2011-11-28

Carry and Expand: A New Nomadic Interaction Paradigm

Richard B. Arthur
Brigham Young University - Provo

Carry and Expand: A New Nomadic Interaction Paradigm

Richard B. Arthur

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Dan R. Olsen, Jr., Chair
Parris Egbert
Michael A. Goodrich
Eric Mercer
Kent Seamons

Department of Computer Science

Brigham Young University

December 2011

ABSTRACT

Carry and Expand: A New Nomadic Interaction Paradigm

Richard B. Arthur
Department of Computer Science, BYU
Doctor of Philosophy

People are nomadic; traveling from place to place. As a user travels, he may need access to his digital information, including his data, applications, and settings.

A convenient way to supply this access is to have the user carry that digital information in a portable computer such as a laptop or smart phone. As Moore's Law continues to operate, devices such as smart phones can easily perform the computing necessary for a user's work. Unfortunately, the amount of data a human can receive and convey through such devices is limited. To receive more information humans require more screen real estate. To transmit more information humans need rich input devices like mice and full-sized keyboards.

To allow users to carry their digital information in a small device while maintaining opportunities for rich input, this research takes the approach of allowing users to carry a small portable device and then annex screens, keyboards, and mice whenever those devices are available in a user's environment.

This research pursued the "carry it with you" paradigm first by building an ideal annexing framework which helps maximize the screen real estate while minimizing the resources—RAM, CPU, and wireless radio—consumed on the personal device. The resource consumption is demonstrated through a comparison with existing remote rendering technologies. Next, a privacy-aware framework was added to the annexing framework to help protect the user's sensitive data from damage and theft when he annexes a potentially malicious device. A framework like this has not existed before, and this research shows how the user's sensitive data is protected by this framework. Third, legacy machines and software are allowed to participate in the carry-it-with-you experience by scraping pixels from the user's existing applications and transmitting those pixels to an annexed display. Finally, when a user encounters a display space he does not own, but which he needs to control (e.g. by preventing anyone else from annexing it simultaneously, or by constraining each user to a different section of the display space), rather than forcing the user to learn and use control software supplied by the display, the user can bring his own control software and use it to enforce the user's desired control paradigm.

This dissertation shows the carry-it-with-you paradigm is a powerful potential avenue which allows users to confidently use display spaces with varying configurations in an assortment of environments.

Keywords: screen annexation, nomadic users, privacy-aware, display server

ACKNOWLEDGEMENTS

I would like to thank all of my committee members for their support and encouragement as I worked on this research. Their insight has helped me clarify this work, in terms of both research quality and writing precision.

I also appreciate BYU for funding this research.

I also would like to thank my family, especially my wife, for helping me pursue this research and supporting me as I documented the results of this research.

Contents

Figures

Chapter 1 A New Model of Nomadic Interaction

1.1 Introduction

People are nomadic. As a user travels he[1] may require access to digital information wherever he is located. In particular, he may require his data, applications, and settings.

Portable digital devices can easily perform much of the processing that a user needs, but the amount of data a human can receive and convey through such devices is limited. To receive more information humans require more screen real estate. To transmit more information, humans need rich input devices like mice and full-sized keyboards.

Instead of carrying devices, a user could employ a device at his destination, such as a kiosk, TV, or projector. These devices are also called *display servers* and may supply input. A room-supplied computer has a much larger space for output and much richer input devices than portable devices can supply.

To annex a display server, the user has a couple of choices. Give his data directly to a display server or get his data via the display server. Giving data directly to the display server is highly insecure because the display server could easily steal or damage the user's data. However, to get the user's data via the display server requires the user to supply his credentials which creates three problems. First, supplying credentials requires a trusted third-party to authenticate those credentials. Second, the user can only log into computers that can reach the trusted third party's authentication servers. And finally, supplying credentials is insecure because the display server could easily steal those credentials or damage the user's data. The only way to properly guard against such attacks is if the user completely trusts the display server; either he or a trusted

[1] This gender-specific pronoun is used for convenience and should be interpreted as gender neutral.

third-party (e.g. employer) must own and maintain that machine. Such an approach does not allow users to anonymously annex machines wherever they may be located which limits users to machines that they or a trusted third-party controls (e.g. work and home devices, only).

In addition, with either of these choices, a user may be required to relearn known software at the display server. A foreign display server may have the wrong software version or a competitor's version, may not have the software at all, or may have an augmented and potentially confusing user interface. With the ever-increasing number of software products a user must learn and interact with, requiring the user to learn and remember all the different versions of software adds to the complexity in his life. Ideally the user should be required to learn the software once and then use it anywhere else.

The goal of this research is to allow a cellphone or smaller-sized device to become the user's dominant computing device, to remain physically small, and, in a relatively safe manner, to supply a consistent, larger interface for him to interact with at any display server he encounters. This is accomplished through a hybrid approach where the user carries a small trusted portable device which executes his software and distributes the UI of the application, unchanged, to a display server.

This new approach allows the user to access his data, applications, and settings at any time. It also ensures that he does not supply his credentials to an untrusted display server, he does not supply his data to the display server, and he does not supply his software to the display server. The user can connect to and disconnect from such display servers at will and can selectively choose which applications to show. Finally, because the user always carries his software with him and the application rendering is consistent regardless of the display server's software, the user only needs to learn his software once and can then use it at any display server.

In this interaction model, the user annexes screen space or input devices in whatever room he enters. As illustrated in Figure 1:1, annexing these devices is done through a network protocol as opposed to direct cable connections.

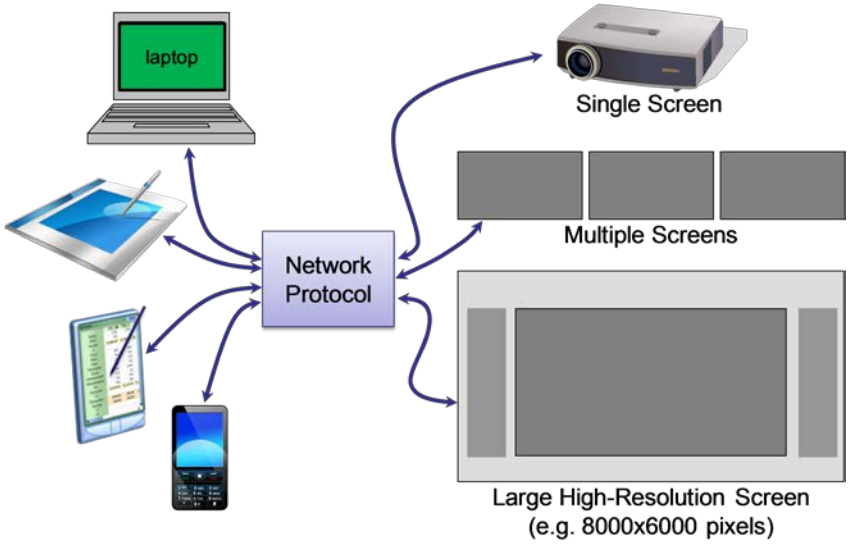


Figure 1:1—Nomadic hardware and software organization. A user carries his own computer (illustrated on the left) to rooms where he can annex the available screen space and input devices (on the right).

To help explain where this new carry-it-with-you interaction model fits in the ecosystem of nomadic interactions this dissertation will proceed as follows. Section 1.2 organizes the different solution approaches with respect to their software and hardware organization strategies. In section 1.3 the author's new nomadic interaction model is discussed in greater detail as it pertains to the user's interaction model. Finally, in section 1.4, the plan for researching this new interaction model is laid out detailing some of the aspects of this research model which were measured for this dissertation.

1.2 Design Strategies

A nomadic solution has several hardware and software components. The user will of course require access to his data. The user will also need his settings. He also needs an

application for altering his data. Each application needs display and input devices so that the user may see the application's output and provide input to it. In this dissertation these needs are separated into five components: data, settings, applications, output, and input.

This dissertation surveys several broad approaches to combining these components: self-contained, transmit data, transmit code, transmit the output and input, use the web, and brokered access. Figure 1:2 illustrates of the basic design of each of these approaches. The components (listed in the legend) are combined using one or more computers and are grouped according to which computer is responsible for each component. The computers may be supplied by the user (a portable device), supplied by a room owner (environment), or supplied by a remote machine (typically a website or corporate server). In these illustrations, any device carried by the user into a room (e.g. cell phone, laptop, or tablet) is a *personal device* which is denoted as a green group (always safe and trusted) with long-dash borders; any device situated at a distant location (e.g. a web server, a computer at home, or a distant office computer) is a *remote device* indicated by a blue group with alternating short- and long-dash borders (trusted and possibly owned by the user); and any device supplied within the room which the user may annex is an *environment device* signified by a red group with short-dash borders (probably not owned by the user).

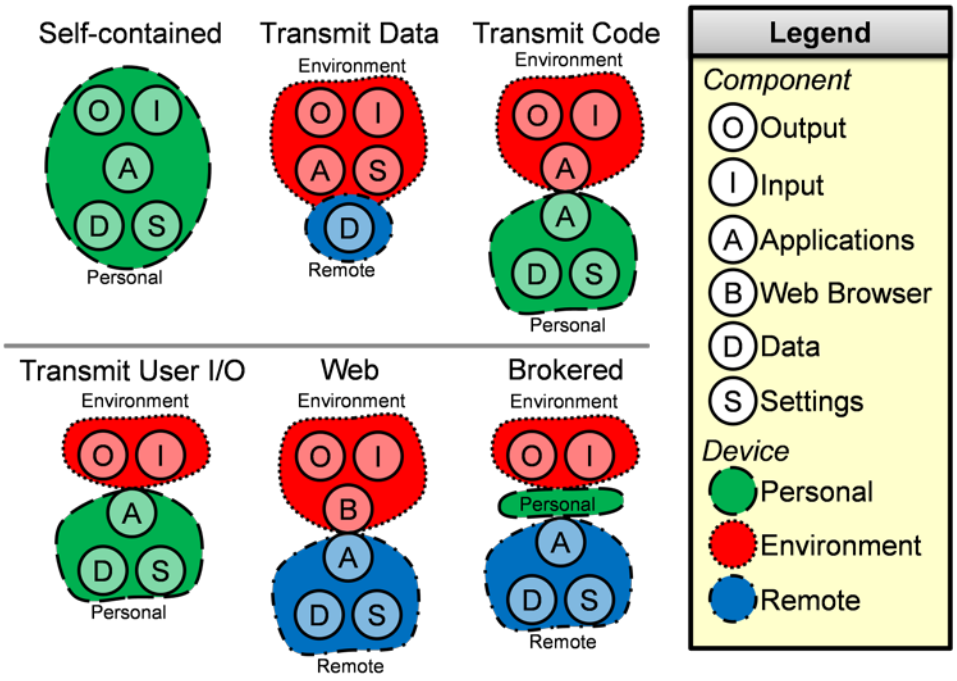


Figure 1:2—Several approaches to combining components. The different components are combined using the network and a personal device, environmentally-supplied device, and/or a remote device.

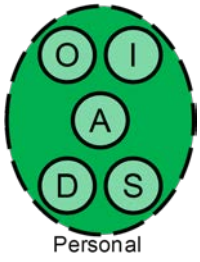


Figure 1:3—Self-contained approach.

1.2.1 Self-contained

In the self-contained combination approach the user's personal device performs all the work. This personal device may be a laptop, smartphone, tablet, or some other portable device. This situation is embodied in the example of a person who's whole life is contained in a laptop which he carries everywhere. The personal device renders output to its own screen, provides its own input devices (e.g. touch, stylus, or buttons), stores and executes all applications, and stores all of the user's data and settings.

The self-contained approach provides many of the user's requirements. Because the device is portable and carries the user's data, settings, and applications, this approach provides familiar applications, easy access to his data, a physically lightweight device, and protection

from malware (because it does not accept software from other sources or accept input from external devices).

Regrettably, small devices are not as powerful as desktop machines. The smaller the device, the more portable it is, but also the more limited its input and output. A laptop keyboard is somewhat adequate, but full-size ergonomic keyboards may be better for the user. Netbook and cellphone keyboards are much more difficult to use. Pointing hardware on such devices can also be frustrating, imprecise, and/or cumbersome. And although the user could carry around additional input devices (e.g. an external keyboard and wireless mouse), this requires the user to carry and manage more hardware. In addition, the visual output of a device is limited by the size of the display. A common way to provide more output for a portable device is to attach another monitor via a VGA or DVI port. Unfortunately, even if there are multiple screens, a user can annex only one additional screen. All the other screens are unavailable without adding hardware to the portable device. Smaller devices without such ports cannot annex any screens. Because the screen annexation is via a VGA or DVI cable, the annexing is exclusive; the number of users that may simultaneously annex a single screen is limited to one. A more flexible solution is necessary.

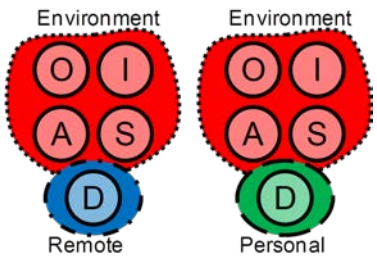


Figure 1:4—Transmit data approach.

1.2.2 Transmit Data

Transmitting data is a common approach for reducing the weight that users must carry (possibly to nothing) while increasing the interactive power. Each room is equipped with a *display server* (environment device) which provides all the input and output for manipulating the user's data. The display server provides all applications for manipulating the user's data and often provides the storage for the application's settings. The

user provides his data via the network, Internet (remote device), or a physical device like a USB hard drive (personal device).

Examples of network- or Internet-supplied data include Windows Roaming Profiles [16], iRoom [12], UbiTable[27], MultiSpace [18], MultiBrowsing [25], i-LAND [28], and Roomware [22]. In these cases data moves to the machine the user is at either automatically or because the user instructs the room's framework to move his data to that machine.

Rather than moving data from one machine to another, the data could be synchronized among all machines the user interacts with. For instance, tools such as Dropbox [7], Groove Folder Sync [18], and Windows Live Sync [19] each monitor data files on a user's machines. If the user has such a tool installed on his work and home machines, then when he updates a file at work the file is automatically copied to his home machine. Conversely, when he changes a file at home, the tool automatically copies that file to his work machine.

Another approach is for users to just carry their important files on a USB device. The user can plug that device into any machine he encounters and use the software on that machine to alter his data. This approach is common for existing users. Dynamo [11] is an advanced approach which allows users to carry a USB with their data to a collaborative display server where multiple users can work on the same display space and share data with each other.

Transmission or synchronization works well under controlled circumstances where the user knows and completely trusts each machine he will interact with. He can install his needed software on those machines, and he can ensure that those machines are free of malware.

However, encountering a display in an airport, café, taxi, or some other situation is not controlled or safe. The display may not have the software the user needs, or the software it has may conflict with the data he has stored. Imagine a user accustomed to working with Microsoft

Word 2003. Over lunch, this user needs to update a document and uses the display server embedded in a restaurant's table. However, that display has Office 2002 installed so the user cannot open his document. Or, that display has Office 2007 installed which has a radically different UI that the user is not accustomed to. Worse, the display could have no software installed for manipulating that document.

The display may not be safe, either. A previous user may have infected the display server in the restaurant's table with malware. If the user opens his Word document at that table then his data will likely get infected.

Transmitting data may work for controlled environments, but does not scale well for the wide variety of places that the user may encounter. Although it provides easy access to the user's data on screens he owns, it may not provide such access on other screens. This approach likely will not provide familiar applications, protection from malware, or the ability to collaborate.

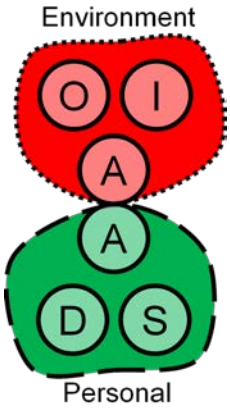


Figure 1:5—Transmit code approach.

1.2.3 Transmit Code

Instead of carrying just data, the user could carry his applications and settings on his USB device. Such an approach is available on U3-enabled [30] USB drives. When the user plugs his USB drive (personal device) into a display server (environment device) his software is transmitted to the display server where it can execute. This approach provides the user's applications, data, and settings wherever he is located. He has familiarity because his applications are always available and he does not need to rely on the display server owner installing the proper applications.

Unfortunately, a U3-enabled device relies on the display server having a specific operating system, code base, and processing hardware. A different approach would be to compile the application to byte-code such as Java [10], Silverlight [17], or Flash [1]. Byte-code theoretically makes the software compatible for any OS and platform. But byte-code has its drawbacks: it frequently involves "write once debug everywhere" development, the byte-code vendor has a large software base, the vendor can only support a limited number of platforms, and byte-code frameworks often limit the developed software. Frequently byte-code implementations each have slight differences from each other. The complexity of the byte-code framework inhibits creating and running comprehensive tests that guarantee uniform implementations. Although byte-code implementations could be built to be "uniform enough" for widespread use, the volume of tests discourages targeting more than two or three platforms, and does not facilitate outsourcing implementations to end display server manufacturers. In addition, byte-code frameworks often limit applications, which curbs application power. For instance, Silverlight applications cannot listen for incoming network connections.

Transmitting code is also slow and unprotected. A simple observation of the installation size of application installation folders reveals that applications consume far more space than the data files they alter. Transmitting all that code to the display server can be burdensome, especially if the user initiates the transfer over a wireless network.

Transmitting applications to the display server also leaves users vulnerable. After an application is transmitted to the display server, the user's data must be transmitted to and from the display server. Malware on the display server could steal or infect his data.

