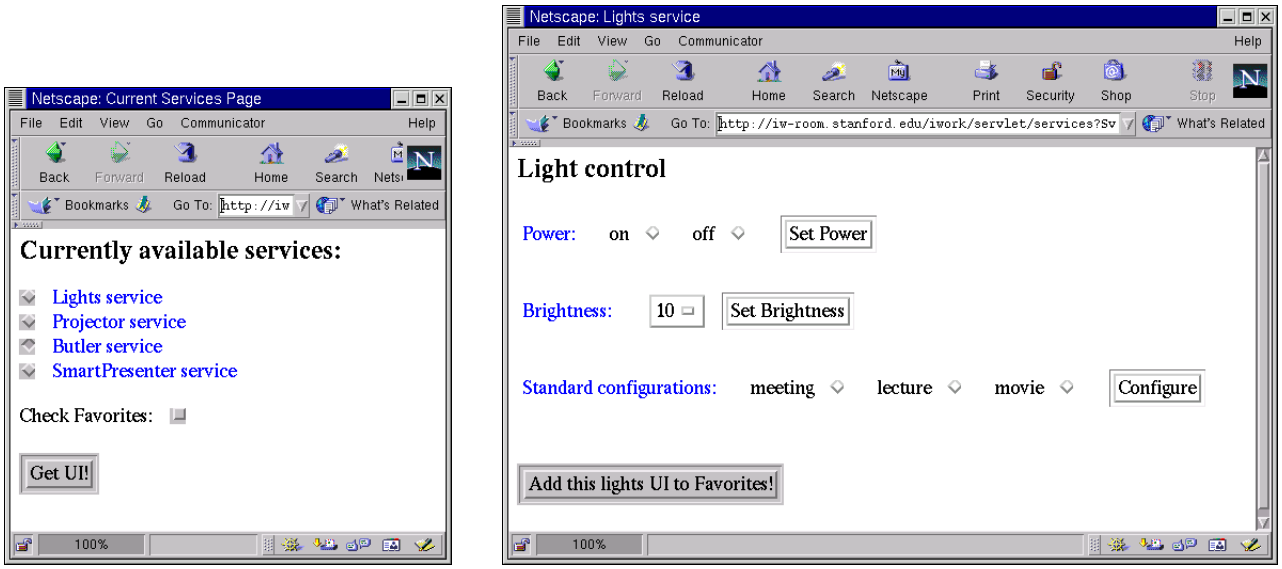
In principle, the functionality we have placed in the IM can be moved to the user appliance-side or the service-side, or can perhaps be split across the appliance-side and the service-side. This represents the quandary of "smart client, smart server, or smart middle?", which often recurs in network systems. The approach we have described in this paper may be considered as the smart middle approach. The other approaches are briefly described below:

- Smart service: In the smart service approach, the services would be written such that they automatically look for UI's in a remote repository (possibly maintained by the service vendor) and return them to the user.

- Smart appliance: In the smart client approach, the appliance software would automatically look for UI's in a remote repository, if no suitable UI is found among the favorites or provided by the service.

These approaches are *not* mutually exclusive. Also, each has its own disadvantages. The smart middle approach has the disadvantage of requiring a third party (IM), which may not be feasible in all environments. On the other hand, the IM also provides a central "control point" for changing the ad-hoc interaction behavior of the entire workspace without having to make changes at each service/appliance.

The smart service approach does not generalize well for multi-service UI's. (With this approach, it's not clear where multi-service UI's are maintained, or where a user queries when the UI for a group of services is desired.)



(A) (B)

Figure 6. (A) Bootstrap UI showing all services in a workspace. Notice the "check favorites" checkbox (B) Lights UI. Notice the "add to favorites" button. Only instance-independent generators should contain this button.

The smart appliance approach has the disadvantage of requiring more functionality on the appliance, which implies that this functionality has to be developed and ported for every appliance type and platform. Besides, there is reluctance among end-users to install software (especially evolution-prone software). In contrast, with both the smart service and smart middle approaches, minimal or no installation may be necessary on the appliance. For example, for web-based UI's, our current approach requires only a web browser on the appliance.

6 Related work

Software frameworks for future office, home, and laboratory environments are being investigated by several research and industry groups such as Microsoft EasyLiving [3], Sun Microsystems' Jini [2], Oxygen [4], Aware Home [1], and one.world [7]. In this context, several service frameworks have been proposed: Jini [2], UPnP [5], Hodes et al. [8], Roman et al. [11], etc. Most of these frameworks allow ad-hoc service interaction. However, none of these systems really address the distribution/administration or personalization problems.

Hodes et al. [8] briefly discuss (but do not provide a solution for) the problem of adapting an existing UI to work with a service exporting a different programmatic interface. Spreitzer et al. [12] point out some limitations in current programming language interfaces with respect to evolvability and propose the use of flexible types. Techniques for evolvable interfaces and adapting existing UI's to work with different service interfaces are very relevant but complementary to the work presented in this paper.