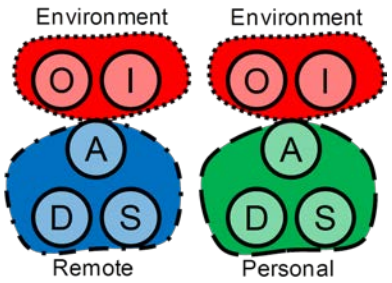


Figure 1:6—Transmit output and input approach.

1.2.4 Transmit the Output and Input

The user needs a way to maintain control over his data, applications, and settings: placing those under the control of a foreign machine is potentially dangerous. If the user can execute his applications on a machine he trusts, but transmit the application's output to a different machine and receive input from that machine, then he can reduce startup time (because the application is not transmitted) and keep his data safe from infection. For example, the user may bring his laptop (personal device) to a conference room and annex the display (environment device) using Remote Desktop Protocol (RDP) [29]. Or, the user may just come to the room and leave his computer at home or in his office (remote device), and use RDP to access his computer via the display.

How effective transmitting the output and input is at meeting the user requirements is dependent on how the connection between the display server and personal device is established. The connection may be established from the display server or from the personal device.

Transmitting the output and input has the same benefits and drawbacks regardless of the approach to establishing connections. Transmitting a UI eliminates the version conflicts from transmitting data or code because the user carries his own software. This approach also provides easy access to data because the user carries his own data.

Unfortunately, the transmitted application UI is sometimes not as familiar to users. Tools such as RDP and X-Windows (X11) [24] often augment a UI to match the display server's theme, which may unify the interaction among all applications shown on the display server, but can confuse users. For example, sharing a window from a PC to a Mac or Linux box can greatly

alter the UI and interaction techniques for window management. However, if the protocol is Virtual Network Computing (VNC) [23], then applications remain consistent. Although specific protocols such as RDP or X11 are faster than VNC, VNC is the most broadly compatible protocol. This fact makes it more likely that existing devices will use VNC instead of a different UI distribution protocol.

Regrettably, current systems for transmitting the UI are typically not sufficiently collaborative. The combination of these protocols (RDP, X11, or VNC) and the windowing toolkit prevent multiple users from simultaneously interacting with data on the display server; only one person at a time may use the input hardware on the display server. If the display server supported multiple users interacting with data simultaneously then this option becomes more collaborative. Each user would bring his or her information and share that information on the display server. Like with Dynamo, if the display server has multiple input devices, then each user should be able to use those devices to simultaneously interact with his or her data.

Physically light devices may not have the resources to supply a rich interactive environment using these existing protocols. These protocols often impose a heavy network, RAM, and processing burden on portable devices. Because each of these burdens requires electricity, the battery can be drained more quickly. For example, if using VNC, the handheld must allocate bytes for each pixel of a window shown on the display server and must also transmit each pixel to the display server. On large displays (i.e. on the order of two million, ten million, or more pixels) such pixel and network allocations can be prohibitive and slow. These restrictions may prevent small personal devices from participating meaningfully in user interaction.

Most importantly, the input and output of existing UI transmission technologies are insufficiently secure. The device or application is often put completely under the display server's control. Any input is accepted from the display server, which allows a malicious display server to install malware or perform other malevolent activities. Any output shown on the display server can be stolen by that display server; this gives a malicious display server access to the user's email addresses, user names, or other sensitive information.

As part of their solutions the SPICE and XICE protocols both take an approach similar to transmitting output and input. However, these protocols address the drawbacks of this approach (changing UIs, insufficient collaboration, the resource burden imposed, as well as input and output security), as is discussed in section 1.3.

1.2.4.1 Connect via Display Servers

Establishing a connection from the display server is where the user interacts with the input hardware on the display server to provide it with the information necessary to connect to the user's personal device or some other remote machine. Another way of looking at this interactive paradigm is that the user is *pulling* a UI from his personal device to the display server. This approach is frequently used by people who manage datacenters via technologies such as VNC, RDP, and X11. This approach is also used by the consumer who must access his home or work computer remotely via GoToMyPC [14] or WebEx [13]. Figure 1:7 illustrates the basic designs of the pulling approach; the application executes on a machine that is further away from the display server than the user is, and there is only one user interacting with the input hardware on that display server.

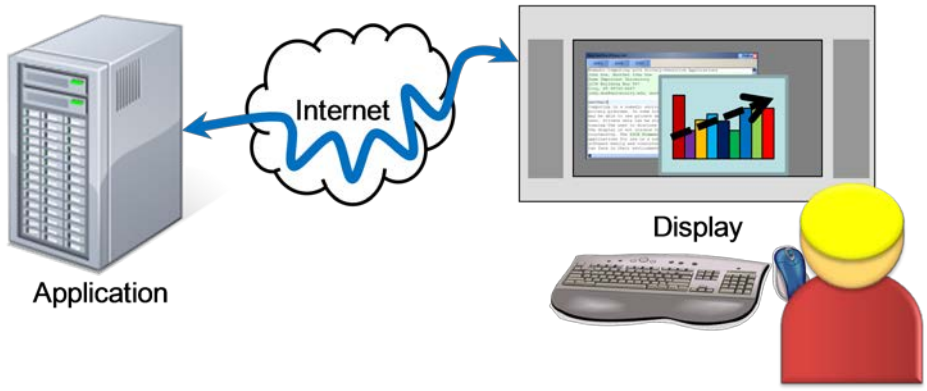


Figure 1:7—Design of VNC, RDP, X11, and HTML UI distribution and user interaction. A single user interacts with the input hardware on the display server. The application runs on a remotely located machine.

The remote machine may be in the same building as the user, same corporate campus, across town, or across the world. As the distance increases, so does latency, making interactive cycles longer. Unfortunately, if the remote machine is located in New York and the user is in California, such interactions can be slow as actions require a long round-trip. If VNC is the protocol, applications remain consistent but the slowdown is compounded.

Most importantly, the pulling approach is insecure. A user must walk up to a display server, and using that display server's hardware, enter the connection information necessary to connect to his remote device, including his credentials. Once the user pulls the UI from his remote device, the display server is given complete control over that remote machine. A display server infected with malware could easily capture the user's sensitive information, install itself on his remote device, or otherwise harm his data and machine.

1.2.4.2 Connect via Personal Devices

The user could establish the connection from his personal device to the display server. Then the user could *push* an application's UI from his personal device to the display server.

The largest benefit of pushing a UI is that it can protect some sensitive data (namely the user's credentials), can protect the user's desktop from full exposure to the display server, and can allow multiple users to simultaneously push windows to a display server. RDP and X11 have options for pushing a UI to another machine. Research tools such as Lacome [14], WeSpace [32], and IMPROMPTU [4] use VNC to transmit a UI from one machine to another. Some of these research tools also allow multiple people to interact simultaneously (in addition to showing output at the same time).

The latency problems for connecting a remote machine to a display server are no longer a problem when a user connects his portable personal device to the display server. As Figure 1:8 shows, the display server is in the same room as the user, the user's personal device is physically accessible, and the network distance is short both in terms of physical distance and the number of network hops. Because the user is physically close to his personal device and has input hardware on that device, he may use the input hardware to interact with his applications. One key feature of this setup is that the user only needs the local network, which has a short network distance from the display server and is likely much faster than an Internet connection. This short distance, high-bandwidth connection means the user can have a rich interactive experience with little to no delay.

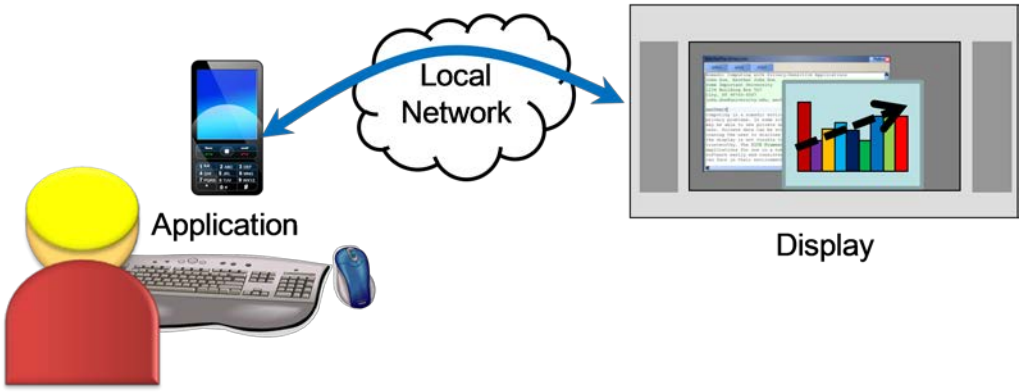


Figure 1:8—Nomadic user experience. The user carries his application with him and can interact with the application using the hardware on his personal device. The display server is a physically distant, but not remote, machine.

Unfortunately, with RDP, X11, and VNC, the display server is given complete control over the user's application. VNC still limits users to personal devices with larger screens and more resources.

Pushing provides some protection from malware because the display has does not have full control of the user's desktop. But, because applications completely trust input from the display server individual applications are still vulnerable. For example, imagine sharing a web browser's window to the display server. If the user is unaware that the display server has malware, and the malware recognizes the browser and is aware of a potential exploit, the display server could surreptitiously direct the browser to fall prey to that exploit.

RDP allows personal devices to prohibit all input from the display server, which helps to mitigate the potential malicious input exploit. However, an all-or-nothing policy for controlling input drastically limits a user's flexibility and power when annexing a display server. Typing up an email is unlikely to be exploitable, while selecting a destination email address could be. Applications should be allowed to accept some input, but limit other input. RDP also does nothing to protect against stolen output. As is shown in section 1.2.6, a similar all-or-nothing policy for output is insufficient. To better-protect the personal device the software must also be

aware of whether the user trusts the display server's input and output resources. Then applications can more intelligently filter the received input and their output.

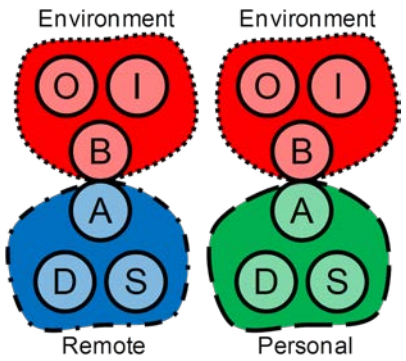


Figure 1:9—Use the web approach.

1.2.5 Use the Web

A common way of transmitting output and input is with HTTP and HTML. A web server generates the HTML which is then transmitted to a client machine and viewed in a browser on that machine. The machine executing the application may be a remotely-located web server or it may be a web server embedded in a personal device like the Personal Server [11]. For example, a user could use a kiosk at the mall to access his bank's website.

Using HTML and HTTP has several nice properties. Using these protocols can provide users with easy access to data, and physically light devices (either no device, or a handheld). These protocols can also provide some powerful interactions. In configurations such as MultiBrowsing [25] some collaboration can be had as well.

One large benefit that HTML has over VNC, RDP, and X11 is how it takes advantage of the display server's computing resources such as RAM and CPU. VNC, RDP, and X11 assume that the display server has limited CPU and RAM (useful for the mainframe era), while HTML assumes the display server (machine hosting the web browser) has plenty of CPU and RAM to hold a representation of a page and re-render it as necessary. Transmitting the page reduces network bandwidth and increases interactivity because the user can perform a lot of interactions within that page without further communication to the server. In particular, scrolling a window over RDP or X11 generates a lot of network usage as portions of the UI must be repainted.

Scrolling an HTML page generates minor network usage. A small personal device may be limited in RAM, CPU, and network bandwidth, so transmitting a representation of the UI to the display server can reduce those requirements while greatly increasing the screen space that the application can consume. Applications can then run on small devices while presenting a rich UI on large screens.

Unfortunately, HTML has some serious drawbacks. HTML is inconsistent which reduces the familiarity of applications. Rich interaction requires distributing code (e.g. JavaScript [20] or plug-ins such as Flash [1]). The browser may not be familiar to the user potentially because it may not have the requisite plugins installed for proper interaction. And, there is only some protection from malware. The two largest problems for HTML are code distribution and malware protection. The people who own a display server must maintain those display servers as well. Web browsers are complex applications that are frequently patched or upgraded. This frequent patching requires maintenance overhead. In addition, because the browser accepts JavaScript or plug-ins, which could potentially damage the display server, the display owner is likely to disable these browser features. If the user's web application requires JavaScript or a specific plug-in then his attempts to use that application will be frustrated. In addition, using a plug-in or JavaScript opens the user to all the problems introduced in section 1.2.3. Finally, using HTML is insecure for users; the user's sensitive data is frequently exposed to the display server, which could easily steal that data.

1.2.6 Brokered Access

A large problem with using HTML or distributing a UI is that accessing a remote site requires exposing the user's credentials directly to the display server. The user accessing his bank's website via a mall kiosk must supply his username and password to that kiosk which can

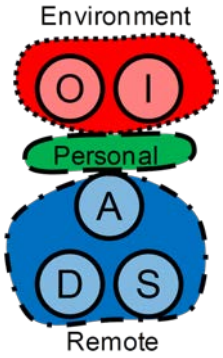


Figure 1:10—Brokered access approach.

easily steal those credentials. To overcome this problem, research by Oprea et al [21], Sharp et al [26], and Mobile Composition [25] use a brokered approach. In these cases the user brings a low-powered personal device to a display server, and the personal device is used to initiate a connection between the display server and a remote machine. This connection may be a VNC or HTML connection. For example, the user at the mall kiosk would bring his phone to the kiosk. The phone talks to both the kiosk and the user's bank website to establish a connection. At no point is the kiosk given the user's credentials, and the connection is a one-time-use connection. By using a third trusted device, the user's sensitive data can be protected and the display server cannot independently initiate a connection to the remote machine.

Sometimes the personal device and display server may generate and deliver temporary credentials to the user which the user can then enter into the display server. For instance, Hotmail [15] can generate a one-time-use security code and send it to the user's phone. Then the user can enter that security code in lieu of a password.

Research in brokered access focuses on using more resources on the user's personal device to provide greater protection. Figure 1:11 illustrates the software and hardware organizations for Oprea, Sharp, and Mobile Composition, respectively. Each of these configurations progressively adds functionality to the personal device or display server in an attempt to make the browsing experience more protected.

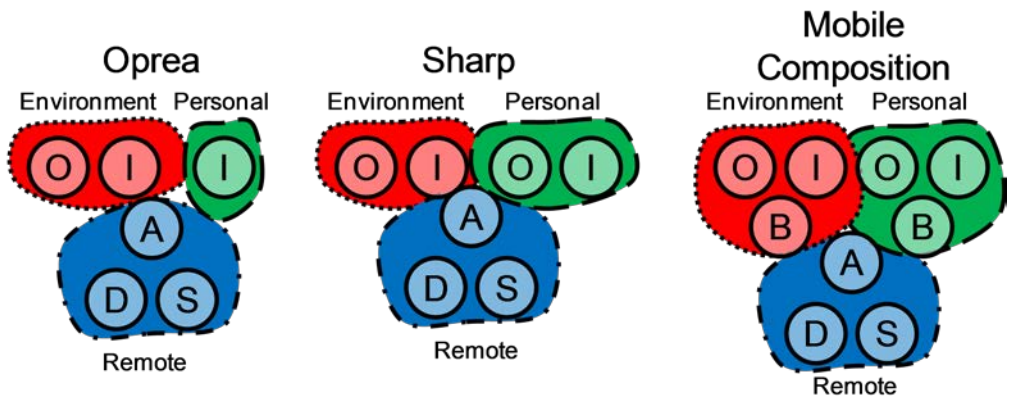


Figure 1:11—Research approaches to brokered access. Oprea filters some input, allowing only mouse clicks and some accelerator keys to come from the personal device. Sharp adds to Oprea by blurring out all text on the display server and showing it un-blurred on the personal device. Mobile Composition uses websites that are specially designed to show most content on the display server and sensitive content on the personal device.

Oprea uses the personal device to broker an augmented VNC connection between a display server and a remote computer. The connection is augmented so that the remote computer does not accept pointing input nor keyboard accelerators from the display server; the display server can only provide strictly textual input. The personal device is used to supply pointing input. Such limitations prevent the display server from accessing any resources independently of the user.

Unfortunately, because Oprea uses a VNC connection to a remote computer the user experience degrades with increased distances. Also, the display server is trusted to show all of the user's sensitive data and accept some sensitive input (e.g. sensitive text within an application).

Sharp extends Oprea's approach by rendering on the remote computer twice. The first rendering blurs all text and is sent to the display server. The second rendering does not blur anything and is sent to the personal device. The personal device has a much smaller screen so it

can readably show only a portion of the remote computer's screen. Consequently, the personal device shows the section of the remote screen that is near the cursor.

Sharp protects sensitive data but overdoes the protection. Most of the text shown on the screen is unlikely to be sensitive and requiring the user to look at the personal device to read any text is burdensome. Applications need to be able to intelligently filter information so that only sensitive data is protected. In addition, Sharp suffers from the problems of using VNC over large distances.

Mobile Composition uses HTML as the UI distribution mechanism instead of VNC. A browser on the display server renders most of the page. However, sections of the page may be tagged by the web server as sensitive and encrypted so that only the personal device's browser may decrypt that data. This is a more intelligent approach to protecting sensitive data.

Regrettably, by using HTML Mobile Composition suffers from all the problems introduced by using the web for interaction (section 1.2.5). Requiring all websites that use sensitive data to change so that they conform to this new standard is also a heavy burden.

1.2.7 Summary

None of the current approaches are sufficient to meet a nomadic user's needs. However, they do provide knowledge that can be used to craft a more effective approach. By observing the different facets of each of the design strategies we can produce a set of *technical requirements* for the nomadic computing vision.

First, the vision will require a self-contained device. The self-contained device gives users access to their data in any situation, especially in situations without other computers. These devices also eliminate software version conflicts because the user carries the data with them. In addition, these devices can protect sensitive data from malicious display servers.

Second, the vision will require transmitting output and input and *not* data or applications. This approach gives users the ability to annex significant input and output resources in their environment while allowing the user to keep his data, applications, and settings with him wherever he travels.

Third, a brokered access approach will give users richer interaction (via transmitting input and output) while simultaneously protecting a user's sensitive data. Because the user may be required to interact with sensitive data, the portable device will require its own input and output modalities.

Fourth, the application should be located near where the user is working. Long network distances can degrade the user experience while short network distances (both in distance and network hops) can perform well.

Fifth, because the user is carrying a portable *personal device* and that device is likely going to communicate via a wireless technology, the vision should help maximize battery life while simultaneously maximizing the interactive experience. Maximizing battery life will involve a protocol which minimizes the wireless radio and CPU usage.

Sixth, the vision should include a UI transmission protocol which reduces network bandwidth compared to technologies such as RDP, X11, and VNC but also provides a consistent output. Such a protocol is likely to involve transmitting a significant portion of the UI to the display server and then periodically updating that UI in bandwidth-friendly ways.

Finally, the vision should also support multiple users interacting simultaneously at a display server. These users may be interacting in independent applications or in the same application.

So, in summary, the technical requirements for the vision of a nomadic environment include:

- A portable, self-contained device
- Transmitting input and output
- Brokered display annexation
- Short network distances to the data and applications
- Minimized radio and CPU usage
- A graphical protocol that helps minimize network usage
- Collaboration among users

An effective nomadic environment will meet all six of the user requirements as well as these technical requirements.

1.3 Research Model

Now that the user, display server owner, and technical requirements have been distilled, this section will present a new interactive model to research.

1.3.1 Core Model

The interactive model starts from the perspective of a user who carries a personal device wherever he travels. The personal device is the primary computing device for the user, and the core of applications always stay on the personal device. In environments which support annexation the user may annex additional screen space and input.

Because the personal device is portable, it relies on battery power for continued operation. Although the user could plug the device into a wall outlet, such a requirement is restrictive on the user's mobility and a restriction on the kinds of rooms the user may annex other devices within. With battery power a significant concern, the rich interaction (in particular the rendering effort) should be pushed to the other devices in the room in ways that are safe for the user's device regarding infection and exploitation.

1.3.2 Self-contained Interaction

One nomadic interaction style is for users to interact exclusively with their personal device, as is illustrated in Figure 1:12. This style is common to how people interact with their cell phones. Such interaction allows a user to gain access to his data, settings, and applications whenever and wherever he may be located.

Figure 1:12 – Mobile interaction in a stopped car. All application processing happens within the handheld (highlighted).

1.3.3 Annexing a Display Server

Self-contained interaction, although possible, may not be as fast or satisfying as desktop interaction. In many ways small mobile devices represent a step backward in usability and efficiency [20]. The author proposes a way for a user's computing to stay mobile, but to also gain the richness and efficiency of a desktop experience.

In this nomadic model, the author envisions users expanding to a desktop experience by annexing a display server and input devices from their handheld. This annexation can happen in a wide variety of environments which have a display server available. As illustrated in Figure 1:13, the user's desktop will be a display server instead of the primary computing device. A

projector in a conference room would also be a display server. Or, a team-room may have an interactive whiteboard which may be annexed, as in Figure 1:14.

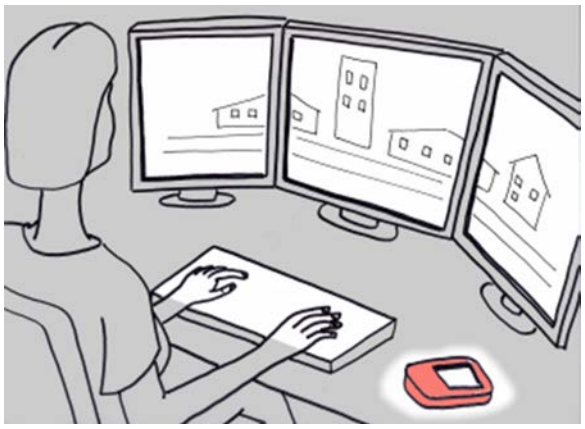


Figure 1:13 – Desktop interaction. All application processing happens within the handheld (to the right of the user).

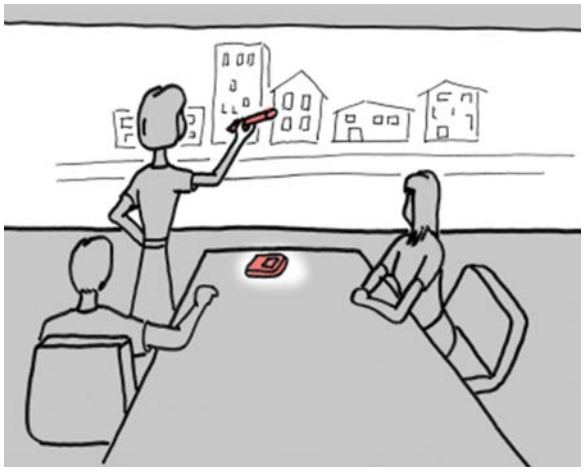


Figure 1:14 – Collaboration on a large screen. All application processing happens in the handheld.

In other situations the display space may not provide any input devices which the user may be able to annex. For example, a conference room may have a large screen, but the screen is not interactive and is unlikely to have a mouse or keyboard. The user should be able to annex such a screen and should be able to interact with his applications via the input hardware on his

portable device. Effectively, the input from the portable device is redirected to the windows shown on the portable device.

Key to this annexing scenario is that the user's personal device continues to perform the essential computation for the user's applications. The user's data, settings, and applications always remain on the personal device and are never given to an annexed display server. Instead, the UI for the application is transmitted to the display server where the UI is rendered.

The UI presented on the annexed displays is supplied in an abstract format by the software on the personal device. The display server performs the computation necessary to render the abstract UI representation as pixels. The software on the personal device can make changes to the abstract UI representation; those changes are transmitted to the display server and reflected in updated pixels.

The input to the UI is also represented in an abstract format and transmitted to the personal device for processing. When the input is pointing input, all the input is transmitted relative to the screen coordinates. In the case of a display server not having input devices, the user's personal device will likely transmit a mouse pointer's location to the display server, but none of the mouse clicks.

The author believes that such an approach for transmitting a UI can greatly reduce the computation necessary on the personal device, which can reduce the radio, CPU, and ultimately the battery requirements.

1.3.4 Privacy-Aware Annexation

In this model the user would annex display servers owned by other people, much like users currently visit websites owned by other people. These annexed display servers could have malicious software installed on them. Not only might these display servers attempt to infect the

user's device, because the user is distributing application UIs, a malicious display server could take advantage of additional exploits such as stealing output or input, or producing false output or input.

A significant design consideration when choosing to transmit only the UI and not the user's data to the display server is to protect the user's data from being infected or stolen by a malicious display server. However, the output from an application can still contain sensitive data (e.g. email addresses, social security numbers), or could request sensitive data (e.g. login credentials). Via such UIs, a malicious display server could glean sensitive data from the user.

To compensate for this potential breach of the user's sensitive data, the author envisions a UI framework designed to allow privacy-aware applications to operate and help protect the user. Such privacy-aware applications can be sure not to display or request sensitive data through annexed display servers. Instead, the privacy-aware applications always show or request data via the personal device, which is always trusted.

Some display servers, such as the user's desktop, are likely to be devices which the user trusts. If the user can designate to his personal device whether he trusts a display server, then applications can adjust their input and output based on the annexed display server's trust level.

1.3.5 Collaborative Interaction

Nomadic users also need to collaborate with each other. In this new research model, the author envisions users being able to simultaneously annex screen space and interact in a wide variety of locations and configurations. For example, users may meet at a small display space and have a closer discussion, while other meetings may occur in a more sophisticated presentation scenario which allows audience members to capture the presentation as well as point at the shown information or even provide new information.

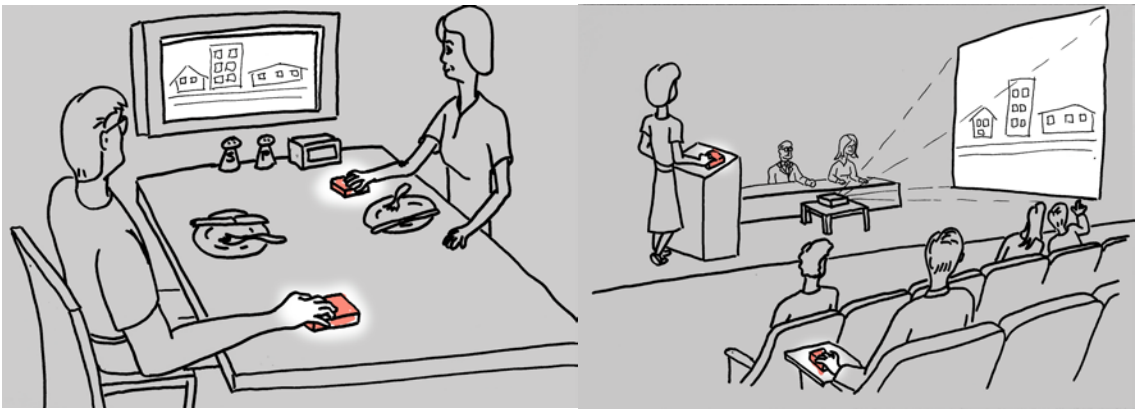


Figure 1:15 – Collaboration in a restaurant or a public forum. Multiple people can interact, but personal processing happens in the handheld.

In these collaborative situations some control may be necessary. For example, a presenter may not want viewers in the room to show content while he is presenting, but later he may open the floor to more discussion where audience-supplied content may be warranted.

The problem is how the users should exhibit control. The display server could supply software which allows the presenter to control who may show content on the display server. However, if that presenter travels to various locations then each location may have different software for controlling the display space. Having to learn a new piece of control software at each location may be difficult and could lead to failures in controlling the environment.

To remedy this problem, the author envisions users being able to carry their own control software with them wherever they travel. Then the user could learn one piece of software and use it in any environment regardless of the software powering the display server.

Unfortunately, the control software cannot be transmitted to the display server; otherwise that software may damage the display server. Consequently, the way control is exhibited must remain on the personal device but the effects must still be transmitted to the display server.

1.4 Research Implementation

With this new research model in mind, the author executed a research plan to help elucidate the validity of various aspects of this model.

1.4.1 Idealized Framework

The first approach is to substantially implement an ideal annexing environment. The implementation will include a screen and input annexing framework, along with a platform independent annexing protocol.

The protocol is designed to transmit UIs as an abstract collection of drawing commands (called a scene-graph), and to transmit input to the personal device in an abstract format. The set of drawing commands is rich and will include text, lines, rectangles, ellipses, polylines, clipping, and transforms.

The UI from an application is able to seamlessly move (i.e. without the application needing to be informed) from a mobile device to a large display surface. The decision to move a UI from the personal device to a display space is under the user's control.

Part of this annexing protocol is allowing users to supply input from their personal device in various ways. For instance, input supplied from a touch pad, a mouse, or a stylus may vary greatly in how it is redirected to windows shown on a large display space; a stylus involves direct interaction with widgets, but translating that input to an abstract mouse pointer on a large display space may require some extra thought.

Portable devices rarely have more than one direct pointing apparatus, so the user must swap between different modalities when interacting with windows shown on the personal device versus those shown on an annexed display space. However, experimenting with devices which have at least two independent dedicated pointing input techniques may prove beneficial.

A large part of the purpose of this aspect of the research is as a proof of concept. Primarily, the author will observe and document how such an environment could be constructed. But another aspect is to demonstrate whether or not the ideas inherent in distributing a UI using scene-graphs will provide a rich interactive experience while outperforming RDP, X11, and VNC. Part of measuring performance is to see how network (and thus radio) and CPU usage compare with the other UI distribution technologies, with expected results greatly exceeding those technologies.

1.4.2 Nomadic Privacy

The next large step in the research is to build a privacy-aware framework into the idealized UI toolkit. This privacy-aware framework will give users a simple approach to selecting a trust level for a display space (e.g. do they trust the output device or not and do they trust the input device or not), and then the toolkit will inform applications of the trust level when the application creates a window on a distrusted display space or the user moves a window to such a display space.

The purpose of this research is to develop a proof of concept and to document its design. The author is aware of no other UI toolkits which collect such information from users and make it available to applications at the windowing toolkit level. This dissertation will show the considerations that should be taken into account when designing such a system. These kinds of design decisions are shown as independent of the UI sharing toolkit and may be implemented in other windowing toolkits.

1.4.3 Legacy Support

Requiring developers to rewrite all code to support a new rendering toolkit may be unreasonable; there is a significant base of existing code which users may want to employ in this

new nomadic environment. Consequently, legacy software should be usable in this nomadic environment.

The author developed a framework that supports existing applications through a screen capture method similar to the Remote Frame Buffer (RFB) protocol which underpins VNC [23]. The frame buffers are embedded within an abstract representation of a UI similar to that used in the idealized framework. However, this framework does not support the wide variety of rendering commands, also known as *machine-readable primitives*. Instead, this framework will only support the *human-readable primitives:* audio, video, and images (where screen captures are a form of image). This framework also supports clipping, and transforms for positioning and presenting those primitives.

By supporting legacy software users are able to continue to operate familiar hardware and software while still operating in a nomadic environment. This framework will also more adequately demonstrate the usefulness of this new nomadic interaction model.

1.4.4 Display Space Control

As discussed in section 1.3.5, when users are collaborating some control over the display space may be necessary. Unfortunately, a user may need to exhibit control over a communal display space that neither he nor someone he trusts owns.

This segment of the research is building an API into the framework which allows users to exhibit significant control over the presentation and arrangement of windows on a display space without installing new software on the display server or getting/supplying credentials to the display server owner. Such an API will allow users to carry their own familiar control software with them to any display server which supports the API.

This software will further demonstrate the carry-your-software-with-you paradigm. Rather than just carrying software to a display space and distributing the UI to the screen, users can now exhibit some control over display spaces they do not own. This is demonstrated in the variety of control software which can be exercised on such screens.

1.4.5 Other Areas of Research Open

Many other avenues for research open up with this framework in place. These include screen-size-flexible widgets, input-type agnostic widgets, the best privacy-aware distributed UI applications, as well as measuring the usability of the privacy-aware toolkit and API, legacy application-sharing techniques, or display-space-control software.

Screen-size-flexible widgets would allow developers to create applications which are easily scalable to a wide variety of extremes in dimensions. An application could then easily be written which shows itself properly on a handheld screen as well as a wall-sized display.

Mice, pens, styluses, fingers, laser pointers, and gestures are a small set of the possible input modalities. Trying to develop widgets and applications which are adapted to all possible input types introduces significant complexity into an application. Developers need tools which help them write widgets that act properly regardless of the input system with as few considerations to the variations in input types as possible.

Users are the ultimate beneficiaries of the privacy-aware framework, but researchers currently do not know what the best presentations are for a variety of privacy-protection techniques. Developers need opportunities to develop an assortment of UIs and test them on users to see which interfaces protect well, are intuitive, and have the necessary features for users. Researchers should also develop a set of tools which help developers make better decisions with regard to privacy awareness as this knowledge is gained.

Although users are the ultimate beneficiaries of a privacy-aware framework, developers need an API which helps them effectively write privacy-aware applications. Some research could be performed which evaluates the effectiveness of various privacy-aware toolkit APIs.

Or, consider how to share legacy software onto large screen spaces. There are a variety of GUI techniques that could be employed which allow the user to choose which windows to share and to arrange those windows on the shared display space. A usability study of these different techniques may be fruitful toward helping users quickly and effectively share windows.

On a related note, when users are interacting at a shared display space and one user enforces a particular control technique, the effectiveness of that technique could be evaluated. This kind of measurement is largely independent of the protocol which makes it possible to control the display space. However, some features may be added to the protocol in order to support some needs that may be elucidated with this kind of research.

As has been shown, there are many areas for potential research within the new nomadic interaction model presented in this dissertation. As knowledge is pursued in these areas and others, more avenues are likely to emerge toward the end of helping users take their data, settings, and applications with them wherever they may go.

1.5 Dissertation Outline

Chapters 2 through 5 are two published papers and two yet-to-be published papers. Each published chapter includes its publication reference.

Chapter 2, “XICE Windowing Toolkit: Seamless Display Annexation,” introduces the concept of cellphone-sized devices annexing large display spaces. This chapter details core design choices within the XICE framework which is the idealized framework mentioned in section 1.4.1. It also demonstrates how efficiently a scene-graph presentation system can perform

with respect to distributed rendering technologies such as VNC, RDP, and X11. This chapter was originally published in the ToCHI journal and has been formatted to comply with the styles in this dissertation [2].

Chapter 3, “Privacy-Aware Shared UI Toolkit for Nomadic Environments,” explains how the privacy-aware tools operate within the XICE framework. These tools help protect users when annexing potentially malicious display servers by restricting output and input to non-sensitive data. This chapter explores many possible applications that have been or could be developed with such a framework in place. With these new privacy-aware hooks, users can more confidently annex machines they do not own, get their desired tasks accomplished, with greater assurance that their data, settings, applications, and device will not get damaged or exploited. This chapter was originally published in the journal “Software: Practice and Experience” and has been formatted to match this dissertation’s styles [3].

Chapter 4, “SPICE: Lightweight, Media-rich, Screen Annexation,” explores an alternate annexation framework to XICE which transmits only images, audio, video, and screen shots. By transmitting these primitives, existing devices can participate in the screen annexation experience. In addition, applications that are aware of the SPICE protocol can show even richer UIs on the annexed screens. As a result, users can carry their existing data, settings, and applications with them and share the output from applications on annexed screens.