The Mobisaic system [13] extends the web with "active documents" that can react to changes in the user's context. This system is specifically designed for the web and is also primarily meant for information browsing rather than service control. Also, it is difficult to implement generators as active documents since the latter only provide environment variables for customization. (We use an embedded scripting language in our current implementation of generators.)

Auto-update features are becoming increasingly popular in commercial operating systems as well as applications such as browsers and media players. In principle, automatic downloading of generators by the IM is similar to the automatic downloading of needed plug-ins or codecs by browsers and media players respectively. However, there are differences both at the mechanism-level and what the mechanism enables:

- When a new generator is downloaded, it changes the behavior of the workspace, rather than any specific machine.
- The distribution architecture also effectively facilitates

centralized administration of all workspaces in a single administrative domain.

- Matching generators is more complicated than matching plug-ins and this affects the mechanism design.

7 Conclusions and Future Work

In this paper, we proposed an architecture that addresses some of the deficiencies in existing ad-hoc interaction systems. Specifically, the architecture we propose provides the following mechanisms:

1. Distribution/administration mechanism: A workspace can be setup such that the third party UI's appropriate for the services in the workspace are automatically procured without manual intervention. Further, it can be ensured that the ad-hoc interaction behavior in a set of workspaces is similar.

2. Personalization mechanism: A UI returned on-demand during ad-hoc interaction with a service can be "bookmarked" by the user. Once a favorite UI is bookmarked, it is automatically re-used for future interactions (wherever applicable).

Much work remains to be done to improve the system. In particular:

- In addition to the mechanisms themselves, effective policies are also needed to produce the best results with minimum manual intervention. While we have currently implemented the simpler policies, more experience and experimental evidence is needed to determine the "best" policies.

- We are exploring the possibility of hierarchically organized repositories for efficient and scalable generator distribution.

- For true personalization, the favorite generators need to be remembered for each user across all the appliances she may use. We are investigating efficient and scalable mechanisms for enabling this.

Acknowledgements: We are grateful to Ed Swierk and Guido Appenzeller for reviewing earlier drafts of this paper and providing helpful suggestions for improvement. Special thanks are due to Brian Lee and Susan Shepard for their help in the design and implementation. Suggestions of the anonymous reviewers helped improve this paper, and we are particularly grateful to them for their valuable insights toward future directions to pursue in this work.

References

[1] G. D. Abowd, C. G. Atkeson, A. F. Bobick, I. A. Essa, B. MacIntyre, E. D. Mynatt, and T. E. Starner. Living Laboratories: The Future Computing Environments Group at the Georgia Institute of Technology. In *CHI'00 Proceedings*, The Hague, Netherlands, April 2000.

[2] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison Wesley, 1999.

[3] B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. A. Shafer. EasyLiving: Technologies for Intelligent Environments. In P. J. Thomas and H.-W. Gellersen, editors, *HUC*, volume 1927 of *Lecture Notes in Computer Science*, pages 12–29. Springer, 2000.

[4] M. Dertouzos. The Oxygen Project. *Scientific American*, 282(3):52–63, August 1999.

[5] U. Forum. Universal Plug and Play. `http://www.upnp.org`.

[6] L. Gong. Java 2 Platform Security Architecture. `http://java.sun.com/j2se/1.3/docs/guide/security/spec/security-spec.doc%.html`.

[7] R. Grimm, J. Davis, B. Hendrickson, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. Systems Directions for Pervasive Computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 147–151, Elmau, Germany, May 2001.

[8] T. D. Hodes and R. H. Katz. A Document-based Framework for Internet Application Control. In *2nd USENIX Symposium on Internet Technologies and Systems (USITS 99)*, Boulder, Colorado, USA, October 11-14 1999.

[9] T. D. Hodes, R. H. Katz, E. Servan-Schreiber, and L. Rowe. Composable Ad-hoc Mobile Services for Universal Interaction. In *Third ACM Conference on Mobile Computing and Networking (MobiCom 97)*, Budapest, Hungary, September 1997.

[10] S. R. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. In G. D. Abowd, B. Brumitt, and S. A. Shafer, editors, *Ubicomp*, volume 2201 of *Lecture Notes in Computer Science*, pages 56–75. Springer, 2001.

[11] M. Roman, J. Beck, and A. Gefflaut. A Device-Independent Representation for Services. In *Third IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2000)*, December 2000.

[12] M. Spreitzer and A. Begel. More Flexible Data Types. In *Proceedings of The Eighth IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE '99)*, 1999.

[13] G. M. Voelker and B. N. Bershad. Mobisaic: An Information System for a Mobile Wireless Computing Environment. In *First IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 1994)*, December 1994.

[14] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architecture for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, Saint Malo, France, October 5-8 1997.

[15] M. Weiser. The Computer for the Twenty-First Century. *Scientific American*, pages 94–100, September 1991.