Chapter 5, “Window Brokers: Collaborative Display Space Control,” demonstrates how users can exhibit some control over display spaces they do not own. When a user travels to a room someone else owns, and he has never annexed the display in that room before, the user may need to ensure certain properties about how that display is used. For example, if the user is giving a presentation, he may not want anyone else to be able to annex the display server

simultaneously; he may be concerned that audience members may use the display server to disrupt the presentation. To ensure this user can effectively and confidently annex and control that display space, the user could carry his control software to employ with the display space rather than using the pre-existing software in that room.

Chapter 6 summarizes this research's contributions and discusses potential areas for future research.

1.6 REFERENCES

1. Adobe Systems, Adobe Flash, http://get.adobe.com/flashplayer/, 1996, accessed June 2010.
2. Arthur, R.B. and Olsen, D.R. 2011. "XICE windowing toolkit: Seamless display annexation." *ACM Transactions on Computer-Human Interaction.* ToCHI volume 18, number 3, ACM, New York, NY, Article 14 (August 2011), 46 pages. DOI=10.1145/1993060.1993064
3. Arthur, R.B. and Olsen, D.R. "Privacy-aware shared UI toolkit for nomadic environments." *Software: Practice and Experience*. May 2011. John Wiley & Sons, Ltd. DOI=10.1002/spe.1085
4. Biehl, J. T., Baker, W. T., Bailey, B. P., Tan, D. S., Inkpen, K. M., and Czerwinski, M. 2008. "Impromptu: a new interaction framework for supporting collaboration in multiple display environments and its field evaluation for co-located software development." In *Proceeding of the Twenty-Sixth Annual SIGCHI Conference on Human Factors in Computing Systems* (Florence, Italy, April 05 - 10, 2008). CHI '08. ACM, New York, NY, 939-948.
5. Cisco Systems, Inc. WebEx, http://www.webex.com/, 1997, accessed June 2010.
6. Citrix Systems, Inc. Citrix Online, http://www.citrixonline.com/, 1997, accessed June 2010.
7. Dropbox, http://www.dropbox.com/, accessed July 2010.
8. Everitt, K., Shen, C., Ryall, K., and Forlines, C. 2006. "MultiSpace: Enabling Electronic Document Micro-mobility in Table-Centric, Multi-Device Environments." In *Proceedings of the First IEEE international Workshop on Horizontal Interactive Human-Computer*

Systems (January 05 - 07, 2006). TABLETOP. IEEE Computer Society, Washington, DC, 27-34.

9. Flanagan, D. 2006 *JavaScript: the Definitive Guide.* O'Reilly Media, Inc.
10. Gosling, J., Joy, B., Steele, G., and Bracha, G. *Java Language Specification, Second Edition: the Java Series. 2nd*. Addison-Wesley Longman Publishing Co., Inc. 2000.
11. Izadi, S., Brignull, H., Rodden, T., Rogers, Y., and Underwood, M. "Dynamo: a public interactive surface supporting the cooperative sharing and exchange of media." *User Interface Software and Technology (UIST '03)*, ACM 2006, 159-168.
12. Johanson, B., Fox, A., and Winograd, T. 2002. "The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms." *IEEE Pervasive Computing 1, 2* (Apr. 2002), 67-74.
13. Johanson, B., Ponnekanti, S., Sengupta, C., and Fox, A. 2001. "Multibrowsing: Moving Web Content across Multiple Displays." In *Proceedings of the 3rd International Conference on Ubiquitous Computing* (Atlanta, Georgia, USA, September 30 - October 02, 2001). G. D. Abowd, B. Brumitt, and S. A. Shafer, Eds. Lecture Notes in Computer Science, vol. 2201. Springer-Verlag, London, 346-353.
14. Liu, Z. "Lacome: a Cross-Platform Multi-User Collaboration System for a Shared Large Display," Computer Science, University of British Columbia, 2007. http://hdl.handle.net/2429/378.
15. Microsoft Corporation, Hotmail, http://www.hotmail.com/, accessed August 2010.
16. Microsoft Corporation, Roaming User Profiles, http://msdn.microsoft.com/en-us/library/bb776897(VS.85).aspx, accessed August 2010.
17. Microsoft Corporation, Silverlight, http://www.microsoft.com/silverlight/, accessed June 2010.
18. Microsoft Corporation, Microsoft Office Groove 2007 renamed SharePoint Workspace 2010, http://office.microsoft.com/en-us/sharepoint-workspace/microsoft-sharepoint-workspace-2010-FX101825648.aspx, accessed August 2010.
19. Microsoft Corporation, Windows Live Sync, https://sync.live.com/, accessed July 2010.

20. Norman, D. A., and Nielsen, J. “Gestural Interfaces: A Step Backward in Usability.” http://www.jnd.org/dn.mss/gestural_interfaces_a_step_backwards_in_usability_6.html accessed June 2011. Also to be available in ACM Interactions.

21. Oprea, A.; Balfanz, D.; Durfee, G.; Smetters, D.K., “Securing a remote terminal application with a mobile trusted device,” *Computer Security Applications Conference, 2004. 20th Annual* , vol., no., pp. 438-447, 6-10 Dec. 2004.

22. Prante, T., Streitz, N., and Tandler, P. 2004. “Roomware: Computers Disappear and Interaction Evolves.” *Computer* 37, IEEE, 12 (Dec. 2004), 47-54.

23. Richardson, T., Stafford-Fraser, Q., Wood, K. R., Hopper, A., “Virtual Network Computing,” *IEEE Internet Computing*, Vol. 2, No. 1, 1998.

24. Scheifler, R. W. and Gettys, J. “The X Window System,” *ACM Transactions on Graphics*, vol 5(2), (April 1986), pp 79-109.

25. Sharp, R., Madhavapeddy, A., Want, R., and Pering, T. 2008. Enhancing web browsing security on public terminals using mobile composition. In *Proceeding of the 6th International Conference on Mobile Systems, Applications, and Services* (Breckenridge, CO, USA, June 17 - 20, 2008). MobiSys '08. ACM, New York, NY, 94-105.

26. Sharp, R., Scott, J., and Beresford, A. R. 2006, Secure Mobile Computing via Public Terminals. In *Proceedings of the International Conference on Pervasive Computing* (PerCom 2006). IEEE Computer Society, 238-253.

27. Shen, C., Everitt, K., and Ryall, K., “UbiTable: Impromptu Face-to-Face Collaboration on Horizontal Surfaces.” UbiComp 2003, Springer Berlin/Heidelberg, 2003, pp 281-288.

28. Streitz, N. A., Geißler, J., Holmer, T., Konomi, S., Müller-Tomfelde, C., Reischl, W., Rexroth, P., Seitz, P., and Steinmetz, R. 1999. “i-LAND: an interactive landscape for creativity and innovation.” In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*: the CHI Is the Limit (Pittsburgh, Pennsylvania, United States, May 15 - 20, 1999). CHI '99. ACM, New York, NY, 120-127.

29. Tritsch, B. Microsoft Windows Server 2003 Terminal Services, Microsoft Press 2003.

30. U3 LLC., U3 Launchpad, http://www.u3.com/ (dead link), see http://en.wikipedia.org/wiki/U3, accessed July 2010.

31. Want, R., Perins, T., Danneels, G., Kumar, M., Sundar, M., and J. Light. “The Personal Server: Changing the way we think about Ubiquitous Computing.” *Ubiquitous Computing* (UbiComp ‘02), Springer Verlag, (2002).

32. Wigdor, D., Jiang, H., Forlines, C., Borkin, M., and Shen, C. 2009. “WeSpace: the design development and deployment of a walk-up and share multi-surface visual collaboration system.” In *Proceedings of the 27th International Conference on Human Factors in Computing Systems* (Boston, MA, USA, April 04 - 09, 2009). CHI '09. ACM, New York, NY, 1237-1246.

Chapter 2 XICE Windowing Toolkit: Seamless Display Annexation

Richard Arthur and Dan R. Olsen, Jr. 2011. XICE windowing toolkit: Seamless display annexation. *ACM Trans. Comput.-Hum. Interact.* 18, 3, Article 14 (August 2011), 46 pages. DOI=10.1145/1993060.1993064

ABSTRACT

Users are increasingly nomadic, carrying computing power with them. To gain rich input and output, users could annex displays and input devices when available, but annexing via VGA cable is insufficient. This article introduces XICE, which uses wireless networks to connect portable devices to display servers. Network connections eliminate cables, allow multiple people to share a display, and ease input annexation. XICE mitigates potentially malicious input, and facilitates comfortable viewing on a variety of displays via *view-independent coordinates*. The XICE distributed graphics model greatly reduces portable device CPU usage and extends portable device battery life.

2.1 INTRODUCTION

Computing becomes increasingly nomadic as Moore's law continues to push more computing power into smaller devices. Portable devices now have significant computing resources and are becoming primary personal computing devices.

The interactive input and output of personal portable devices are limited by their size. A user could carry around a larger screen, keyboard, and mouse for richer interaction, but not in a handheld form-factor. Keyboards have shrunk to fit small devices but are difficult or slow to use. The screens on small devices cannot provide enough visual output to meet a typical user's needs, even with a lot of panning and zooming.

A richer nomadic interactive experience would be to carry data and applications in a personal device, then annex display servers such as screens and input devices when necessary [46, 51, 54]. Screens are accessible in many places such as offices, kiosks, home entertainment systems (e.g. televisions, projectors, etc.), and conference rooms (Figure 2:1). Users do not need to carry screens if they can effectively use available screens. Keyboards and mice are inexpensive, so they too can be readily available for use.

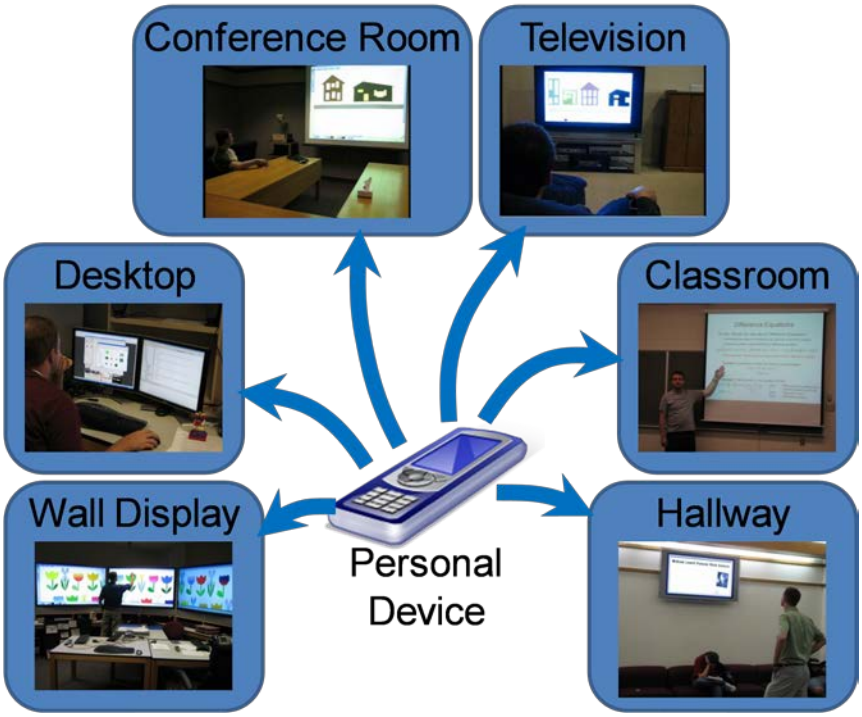


Figure 2:1 – Display servers a nomadic user could annex.

This article discusses the XICE (eXtending Interactive Computing Everywhere—pronounced "zice") Windowing Toolkit, which seamlessly distributes the user interface (UI) of applications to annexed displays and directs input from annexed devices to applications. XICE operates by letting personal devices annex display devices through wireless networks instead of through direct cable connections like Video Graphics Array (VGA) and Universal Serial Bus (USB). The user experience becomes much more flexible when the wireless network is used.

This architecture requires one or more *display servers*, each of which includes a processor and software to transmit user input to and show visual output from applications running on a personal device.

In a nomadic environment, users should be able to easily annex screens and input from display servers. These display servers should have a stable annexing protocol. Users should also be protected from malicious display servers. In particular, the protection should address attacks specific to distributing input and output. Software for nomadic users should require minimal overhead for annexing display servers to maximize the personal device's battery life. Display servers vary according to viewing distance and screen size, and applications must adapt to these differences. In addition, multiple people should be able to annex a display server simultaneously.

2.2 Nomadic Computing

A nomadic user goes various places and uses a variety of display servers. She may experience three *nomadic situations*: personal device alone, annexed screen and input, and annexed screen alone.

The *personal device alone* situation involves using the personal device (e.g., a smartphone) directly as seen in Figure 2:2. This setup provides the nomadic user with the requisite mobility but poses serious interaction problems with regard to the personal device's small form factor.

Figure 2:2 – Mobile interaction in a stopped car. All application processing happens within the handheld (highlighted).

The *annexed screen and input* situation is illustrated in Figure 2:3. When a nomadic user arrives at her personal workspace, the personal device should annex a mouse, a keyboard, multiple screens, and other available interactive resources. Though applications run on the personal device, her interaction with them should feel like she is using a desktop.

Figure 2:3 – Desktop interaction. All application processing happens within the handheld (to the right of the user).

A user may move from her private workspace to a collaborative, company-controlled space that includes display servers like the wall screen in Figure 2:4. In this environment, screens may be larger and farther away and input devices may differ, but all of the inputs can be fully

trusted. Although the environment is different from the personal workspace setup, this scenario is also an annexed screen and input situation.

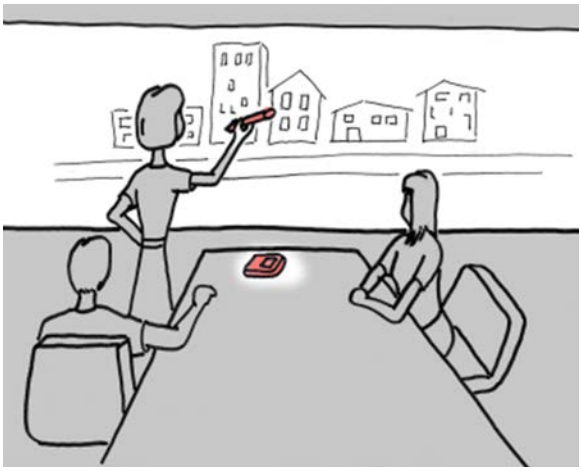


Figure 2:4 – Collaboration on a large screen. All application processing happens in the handheld.

The *annexed screen only* situation applies when annexed input cannot be trusted. A user may move to a collaborative setting, such as the restaurant or public meeting in Figure 2:5. Other institutions control the display servers and input devices. In these scenarios, the personal device accepts input only from itself so that annexed input devices cannot be used as attack vectors.

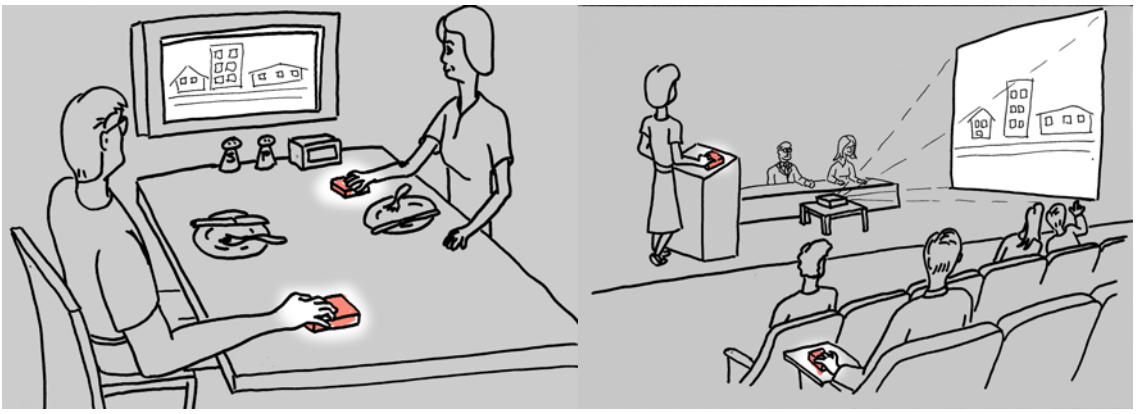


Figure 2:5 – Collaboration in a restaurant or a public forum. Multiple people can interact, but personal processing happens in the handheld.

Not only are input threats present in such public situations, but the screen will also be visible to multiple people. Users may want to annex these screens, but will likely not want

certain pieces of sensitive data (e.g. emails, financial data, file names) to be shown so publicly. Either people in the room or the display server itself could steal that data.

In many collaborative situations, multiple people may annex a display server simultaneously. Each user may want to interact with his or her applications independently on the same display (i.e., each person can bring up various single-user applications). Within these multi-user configurations, each user may choose a different nomadic situation. For instance, if a display server only has one mouse, then only one user may annex that mouse's input (the annexed screen and input situation); other users can annex the screen (the annexed screen only situation) and opt to use the input on their personal devices instead. The display server must accommodate these independent requests.

2.2.1 Solution Requirements

The nomadic situations have seven important challenges that must be addressed by a user interface toolkit for nomadic computing:

- Wireless display connectivity and seamless distribution of application output;
- Disparate display resolutions, sizes, and viewing conditions;
- Input acceptance from a variety of sources;
- Avoidance of attack from annexed input devices;
- Privacy sensitivity for application output;
- Myriad software installations;
- Limited battery life and processing power;
- Shared public display space among multiple users.

2.2.1.1 Wireless Display Connection

Using a cable to physically tie a personal device to a display is severely limiting. In many situations, including business meetings, such a setup is infeasible. Besides unnecessarily tethering the user to a particular display, other problems with using a cable persist. First, large collaborative displays can easily have more pixels (e.g. 7000x2000) than a single VGA or Digital Visual Interface (DVI) cable can handle. Second, in small handheld form-factors, the size of the

monitor connector itself becomes prohibitive. Third, if a display is located some distance away from the user (as with a podium and a projector), the user must find the display's physical cable, or the display's owner must make the connector obvious and useable. Fourth, multiple people cannot use a single display without an awkward and time-consuming cable exchange. In addition, multiple people cannot show and interact with information on the display simultaneously. Fifth, with multiple people frequently connecting and disconnecting, the connector is susceptible to fatigue and failure.

XICE connects to display servers using Wi-Fi and the Internet. This capability provides a highly flexible software-defined display mechanism that can readily adapt to a variety of display services. Because XICE uses the network, it can handle large displays with an arbitrary number of pixels and dimensions, and users are not tethered to a specific location. A network-based protocol also enables multiple personal devices to simultaneously annex a single display server, and a single personal device to simultaneously annex multiple display servers.

2.2.1.2 Varying Displays

Nomadic users will interact with displays of different sizes, ranging from a smartphone's screen to large wall displays. Nomadic users also experience multiple viewing conditions. A personal device's screen is typically viewed from about 12 inches. For the display at a workstation, the viewing distance is about 24 inches. Conference room viewing can easily range from 6 to 50 feet. Typical 10-pixel-tall text on a desktop will look miniscule on a high-resolution wall-sized screen in a conference room.

In XICE, each display server is configured with the pixels-per-inch resolution and the normal viewing distance. These two pieces of information describe a new rendering and input coordinate space called *view-independent coordinates* (VIC). VIC enables the application and

the toolkit to communicate in terms of user-perceived sizes rather than physical or pixel dimensions. VIC provides the necessary adaptation to make text and application UIs readable in each viewing situation. If the user has poor vision, she can configure her device to automatically scale up the application's presentation.

2.2.1.3 Accepting a Display's Input

An annexed display server may provide input devices, such as a mouse or a keyboard. The display server offers a full QWERTY keyboard and a typical mouse for a personal device in the desktop situation (Figure 2:3). If the personal device just uses VGA to connect to a screen, the personal device will not be able to accept input through the same cable; to accept input the personal device must have another connection port, and the user will have to find and connect that cable as well. Because XICE uses Wi-Fi, sending input from the display server to the personal device is straightforward and adaptive.

Applications on a personal device must smoothly and rapidly adjust to each nomadic situation and to dynamically shifting input sources. To facilitate quick and consistent annexation, the toolkit—not the application—must manage the input bindings. Some display servers will not have input. Others cannot be trusted, as described in the next section, and must be protected against.

2.2.1.4 Protecting Nomadic Users

Annexed display servers may be in places that are obviously public, such as large conference halls or mall kiosks, but they may also be in places that seem private, such as hotel rooms. Display servers in all of these locations may be malicious because their owners are malevolent or because previous users have infected the servers with malware. When distributing

a UI, any combination of four serious threats can surface: stolen input, false input, stolen output, and false output.

Stealing input or output is an effort to capture the user's sensitive data. Falsifying occurs when the display server either alters user data or otherwise coaxes a user into situations where the display server can steal her sensitive data. XICE takes a large step toward mitigating these threats by preventing the personal device from accepting input from a distrusted display server. In such situations, the personal device becomes the only source of input to applications. To protect the user, the default setting is to always reject display server input. Using an options dialog on the personal device, the user can inform her personal device that the annexed input and/or output of a particular display server is trusted. The UI toolkit (XICE) must explicitly inform applications when devices are distrusted. Applications must also inform the toolkit about sensitive input and output so that interaction can be appropriately redirected based on the trust settings. This cooperative communication about sensitive data and trust does not exist in current toolkits.

2.2.1.5 Differing Software Installations

For a nomadic architecture to be widely adopted, the display services must be standardized. If servers are embedded in screens or smart keyboards, behavior and protocols must be static. Requiring users to check software compatibility before annexing a server is unacceptable. Many proposed display annexing solutions do not support a lightweight service.

When a user annexes a display, she wants familiar window management behavior. To provide consistency, standards for window management need to be developed. Standards built around human-consumable data types, such as drawings, are more likely to remain constant than

those built around programming environments [43]. The XICE solution is a lightweight service that distributes drawings, not code.

2.2.1.6 Limited Battery Life and Processing Power

When users operate nomadically, their personal devices are used to annex display servers using a wireless network. To properly annex a display server, the personal device's software must translate rendering commands for the network and transmit them. In addition, the personal device must accept similar transmissions containing input from the display server and translate the commands into input applications accept. The consequent radio and CPU overhead has the potential to greatly reduce the battery life of the personal device.

Radio power requirements are related to distance and usage. In the illustrated scenarios, the distances are short (less than 10 meters), so network latency and radio power requirements are both low. As this article later demonstrates, a protocol that distributes scene-graph changes will further reduce bandwidth demands. In addition, using scene-graphs to push graphics rendering off the personal device onto display servers sharply reduces CPU demands and consequently reduces battery requirements.

The user could plug their personal device into a power source within a room. Such an action would obviate the need for low power consumption. However, this approach requires that rooms prepare accessible power sources and that users find those sources. Regardless, the standard for nomadic computing should minimize power consumption for situations when devices operate on battery power.

Obviously, many other power management issues affect development for and use of personal devices. This article is concerned solely with the load imposed by the user interface architecture.

2.2.1.7 Support Multiple Simultaneous Users

Part of why people are nomadic is for collaboration with other people. People need to be able to discuss data while using a shared display space. A display space must be able to support multiple people showing data on it at the same time, and it must support independent input for each user. For instance, with three people at a display space, each user would need an independent pointing device so that he or she may interact independently with his or her own applications.

2.3 Prior UI Distribution Technologies

Wireless connectivity necessitates a protocol for transmitting visual information from the personal device to a display. Several existing technologies could be used to distribute a user interface across a network. These solutions fall into three broad categories: distributing data, distributing code, and distributing graphics.

2.3.1 Distributing Data

One way to interact with data on an annexed screen is to send that data to the display server and have an application there interact with that data. This approach is employed manually via thumb drive in the Dynamo [28] multi-user environment and automatically via central server in iRoom [30]. Sending data to the display server greatly reduces the computing requirements of personal devices by offloading all but the long-term storage requirements. Unfortunately, people often have custom-produced software they want to use on the displays. Expecting all display servers to have exactly the same software or equivalent substitute software is unreasonable. And, any software incompatibilities between the personal device and the display server disrupt the user experience. To function effectively, display servers executing application code must have

regular software updates and be compatible with all applications users want to run. This requirement creates a tremendous deployment burden.

2.3.2 Distributing Code

Another option is that the user could transfer her custom software to the display server for execution. Applications could be transmitted in whole (like Equalizer [19] or blue-c [40]), or in part (like Migratory Applications [10]). This workaround would theoretically allow her to use any display server to interact with data. However, display servers may have incompatible hardware or operating systems for executing her software. In addition, the display server owner may not want anonymous users arbitrarily installing software on his display server.

The user's custom software could be compiled to an intermediate language to become hardware-independent and possibly OS-independent. This is how Flash [1], Java [22], and Silverlight [36] operate. But, many applications are simply too large to dynamically transmit to the display server for execution. By comparing installation folder sizes with data file (not video) sizes for most applications, one can see that typical code consumes significantly more hard drive space than individual data files. Transmitting all that application code just to view and edit some data requires a heavy bandwidth load. The excessive startup time for many Flash and Java applets demonstrates this deficiency, which is why Flash, Java, and Silverlight applets are typically designed as small, quickly-downloadable applications or are distributed via physical media or long downloads.

Distributing code also introduces major security risks. The display server is given both user data and the code to process it. A malicious display server could easily infect, damage, or propagate that data. In addition, if the platform-independent code is not properly sandboxed, then software transmitted from a client machine could infect the display server.

Alternatively, an application could send only UI-related code to the display server, leaving the core of the application on the personal device. NeWS [23] sends such PostScript [23] commands to the display server. For NeWS to effectively reduce bandwidth and computing on the personal device, the display server must be dynamically programmed by the application. This dynamic programming reintroduces the distributed code problems that enable the display server to control the UI and the user's data. In addition, importing foreign code exposes display servers to potential infection by malicious clients.

Shipping code, data, or credentials to distrusted devices, and then accepting data back, does not offer safe interaction. In addition, such approaches are limited by network speeds because of the volume of data transmitted, so interaction may not be quick.

2.3.3 Distributing Graphics

Instead of distributing code, data, or credentials and receiving processed data, applications could distribute graphics to the display server and directly receive user input. Then, user data and credentials can be safely retained on the personal device. Two existing methods for distributing UIs to display servers include *rendering-based protocols* and *web-based protocols*.

2.3.3.1 Rendering-Based Protocols

In a rendering-based protocol, personal devices send either raw pixels or rendering commands to the display server. Systems such as X-Windows (X11) [57], PostScript [23], Virtual Network Computing (VNC) [56], and Remote Desktop Protocol (RDP) [68] utilize rendering-based presentation. X11, VNC, and RDP execute an application on one computer and render it on another. Remote rendering is feasible as long as compatible, standard server software is installed on the display server. X11, PostScript, and VNC provide useful architecture examples

for nomadic computing and have been stable standards for some time. These protocols are stable because each is designed around a simple graphics model.

The core issue with existing graphics distribution technologies is that they assume the machine running applications has plentiful resources and that the display server has limited resources—only enough to show pixels and do simple processing. The situation is actually the reverse for nomadic users—the personal device has limited resources and the display server has much greater resources. These technologies all assume a single user interacting at the display server but nomadic computing frequently involves multiple users interacting simultaneously. These assumptions generate challenges for nomadic computing in four areas: bandwidth, CPU power, trust, and multi-user support.

X-Windows and RDP convert drawing calls into network messages that are sent to the display server and rendered as pixels. However, as this article shows in section 2.5.4, X11 and RDP can generate a high volume of network traffic. A simple scroll operation, for example, causes the draw commands for an entire window to be resent. This function not only incurs bandwidth and radio power costs, but also imposes a large computational burden on the personal device, which executes draw commands to rerender the presentation for each scroll movement.

On Windows Vista, RDP allows users to connect to Network Projectors [35], which are RDP-specific display servers. Users annex Network Projectors which then becomes an extension of the user's desktop, where she can use her mouse to drag windows to the projector. However, the projector can only support a single user, and RDP imposes a rendering load similar to X11 on the personal device.

2.3.3.2 Pixel-Based Protocols

VNC copies pixels from the personal device to the display server. VNC's pixel-based model is a highly stable protocol because a purely pixel data structure does not change. But VNC requires much more bandwidth than X11 in many cases (as is shown in section 2.5.4), and the personal device incurs the rendering costs. Mechanisms in the VNC protocol mitigate net traffic but impose additional computational burdens on the personal device. Pixel-based solutions like VNC also react poorly to the varying screen sizes and resolutions encountered in nomadic computing. If the user annexes a large screen via a smartphone, then copying the pixels from a smartphone's screen to the annexed screen is not satisfying. Copying the window from an off-screen buffer can alleviate this issue, but can consume significant RAM and processing resources on the phone, especially when rendering to large display spaces. The application shown on the shared screen needs to render differently to take advantage of the added screen space.

X11, VNC, and RDP always accept input from the display server, rendering the personal device vulnerable to malicious exploitation. X11 and VNC are known for introducing security holes that must be carefully protected. One such protection mechanism that was developed to address such vulnerabilities is the "xhosts" command which limits the machines that may initiate connections to an X11 display server. To increase interactive security, the devices providing application input must be restricted.

X11, VNC, and RDP do not support multiple users simultaneously interacting on the display space: neither of the situations illustrated in Figure 2:5 are possible. In particular, the desired system must support multiple people interacting with the display server, with each user supplying output and input from their personal devices. If one of these technologies were chosen,

only one user could interact with applications shown on a display server at any one time, there may only be only one mouse cursor on the display, and all conflicts must be handled socially.

Some technologies use RDP or a custom RDP-Like protocol, such as WebEx [15], GoToMyPC, GoToMeeting [16] and MaxiVista [6]. These technologies may add some performance enhancements to RDP, but their annexing model is identical to RDP and is insecure because credentials are entered into foreign machines and the user's PC is under the complete control of the foreign machine.

There are a number of other technologies that use VNC, some variation thereof, or a custom protocol that transmits frame buffers, to implement their interactive experiences, including IMPROMPTU [11]—this implementation detail was discovered via discussion with IMPROMPTU's authors—LivOlay [29], WinCuts [65], Lacome [31], and Reflect [5]. These technologies use VNC because it is a simple, stable, cross-platform protocol, with source code available for any platform. Unfortunately, because VNC is a passive process separated from the applications it captures, it limits a window's size and an application's expressiveness. Because VNC renders on the client machine before transmission, the size of a window and the amount of information shown is limited to the size of the screen buffer. Annexing a large display server provides no real benefit to users with small screens on their personal devices. Because VNC is passive, applications cannot have two windows shown on two separate screens. For example, if a spreadsheet application running on a handheld device could participate more fully in the UI transmission process, then the spreadsheet could show a UI on the handheld which is tuned for that handheld's screen while simultaneously showing a much larger, fuller version of the spreadsheet on an annexed screen. This problem surfaces in reverse with Sharp et al. [59]: only a portion of the shared screen can be seen on the personal device, creating a limited experience.

Oprea et al. [47] has all the same problems discussed relative to using VNC. However, it does lead toward a better nomadic interaction model. In this case, some computing and input is provided by the personal device. The personal device brokers a connection between a remote computer and the display server, as illustrated in Figure 2:6, and then provides text and pointing input to the remote computer. The personal device provides a soft keyboard and is enhanced with an optical mouse to provide pointing input. The user retains control over input to his applications. XICE incorporates the input separation and application control models.

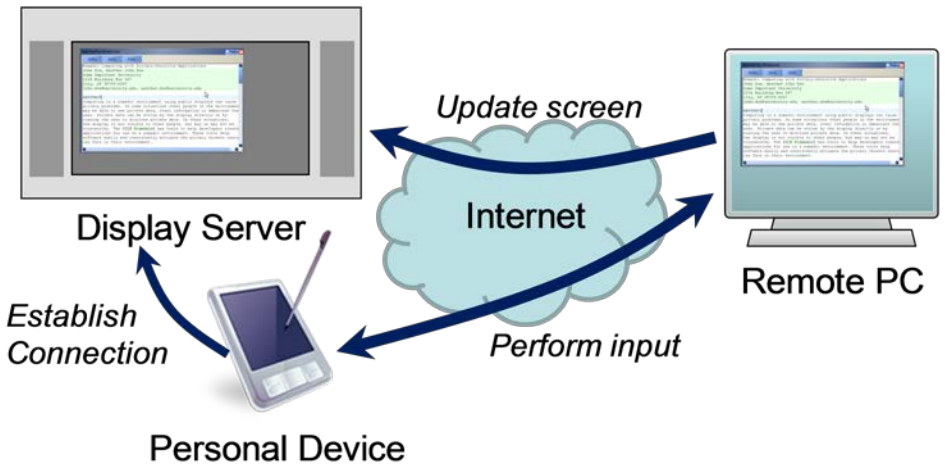


Figure 2:6 – Oprea et al. annexing configuration. The personal device brokers the connection between the remote PC and the display server. The personal device provides pointing input.

2.3.3.3 Web-Based Protocols

In a web-based protocol, Hypertext Transfer Protocol (HTTP) and Hypertext Mark-up Language (HTML) are used for application transmission and presentation. A web server sends an HTML version of application output to a web browser. The web browser then renders the HTML to the screen. HTML is an attractive option, because web technologies are so well-known and prevalent. However, web technologies have several drawbacks, including the following five. First, not all web browsers render identically. Second, HTML versions 1 through 4 do not have

general drawing capacity while HTML 5 requires JavaScript [20] to support generalized drawing. Regardless of support for generalized drawing, HTML does not handle varying display sizes, resolutions, and viewing distances. Without generalized drawing and support for varying display setups, many interactive techniques become difficult to implement. Third, JavaScript implementations are not consistent between browsers. Fourth, JavaScript requires a programmable display and reintroduces many of the security problems related to distributing code (see section 2.3.2). AJAX is becoming more popular as a development technology, but it relies on JavaScript and frequently distributes user data to the browser for manipulation. Fifth, web technologies are difficult to program, because they split code across languages to distribute the UI between the display server and the web server.

Despite the five problems introduced by using HTML, many researchers are exploring how HTML would be used in nomadic environments. Each of these alternatives creates additional obstacles.

There are two basic approaches to using HTML as a network UI distribution technique: the web server and personal server models. The web server model is a familiar paradigm of just logging in to websites via a display server. The personal server model treats the personal device as a web server, so all applications on that device expose their UI as HTML which may be transmitted to a display server for rendering.

2.3.3.3.1 Web Server HTML Model

A frequently suggested option would be to simply use websites for all processing instead of carrying a personal device. To manage personal data, the user would login to her favorite websites at a display server. In this situation, the web browser would play the role of a display server and the websites would be analogous to the personal device. Relying exclusively on the

web is colloquially referred to as “living in the cloud”. The current interaction model for living in the cloud requires display servers to have an acceptable web browser installed. However, just using a web browser for personal computing on display servers introduces several problems. The user must use a browser that is not configured per her preferences (e.g. the display server may have plug-ins or JavaScript disabled, reducing the user’s preferred experience), and she must remember all her favorite sites (which she usually just bookmarks) and usernames and passwords (which are usually stored by the browser). Most importantly, she would have to provide her credentials and interactive input to a foreign device, rendering her vulnerable to identity theft [58]. Display servers would be required to supply input devices which they may not have or which may not be convenient for the environment.

One approach to dealing with security is Mobile Composition [58]. As illustrated in Figure 2:7, the personal device annexes a display server to view specially-crafted web pages. The web pages may contain special encrypted sections which only the personal device may decrypt and view. This approach helps protect the user’s data but requires significant changes to all websites which process sensitive information. Such an approach requires significant changes to large portions of the Internet and is likely unworkable.

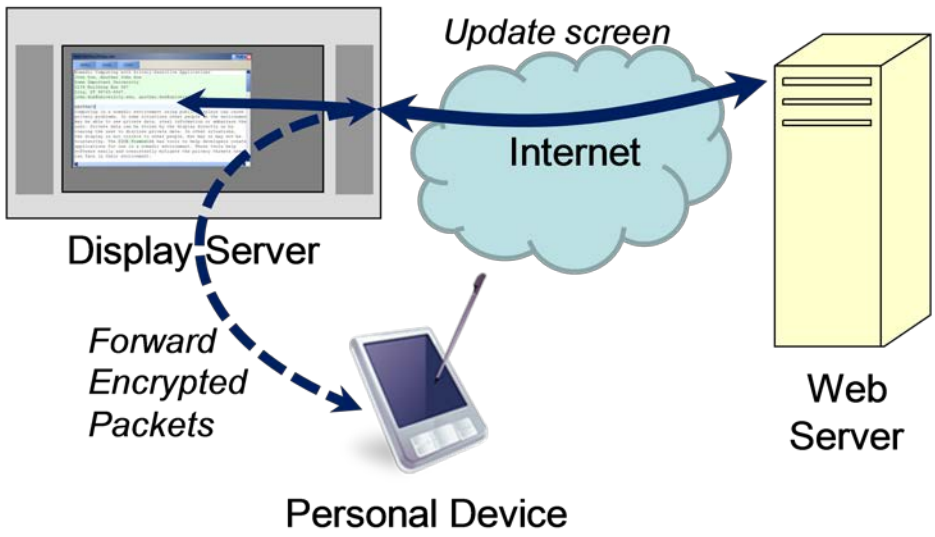


Figure 2:7 – Mobile Composition web-based display annexation. The personal device provides temporary credentials the display server uses to connect to a web server.

It is clearly not suitable to rely on display servers having a web browser, the correct plug-ins, being virus-free, and safe for sensitive input. XICE can provide a more effective interaction model for living in the cloud by merging the browser with a network UI distribution framework. If the user's web browser is implemented in XICE, then she can always use her favorite browser wherever she is, with her plug-ins, even if the browser or plug-in is not installed on the display server. This user always has her browser settings available and can safely and securely enter input (particularly credentials) via her portable device even if she is browsing at a distrusted, public display.

2.3.3.3.2 Personal Server HTML Model

Alternatively, the personal device could act as the web server and produce a UI in HTML for an annexed display server's web browser. Handheld devices can easily host a web server, as demonstrated by the Personal Server [69], which is illustrated in Figure 2:8. The handheld Personal Server wirelessly discovers display servers which are automatically annexed. The

Personal Server transmits web pages to the display server, and provides a jog wheel for scrolling through links on the page, and two buttons for navigating via selected links. However, the personal-device-as-web-server design still suffers from the HTML-induced deficits including insufficiently rich interaction and several other issues.

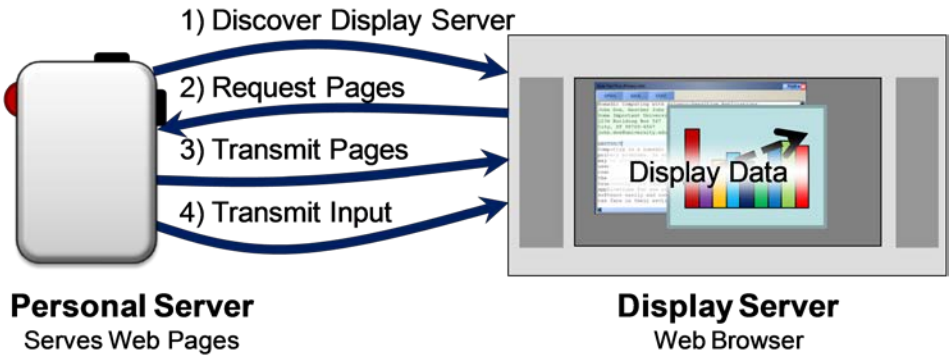


Figure 2:8 – Personal Server connections. The hand-held personal device supplies web pages to a discovered display server. It also provides simple scrolling and selection input while the display server provides no input.

Furthermore, for nomadic computing to use HTTP, connections must be established in a direction that is counter to HTTP's design: currently, a device rendering HTML may request that HTML from a server, but the web server (i.e., the personal device) cannot forcibly push that HTML to a device for rendering. Consequently, the display server must initiate the connection to the personal device to request HTML. As a result, either the connection happens automatically or manually. If it happens automatically, as the Personal Server implements, then the user is exposed to significant security risks because the choice to share information is no longer under the user's control: it is under the display server's control. On the other hand, if the user must establish the connection manually, then she must physically interact with the display server to supply it with the personal device's IP address or URL. This second option creates confusion for the user because her device's IP address is different at each location because that portable device is physically mobile and is likely on a different network with a different IP address.

In addition, the Personal Server automatically trusts input from the display server, so the display server could select any pages from the personal device to be shown or input any data to the personal device. Website developers must proactively work to protect their applications from malicious input [25], creating a heavy developer burden. What is preferred is a system that more proactively protects software from malicious input.

An interesting approach is SessionMagnifier [71] which is a combination of Mobile Composition [58] and Personal Server: the personal device serves up web pages that have been sanitized for the display server. The sanitization process ensures that input happens on the personal device, and that the display server only receives URLs that point to the personal device: no cookies, no URLs to the pages the user is browsing to, etc. Even though this approach still suffers from the five problems of HTML (especially because SessionMagnifier requires JavaScript to be enabled), it does approach data sensitivity in an important way: provide applications with a way to filter out sensitive data. XICE incorporates this sanitation in its privacy implementation: an application can produce a UI using widgets that automatically sanitize the output for a distrusted display server.

Although HTML needs some changes to its standards to reduce the five problems, there are three key observations about the web and nomadic interaction that encourage a new, simple rendering standard: distance, styling, and layout. First, the web is designed for large distances—both in physical distance and in the number of network hops—between the server and the browser and high latency. But, nomadic interaction has short distances—physical and hops—and low latency. A large reason to have JavaScript or Applets is to prevent round-trips to the display server, thus reducing latency and increasing interaction time. With short distances and low latency, preventing round-trips is no longer necessary. Second, CSS is designed to add styling to

web pages so that people in different situations can view the same content differently. However, with applications that show on a single display space, such styling is not necessary. Third, the layout problem addressed by CSS can be handled by the "server" (personal device) instead of the "browser" (display server). With a sufficiently consistent rendering framework and low network latency, users can have a richer interaction so these techniques for dealing with high latency are no longer necessary and complicate the programming model.

2.4 The XICE Protocol

No existing UI distribution technology sufficiently meets nomadic computing needs. Each system that has some promise is missing several other key pieces. With all of these issues, the authors decided to build a new framework from scratch. For the framework to be successful, applications must be re-implemented, but many systems require applications to be rewritten, such as the iPhone [2] and Android [21] platforms.

A new architecture is needed that puts the personal device in control. A user needs to carry "his world" with him: his data, his applications, his settings, all in his own way. This architecture needs to expand applications from the tiny screen space on the personal device to the large space afforded by the screens he encounters. This expansion needs to take place in the context of being efficient on computational and network burdens. The framework must also allow multiple people to interact simultaneously in the same display space. The framework must also address the issues of trust as users more-frequently interact with foreign devices. For example, the iPhone addresses trust by locking up everything: each application lives in its own isolated partition. However, this is a limiting approach to trust so a broader mechanism is needed.

XICE was developed specifically to minimize the overhead of network traffic and CPU usage while distributing the UI so that smaller devices may be better supported in nomadic experiences. The theory raised and confirmed in this paper is that a scene-graph (described in the section 2.5) drastically reduces CPU and network loads, which opens up avenues for smaller devices to present large interfaces on annexed display servers.

XICE is intended as an experimental framework to test the integration of several existing technologies with some newer technologies, as well as to design new interaction techniques. One such new technique is Handheld Spilling [44] where a UI from a handheld device is shown on both a table-top screen and a handheld. As shown in Figure 2:9, the UI is synchronized between the handheld and tabletop so that the portion shown on the handheld aligns with the window shown on the tabletop. The user interacts directly with data via the handheld's screen and navigates by scrolling the UI via the table-top (often with her nondominant hand). The handheld presents a spatially limited interaction area, but the tabletop may be an untrustworthy device. Spilling is beneficial because the user can see more information via the tabletop while the user's software is protected from potentially malicious input by interpreting all incoming input as panning motions rather than input that could alter user's data (potentially injecting malicious data). Another new technique is the MousePuter, which is described in section 2.6.2.1.2.

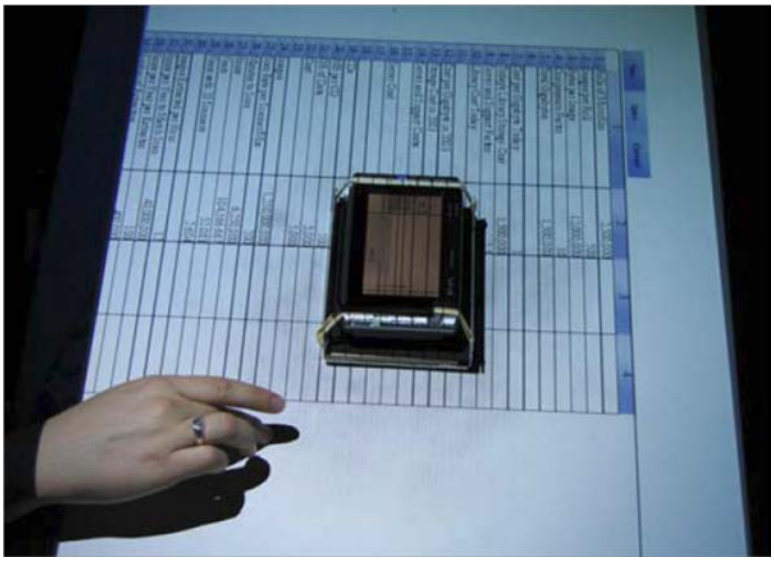


Figure 2:9 – Handheld Spilling: a UI is shown both on a handheld and tabletop computer. The user may interact directly with data through the handheld, and may scroll via the tabletop.

There are two large facets to XICE's implementation: the underlying protocol that communicates between a personal device and a display server, and the windowing toolkit that performs window management and supplies common UI components such as window decorations, buttons, labels, text boxes, and system-supplied dialogs. Although several examples and figures within this article discuss aspects of the windowing toolkit, this paper focuses on the underlying protocol. The windowing toolkit is not as important as the protocol: the protocol has stayed fairly stable across experiments while the look and feel of the toolkit has changed, sometimes drastically. Consequently, this article evaluates the protocol, not the look and feel.

Concentrating on the protocol elucidates the goal of integrating a wide variety of devices into the nomadic experience. Expecting all devices to conform to a new windowing toolkit is unreasonable: each device's toolkit has its own strengths that its users prefer. But, expecting a large subset of devices to be able to support a simple protocol is reasonable, especially because so many devices support networking protocols such as HTTP and HTML.

The XICE protocol has two major pieces which provide a rich and safe nomadic experience: the graphics engine and the input framework. The graphics engine provides a large decrease in network usage when distributing a UI. The input framework adds safety via *input redirection*: the personal device can supply all input to windows the user shares on annexed displays instead of accepting input from the annexed display's input hardware. A detailed discussion of the graphics engine is in section 2.5, and the input framework is discussed in section 2.6.

The XICE protocol is implemented in Java 6. Java was chosen primarily because garbage collected platforms facilitate rapid software development, but Java was also chosen because of its cross-platform abilities. In addition, version 6 brings developer tools such as annotations and generics (from version 5) and tighter integration with a platform's UI-toolkit (e.g., window transparency).

Although the protocol is currently implemented in Java, the protocol is designed to be implemented by other languages on other platforms. The public interface exposed by the UI transmission protocol is platform independent. To demonstrate this, XICE has also been implemented in C# [38].

2.5 Rendering Implementation

The following specific principles guided development of the graphics engine within XICE. First, graphical presentations should be simpler to program than the traditional damage-repaint cycle, reducing development time for an application. Next, the rendering system should be seamless across devices, so developers need not consider whether a window is rendered on the personal device or on an annexed screen. Third, the UI should scale to be easily readable regardless of the display's pixels per inch (PPI) resolution and the user's viewing distance, and

the developer should not need to consider either of these. Finally, the rendering system should handle large screens containing numerous pixels, but only incur a minimal load on the personal device. The scene-graph architecture was selected for its superior ability to meet these criteria.

2.5.1 Automatic Rendering

At the core of XICE is the scene-graph or *presentation tree*. (Those familiar with scene-graphs may skip ahead to section 2.5.2.) The presentation tree is modeled after technologies such as Graphical Kernel System (GKS) [24], Programmer's Hierarchical Interactive Graphics System (PHIGS) [62], Jazz and Piccolo [8], and Windows Presentation Foundation (WPF) [53]. In these technologies, an application builds a graph of draw commands that the toolkit renders onto a display. Scene-graphs are basically display lists structured in graph format. Scene-graphs were first developed with GKS and PHIGS to render 2D and 3D scenes across a network. HTML—especially when represented as a Document Object Model within a browser—is a more popular, but less consistent, scene-graph.

With the scene-graphs in WPF, Jazz, and Piccolo, interactive *widgets*—such as buttons or text boxes—are embedded in the scene-graph. These widgets may contain other widgets as part of the graph, and may supply rendering primitives. So, for these technologies, the scene-graph provides structure for rendering, widget containment, and input dispatching. XICE also follows this approach.

Traditionally widgets are rendered using the damage-repaint cycle. Each widget possesses a widget model, produces visual output, handles input, and raises events. For example, a button widget would have Boolean variables to track its enabled and up states and a string variable for the label. To change its visual output, the widget must inform the windowing toolkit (damage), and after all immediate events that might affect the widget’s visual output have been

processed, the windowing toolkit sends a redraw (repaint) event to the button. Continuing our button example, as a mouse down occurs on the button, the button changes to the down state and then executes the damage-repaint cycle. Next, when a mouse up is received, the button changes back to the up state, raises a “clicked” event, and executes the damage-repaint cycle again.

A significant benefit of scene-graphs is that the toolkit automatically renders pixels from the instructions in the graph, and changes to the graph are automatically rerendered. Figure 2:10 illustrates a scene-graph and the output that is rendered from it. When the scene-graph is altered—in this case by changing the circle’s fill color to yellow—the rendering engine tracks those changes and rerenders the altered portions of the UI.

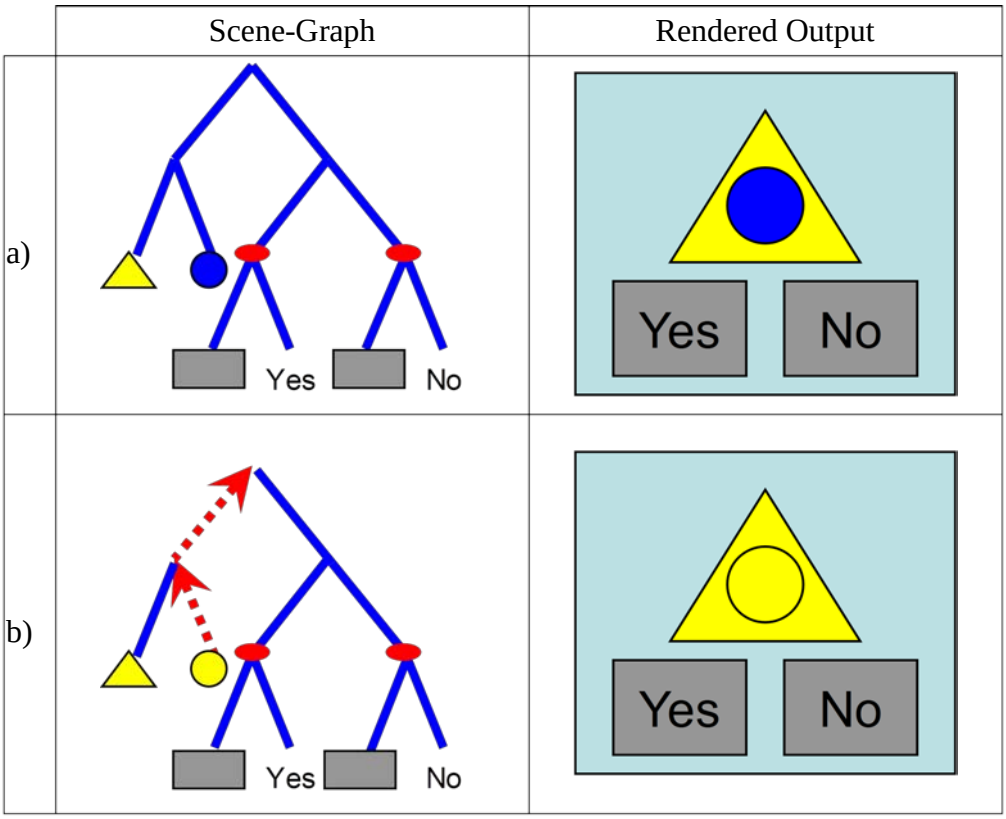


Figure 2:10 – Scene-graph rendering: (a) shows the original graph, while (b) shows how changing the circle’s fill color to yellow causes notifications to travel up the graph and affects the rendered output.

By automatically rerendering the UI's scene-graph changes, code can be minimized, and end developers can be removed from the damage-repaint cycle. Rather than recreating the UI through draw calls each time the paint portion of the cycle happens, developers can create the UI once with the presentation tree, and then update the tree with changes. Piccolo allows developers to perform custom rendering, improving performance in some places; but, custom rendering using Piccolo necessitates the damage-repaint cycle. On the other hand, Jazz removes the damage-repaint cycle, simplifying rendering for developers. XICE does what Jazz does: applications maintain a tree of draw commands instead of calling rendering methods.

The presentation tree represents draw commands—such as draw a line, draw a rectangle, or draw an ellipse—as *nodes* within the tree. Leaf nodes represent drawing primitives, while interior nodes can apply simple effects to those drawing primitives. For instance, the Transformer node applies an affine transform to all child nodes and the Clipper node clips the rendering of its child nodes. Interior nodes can show or hide their children. Nested interior nodes accumulate their effects. For example, embedding a Transformer node with a rotate transform inside a Transformer node with a scale transform causes all leaf nodes within the inner Transformer node to be both scaled and rotated. Embedding a Clipper node inside a Clipper node causes the clipping effects to be accumulated on the child nodes of the inner Clipper node.

The XICE rendering frameworks supports the graphical primitives supplied by the Java Graphics2D object. These primitives include lines, rectangles, ellipses, curves, gradients (linear and radial), images, audio and text. In addition to clipping and transforming, XICE supports transparency. XICE does not support video, but that is because the Java Media Framework (JMF) [49] does not support video well. If the rendering engine were implemented in WPF, it would easily support video.

Transforming and clipping support not only scrolling, but also rotated windows in DiamondSpin [60] and skewed windows in Metisse [14]. XICE does not support the extended zooming architecture of Pad [52], Jazz, or Piccolo.

2.5.2 Seamless UI Distribution

Seamless UI propagation to an annexed display server is critical to a simple nomadic architecture. Scene-graphs within XICE are seamlessly propagated over a network without involving the application. When the presentation tree is rendered on an annexed display the entire tree is serialized to the display server, and the display server maintains a copy of that tree. As the application changes nodes within the presentation tree on the personal device, the altered portions of the tree are serialized to the display server, and the display server's copy of the tree is updated.

To accomplish this seamless UI distribution, the serialization protocol has four key aspects: primitives, messages, graph nodes, and widgets.

2.5.2.1 Serialization Primitives

XICE uses a standard TCP/IP connection to transmit data between a client and a display server. The protocol for serializing and transmitting that data is simple and easily implemented in other languages and frameworks. A display server was built in C# and easily processes such data.

Within XICE there are seven different serializable primitives: integer, double, byte-array, string, start, end, and message. Integer, double, byte-array, and string use standard high-byte first or UTF encodings to transmit the data. Start, end, and message are XICE-specific identifiers.

Before transmitting any primitive, a single-character prefix is transmitted as a 2-byte encoding. Then the data follows that prefix. In the case of the byte-array, the length of the byte

array follows the prefix as an integer before the contents of the byte array are transmitted. In the Java implementation, all values are transmitted using functions on the DataOutputStream [48] object. A C# decoder was a straightforward, simple implementation.

2.5.2.2 XICE Messages

The primitives are available so that messages may be easily transmitted between machines. When writing out a message, XICE first writes out the identifier for the “message” primitive. Then, the message contents follow as a sequence of primitives.

XICE messages are predefined for the framework and are not expected to be created by end developers. Message objects are designed with both a serialize and a deserialize method, and the framework developers kept the two methods coordinated such that each primitive written has a matching read.

2.5.2.3 Graph Nodes

Most messages have a fixed number of properties, so coordinating the serialize and deserialize methods is a matter of reading in data in the exact same sequence and type it was written out in. However, graph nodes may contain an arbitrary number of properties of arbitrary types and an arbitrary number of children.

The properties and children are written out as two separate groups. The serialization mechanism uses the start and end primitives to track the beginning and end of the groups of properties and the groups of children, similarly to how Java, C, and C# developers use curly braces to delimit a group of code.

The scene-graph nodes are transmitted at specific times relative to user interaction. The entire scene-graph is transmitted the first time that presentation tree must be rendered by a

display server. After that initial transmission, changes to the scene-graph are transmitted in batches.

To transmit changes in batches XICE integrates with the application's event dispatching loop. Every time the event queue is emptied (i.e., all the pending events are processed by the application) XICE inspects each scene-graph for changes. If a scene-graph has changed then those changes are collated into a single message which is transmitted to the display server.

After the changes are collated and transmitted, XICE clears the presentation tree of all change tracking flags.

2.5.2.4 Serializing Widgets

In XICE, as with Piccolo, interactive widgets are implemented as subclasses of interior nodes in the presentation tree. Embedding widgets within the scene-graph merges the geometry of presentation rendering with the geometry of input handling.

XICE offers nothing novel with regard to the basics of the serialization mechanism, so a description of that process is omitted from this paper. However, because widgets are embedded within the tree the widget serialization process needs to be detailed.

Standard serialization techniques (e.g., Java and .NET [34] serialization) require the same class structure to exist on both the sending and receiving ends of serialization. In addition, all private fields of a class are serialized. But, the private fields of interactive widgets can contain sensitive data (e.g., email addresses, usernames, or the password a user enters into the password box) that should not be sent to a display server. The display server renders widget output, but does not execute widget code, so serializing the entire widget is inefficient. Developers may also create novel widgets unknown to the display server, which necessitates shipping novel class structures to the display server before serializing the widget. To minimize security risks and to

simplify the architecture, display server implementations should not adapt to class structures of new applications.

The XICE serialization protocol minimizes risk by transmitting *display serialized nodes*: a safe subset of presentation node classes that both the personal device and display server understand. When serializing a widget, its type hierarchy is searched until a display serialized node is found. Then, the properties from that node are serialized to the display server. Consequently, a display server can support the graphical output of any widget without receiving its class definition, and the widget's private data will not be automatically transmitted to the display server.

Consider the simplified type hierarchy of the Button widget shown in Figure 2:11. The gray boxes represent types within that hierarchy that are display serialized nodes. When the serialization mechanism encounters the Button in a scene-graph, it first looks to see if Button is a display serialized node. Because Button is not display serialized, the parent type, Controller, is inspected. It also is not display serialized. However, ResizableContainer is, so the display server is informed of the presence of a ResizableContainer, and all the properties from the ResizableContainer type are serialized to the server.

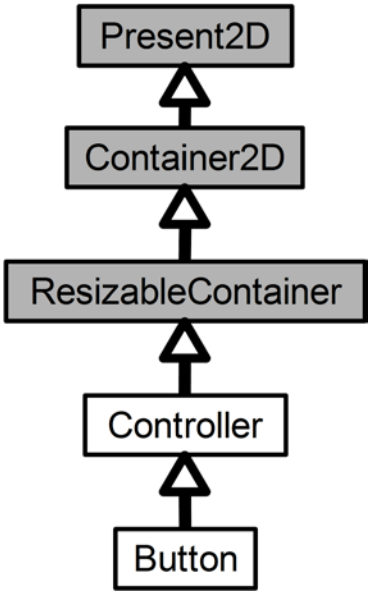


Figure 2:11 – Type hierarchy of the XICE-supplied Button class. The gray boxes are the widgets types that a client and server are both aware of.

Notice that even though the XICE windowing toolkit provides the Button type, the ResizableContainer is the type actually transmitted to the display server. The Button type also performs no rendering on its own: it has child nodes which represents all the graphical elements that represent the output of that button. After serializing the ResizableContainer type, all the child nodes from the Button are also serialized. Consequently, the Button renders identically on the display server. From the display server's perspective the Button appears simply as a ResizableContainer (which is a container with Bounds) with several rendering primitive nodes as children, but none of the button functionality. Not transmitting the Button type is an intentional design decision which keeps the XICE protocol simple, and allows the protocol to be implemented in other languages and on other platforms.

2.5.3 View-Independent Coordinates

Nomadic users will encounter different display and viewing situations. Regardless of the situation, information the user shows on an annexed display must adapt to the display's size, resolution, and position, and produce a readable, interactive interface.

XICE solves the screen adaptation problem by having all applications use view-independent coordinates (VIC). Pixels per inch (PPI) have no real perceptual meaning. Instead of PPI, visual perception is defined in degrees of visual arc (DVA). For projectors, PPI, zoom lens, and viewing distance must all be taken into account.

Employment of VIC is not merely an adaptation to the resolution of a display. VIC is determined by what the user sees rather than just what the hardware can show. 10 VIC is defined as the size of normal, comfortably read text. When viewed from 24 inches, a 10-point font (which is 0.1 inches high) is comfortable to read for most people. This font viewed at this distance is approximately 0.3 DVA. So, 10 VIC equals 0.3 DVA. This model is simple for programmers to understand because 10 VIC is conceptually similar to 10-point font. Originally, distances in XICE were defined using DVA, but programmers were confused about how large 1 DVA would be, so they guessed. The VIC system is much easier to explain and to use effectively.

People annexing the display server do not need to configure it. Every XICE display server has the VIC configuration program shown in Figure 2:12.

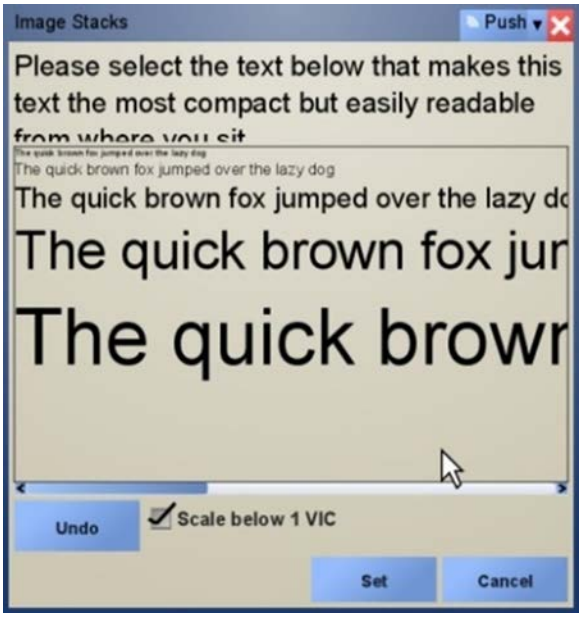


Figure 2:12 – VIC configuration program. A display server's owner uses this program to configure the display server for typical viewing by iteratively selecting the ideal text size for viewers.

An owner runs the VIC configuration program once to configure the display server for use. This program shows text of various sizes on the display and asks the owner to pick the smallest text he can comfortably read from a typical viewing distance. As the owner chooses text, a different set of text with sizes near the selected text are shown. The owner iteratively picks text until he likes the selected size. The program then calculates the appropriate VIC for the display based on the chosen text.

Instead of picking a comfortable font size using the VIC configuration program, the display server owner could rely on the VIC measurement program to calculate a reasonable estimate. The application displays a 200-pixel by 200-pixel square. The owner is asked to measure and enter the width W and the height H of the square on the screen, and the user's typical viewing distance D. The units of measurement are unimportant as long as the same units are used for all three values. In places like conference halls, viewers are located at different viewing distances, so a judgment call must be made regarding the appropriate view distance for

common uses of the display. Given *W*, *H*, and *D*, and the fact that 10 VIC is 0.3 DVA, the program can compute scale factors for X and Y.

$$S_x = \frac{0.3}{10} \times \tan(1^\circ) \times D \times \frac{200}{W}$$

$$S_y = \frac{0.3}{10} \times \tan(1^\circ) \times D \times \frac{200}{H}$$

Two separate scale factors are used, because displays do not always use square pixels.

When a user connects to a display server, his personal device is informed of the display dimensions in VIC. The toolkit software on the personal device then knows the bounds of the screen. Applications can use this toolkit information to adapt themselves to the available display space. XICE does not address the UI adaptation problem [41]; XICE merely informs the application of what the visual parameters are. The application can then adapt using any algorithms or techniques developers may choose.

In some situations, the viewing distance could depend on the distance the user is connecting from. For instance, a user at one end of the room may want to interact directly with the screen, while another user may want to interact with his data from a distance. The two viewing distances could be configured dynamically. Tools that can measure distance such as the Wii [42] or the Kinect [32] could easily measure user distance and configure the VIC automatically. How to best approach this is left as an area for future research.

2.5.4 CPU/Network Load Evaluation

A major issue with distributing the UI from a personal device to a display server is the battery drain. Battery drain comes primarily from the backlight, CPU, and the radio. An interactive architecture can affect the CPU load and the radio, but not the backlight. The process

of generating and transmitting a UI to a different machine influences the CPU and radio usage. Although a user could potentially plug their device into a wall outlet (or other power source) to gain the requisite processing power, such an approach then tethers the user to a limited range of motion. An ideal approach would be to operate for longer periods of time exclusively on battery power. A rendering engine that minimizes both CPU and radio usage will help extend battery life when annexing a display server. This paper contends that the average interactive application spends the majority of its CPU cycles rendering pixels and that using a scene-graph can greatly reduce processing time and network usage by offloading pixel rendering to another device, especially when compared to other network UI distribution technologies such as X11, RDP, and VNC.

Consider a word processor displayed in a 600- by 400-pixel window. Typing the letter “y” will cost at most 10,000 instructions to make room in the document and insert the letter “y”. If the “y” is placed in the middle of the document, half of the pixels in the window may need to be repainted for word wrapping and new lines. Suppose each pixel takes a minimum of 12 instructions to render for a total of 1.44 million instructions to repaint half the window. Less than 1% of the CPU cost is in actual document modification.

Now consider a 10-column by 30-row spreadsheet displayed in the same 600x400 window. If a change in one cell causes all other cells to be recalculated at a cost of 100 instructions per cell, a total of 30,000 instructions must be processed for that one cell’s value change. With 2.88 million instructions necessary to repaint the entire window, the cell recalculations amount to approximately 1% of the cost of repainting the window.

These two simple examples show that in modern graphical applications, the CPU cost of updating the application’s model is far overshadowed by the cost of painting pixels. In the

examples, only 600x400 windows were used. If the personal device distributes these windows' pixels via VNC or VGA, rendering on the personal device may be reasonable. However, with wall-sized displays, the personal device may manage several million pixels, greatly increasing the personal device's effort to calculate pixel values.

XICE addresses the multi-million-pixel rendering problem by imposing only presentation tree manipulations on an application rather than constantly rendering UIs on the personal device. Experiments were conducted to validate the advantage of XICE's distributed scene-graph architecture. The tests compare the UI distribution techniques of TightVNC [66] (Tight), RealVNC [55] (Real), RDP, and XICE on Windows XP, and X11, the system-supplied VNC, and XICE on Linux. With each UI distribution technique, the tool was installed and used as-is. In the case of TightVNC, it was used with the "high-bandwidth" option selected.

XICE, X11, and RDP each render only on the display server, while all versions of VNC render on both the client machine and the display server. One may wonder why XICE should be compared to VNC because the fundamental difference in rendering styles. VNC was chosen because the render-locally-and-transmit-frame-buffers approach is used by several potentially nomadic environments such as IMPROMPTU [11], LivOlay [29], WinCuts [65], Lacome [31], and Reflect [5].

For all experiments, the personal device and display server are each a 3.4-GHz, hyper-threaded Intel Pentium 4 processor with 1 GB of RAM. The network is a single gigabit Ethernet switch. For all windows experiments, Windows XP SP3 is used, with all updates current to December 15, 2010. For all Linux experiments, Ubuntu 10.10 [13] is the operating system used, with all updates similarly current. In both cases, the Sun-supplied JRE is used: the OpenJDK [50] supplied with Ubuntu did not perform as well when rendering. Data was collected using the

logman [33] tool in Windows and nmon [27] in Ubuntu. In all tests, the application that produces the UI is a XICE application. The application is started, after a few seconds the monitoring tool is started and allowed to run for 4 minutes, collecting samples once every second. Then the first 15 seconds and last 15 seconds are discarded and the averages collated.

In the first experiment—the *small rotation task*—a spreadsheet is displayed in a small 50x50 window. Every 100 milliseconds, a separate thread rotates that window back and forth by 1.2 degrees, testing transformation with few overall drawing updates. Results are compared in Figure 2:13.

Win	**None**	**Tight**	**Real**	**RPD**	**XICE**
Client	0.44%	2.02%	12.91%	0.64%	0.42%
Display		0.57%	0.48%	1.95%	0.47%
Bytes/s		1611	5044	1106	3128
Pack/s		3.74	6.35	14.21	14.2
Linux	**None**		**VNC**	**X11**	**XICE**
Client	3.94%		4.97%	6.86%	0.29%
Display			0.40%	3.30%	4.28%
Bytes/s			18408	2470160	3895
Pack/s			87.61	3494	20.13

Figure 2:13 – Performance on a small rotation task.

The None column shows the percent of total processing power the personal device uses to execute without distributing the UI. The client row shows the percent CPU usage on the client device while the Display row shows the percent CPU usage on the display server. The values in the Display row are not consequential, because the display server can be expected to have the resources necessary to process rendering commands. Processor usage includes the application, the UI distribution technology, and the operating system. The Bytes/s row shows the number of bytes transmitted per second on the LAN while Pack/s shows the number of packets transmitted per second. Network usage is important, because radio transmissions can consume considerable power. Decreasing network traffic helps minimize radio utilization.

The most salient result in Figure 2:13 is that the CPU usage drops for both Windows and Linux, although Linux drops 3.94% to 0.29% when using XICE. That decrease translates to a 92.6% drop in processor usage on the personal device and includes the time required to serialize the presentation tree. By contrast, all other distribution techniques increased processor usage, and in the case of RealVNC, by 29 fold. And the network usage—of secondary import for preserving battery life—shows that XICE performs worse in half of the cases. However, it performs much better than VNC or X11 on Linux, and performs a little better than RealVNC on Windows.

In the second experiment—the *large rotation task*—the same rotations are performed with a 600x400 pixel window. Nearly 100 times as many pixels are rendered. The results of this experiment are shown in Figure 2:14. On the personal device, XICE drops CPU usage from about 50% to less than 0.5% (a 99% drop), while RDP does not drop usage at all. In terms of network usage on this second task, XICE greatly outperforms the other options in both Bytes per second and packets per second. In fact, network usage stays roughly the same as the small rotate task.

Win	**None**	**Tight**	**Real**	**RPD**	**XICE**
Client	50.09%	51.72%	50.03%	49.89%	0.44%
Display		2.68%	0.51%	5.46%	49.34%
Byte/s		96933	27240	589899	3111
Packet/s		158	40.5	686	14.1
Linux	**None**		**VNC**	**X11**	**XICE**
Client	48.54%		71.67%	16.98%	0.29%
Display			12.12%	14.51%	48.48%
Byte/s			1229522	28653027	3895
Packet/s			2870	27926	20.14

Figure 2:14 – Performance on a large rotation task.

For the final experiment—the *scrolling task*—a separate thread scrolls a spreadsheet vertically one movement every 100 milliseconds. The spreadsheet is presented in a 600x400 pixel window. While the prior two tasks are designed to force the UI to completely repaint, this

last task more closely matches the common user technique of scrolling a window. The act of scrolling within the application causes many pixels to be rerendered without changing all pixels.

The results of the scrolling task experiment are shown in Figure 2:15. XICE drops processor usage from 25% to 0.44% (a 98% drop), while X11 can only reduce usage from 36% to 17% (a 53% drop). All other distribution techniques increase the client's CPU usage. In terms of network usage, XICE significantly outperforms the other options.

Win	**None**	**Tight**	**Real**	**RPD**	**XICE**
Client	24.86%	27.61%	25.76%	30.53%	0.44%
Display		1.04%	0.55%	2.79%	36.19%
Byte/s		11268	9642	200265	12645
Packet/s		22.5	22.2	237	14.3
Linux	**None**		**VNC**	**X11**	**XICE**
Client	35.84%		70.34%	16.96%	0.39%
Display			15.74%	14.52%	43.17%
Byte/s			1269737	30686223	14232
Packet/s			3615	29470	20.1

Figure 2:15 – Performance on a scrolling task.

A major complaint about distributed interaction is the lag imposed by the distribution mechanism. To address this issue, the interactive response of the various distribution techniques is rated according to the following human observation of the interactive behavior of the application:

- Excellent—no noticeable lag: less than 0.1 seconds
- Good—slightly noticeable lag: between 0.1 and 0.25 seconds
- Fair—noticeable lag: between 0.25 seconds and 1 second
- Poor—excessive lag: more than 1 second

The times presented are a rough estimate of the lag between when a change is performed on the client machine and how long it takes before that change is rendered on the display server. Figure 2:16 lists the overall performance of each distribution technique relative to each task. Surprisingly, Ubuntu's version of VNC performed excellently for all three tests. However, that

performance came at a large CPU and network cost. In terms of interactive responsiveness, XICE clearly performed better than any other technology examined.

	Tight	Real	RDP	X11	VNC	XICE
Rotate Small	Fair	Fair	Good	Fair	Excellent	Excellent
Rotate Large	Fair	Fair	Fair	Poor	Excellent	Excellent
Scroll	Fair	Fair	Fair	Poor	Excellent	Excellent

Figure 2:16 – Interactive lag performance.

The implementation of the Java Graphics rendering pipeline on Windows may affect these results. RDP intercepts Windows Graphics Device Interface (GDI) [70] calls and forwards them to the rendering display. However, Java Graphics2D uses DirectDraw [70] and not GDI. DirectDraw is meant for high-performance gaming with rapidly changing UIs, while GDI is meant for applications in which the UI does not change as frequently. Consequently, RDP does not receive notifications of each draw call within DirectDraw. Instead, RDP periodically sends the DirectDraw frame buffer to the display server. For Java applications, RDP pushes frame buffers just like VNC does, so RDPs true performance characteristics may not be reflected in the collected data. RDP would be expected to perform closer to X-Windows' characteristics.

X-Windows provides a more accurate representation of a competitive distributed UI technology, because X11 intercepts Java rendering calls and forwards them to the display server. The comparison between X11 and XICE is a better indicator of XICE's performance against a tightly integrated UI distribution technology. Even X11, however, could only achieve a 65% reduction in processor usage (48% to 17%), compared with XICE's 99% reduction. And, X11 imposes a gigantic network load—one to two thousand times what XICE uses during the same task. Even though the network protocol for XICE is not optimized for highly efficient binary throughput, it still yields a huge performance boost with minimal effort.

In the two larger experiments, using XICE greatly reduces both CPU load and network usage, while the smaller experiment shows that XICE performs a little worse. However, XICE's interactive experience excels for all experiments. Combining both the interactive experience and the observation that XICE's client-side network and CPU loads stay constant for all three distributed tests, shows XICE to clearly be preferable. The scene-graph architecture greatly reduces processing time, and can greatly reduce network time as well. Nomadic computing with XICE's incremental update of scene-graphs extends battery life and helps ensure that personal devices can handle the required computation and rendering loads on large shared displays while providing a responsive user experience.

2.6 Input in Nomadic Situations

Users must be able to provide both text and pointing input to applications regardless of which of the three nomadic situations applies. The applications must be able to operate without considering the input's source or if that source changes. XICE's scene-graph architecture simplifies input dispatching, compared to input handling in the damage-repaint cycle, and allows applications to operate regardless of the nomadic situation.

2.6.1 Input Handling

In each nomadic situation, users provide input to interact with a widget. Traditionally, input is handled by widgets that use the damage-repaint cycle, and presentation geometry is created in the repaint method. To select an object, pointing input is hit-tested against the object's presentation geometry. If a geometric transformation is applied to an object's presentation, the inverse of that transformation must be applied to properly hit-test the object. Hit-testing in traditional architectures frequently requires the widget to reproduce the presentation geometry. Applying a transform in the repaint method, for example, necessitates calculating and inserting

the inverse transform into hit-testing code. Nested geometric transformations further complicate this process.

In scene-graph architectures, the presentation geometry, including affine transforms, is stored in the scene-graph. In architectures like XICE, Piccolo, Jazz, and WPF, input is dispatched top-down through the scene-graph. For example, Figure 2:17 shows pointing input dispatched to the "Yes" button, represented by the left red ellipse in the tree. The button widget performs a hit-test against the button's background rectangle to determine if the input is within the rectangle's bounds.

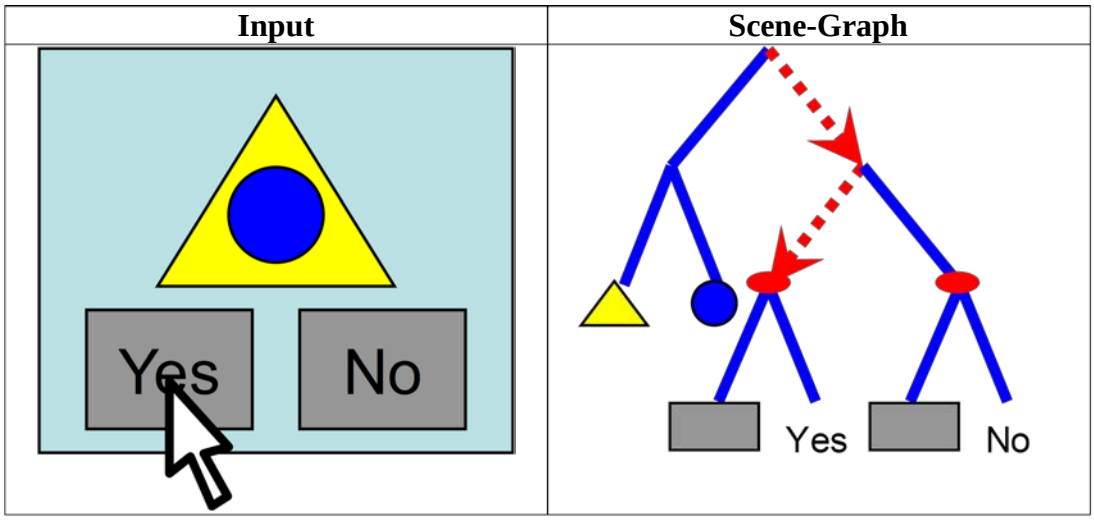


Figure 2:17 – Dispatching input within a scene-graph. The mouse click starts at the tree root and passes through each node down the tree until it arrives at the button widget. The button widget hit-tests the click against the rectangle's bounds.

When using a scene-graph, developers create the tree once, and any transforms that need to be applied are embedded within the tree. As pointing input passes through a Transformer node, that input is transformed according to the transform at that node. Nested transformations result in nested transforms of the input. By transforming the pointing input at each Transformer node, the input is in the proper form for each node's coordinate space, removing input handling complications from the damage-repaint cycle. Developers can hit-test the existing tree instead of

recreating the presentation geometry and remembering to properly apply inverse transforms to input.

Keyboard input is not affected by affine transforms and is therefore dispatched using standard text dispatching mechanisms.

2.6.2 Input In Nomadic Situations

In nomadic situations, different devices can supply user input to local or remote windows. *Local windows* are pieces of software on the personal device that send their UIs to the personal device's screen. *Remote windows* are pieces of software on the personal device that send their UI's to a *proxy window* on an annexed device's screen. In the personal device alone situation (Figure 2:2), local windows receive input from the personal device's input hardware. In the annexed screen and input situation (Figure 2:3), remote windows receive input from annexed devices. And, in the annexed screen only situation (Figure 2:5), remote windows receive input from the personal device. When supplying input, users need to be able to smoothly transition among nomadic situations.

If the user is in the personal device alone situation, no transitioning is necessary, because all interaction occurs on the personal device. If she is in the annexed screen and input situation, remote windows receive input from the annexed input devices. Suddenly, an intimate note from her spouse arrives on her personal device. When the message arrives, she transitions by physically using her personal device instead of the annexed input devices.

The annexed screen only situation, however, requires special consideration for redirecting input. A user in the annexed screen only situation interacts with applications shown on an annexed screen. Her personal device supplies pointing and text input. Pointing input is supplied by translating input on the personal device into mouse clicks and cursor movements reflected on

the annexed screen. Text is entered through a soft keyboard on her personal device. When a potentially embarrassing note arrives from her spouse, the note is shown on the personal display where she can view it discreetly (Figure 2:18). If she ignores the message, personal device input continues to be sent to remote windows. However, if she elects to write a reply, input should be redirected to local windows. She must be able to smoothly transition between local and remote windows while using only the personal device for pointing and text input.

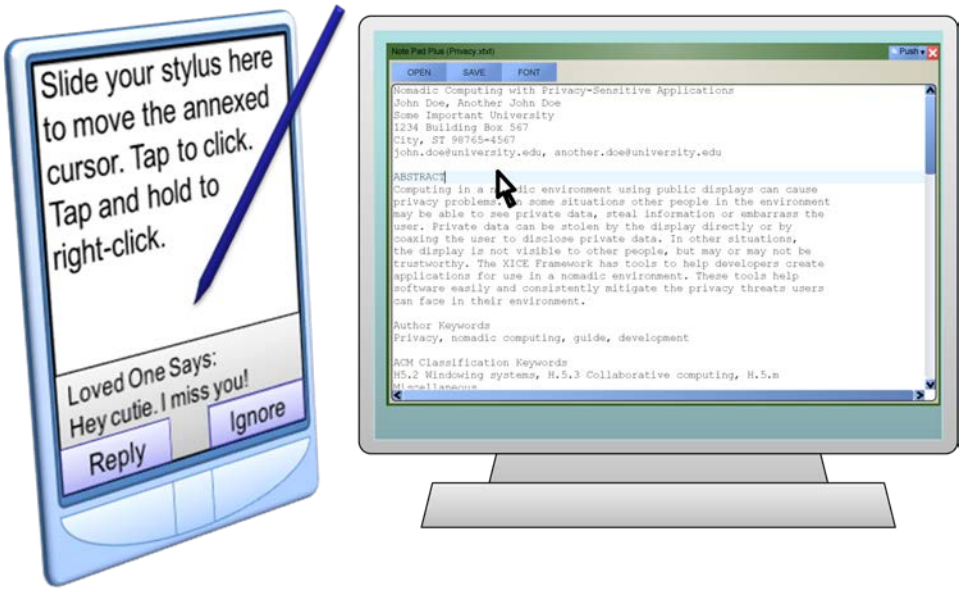


Figure 2:18 – A user may supply input on her personal device for a document in a remote window. If a message arrives on her personal device, she must be able to smoothly transition to supply input for the local window.

2.6.2.1 Pointing Input

The user should not need to look at the personal device to supply pointing input for remote windows. On the personal device, the UI for entering pointing input should not provide mechanisms whereby the user can inadvertently switch from remote to local windows. When she is focused on the display server's UI, looking back at the personal device to ensure she is providing input to the correct windows can annoy her and distract her from accomplishing her tasks.

Existing personal devices are designed to only provide pointing input for local windows. When annexing display servers, that input must be redirected to remote windows. As annexing becomes more prevalent, personal devices may supply a second input source dedicated to remote windows. In the meantime, existing devices need to provide pointing input to remote windows.

2.6.2.1.1 Redirected Pointing Input

Each personal device needs a device-specific way to redirect pointing input to remote windows. Personal devices that accept stylus interaction would require interactive techniques like those used in Pebbles [39]. Personal devices that have other sources of pointing input (e.g., touchpad, mouse, arrow keys, or fingers) would need device-specific solutions for redirecting their input.

One way to redirect input to remote windows is to show a dialog that receives all input and then redirects that input, similar to Pebbles. For instance, on a laptop with only touchpad input, a small dialog could capture cursor movements and translate them into remote cursor movements. After each cursor move, the cursor would be recentered in the dialog so that remote cursor movement is not constrained by the bounds of the laptop screen.

Another way to share input between a personal device and a display server is to treat the personal device and display server as a unified display space. For instance, Synergy [63], MaxiVista [6], iRoom [30], and many other toolkits allow the user to slide the mouse off of one screen onto another screen. Related is the approach of Mouse Ether [7] which allows the mouse cursor to travel in the space between monitors so that the mouse does not jarringly jump to a destination monitor. However, such approaches may allow a window to overlap screens from two separate devices which may interfere with the privacy features that are discussed in section 2.8.

To capture stylus input from the personal device without shifting attention from the annexed screen requires a full-screen dialog on the personal device. A simple example of this dialog is shown on the left of Figure 2:19. If the dialog is not full-screen, the user may unintentionally tap widgets external to that dialog.

Suppose a message—like the one on the right of Figure 2:19—arrives while the user is looking at the annexed screen. If she accidentally taps within the message, she may respond to or ignore the message without realizing. She needs to be notified of the message without closing the capturing dialog or inadvertently interrupting input redirection. She also needs the ability to relinquish pointing input redirection, so she can interact with the message.

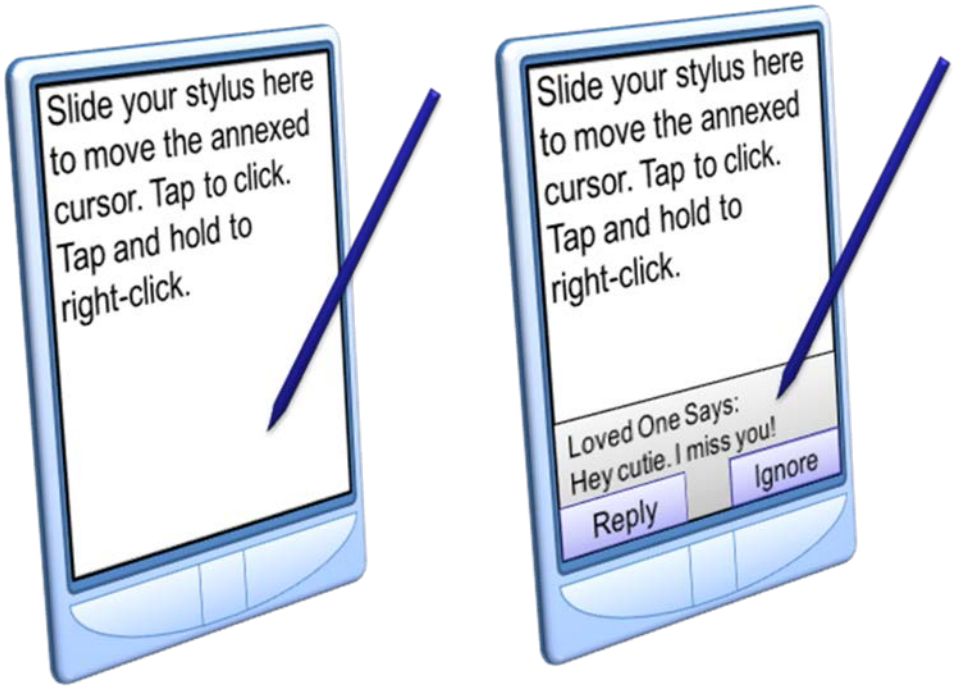


Figure 2:19 – A full-screen window can redirect stylus input as mouse input (left). A message may arrive at any time and inadvertently interrupt stylus input.

Instead of completely obscuring all window UIs on the personal device, the UI for a capturing window could be transparent. A transparent window UI allows the user to see her messages and any other local window UIs. Instructions on the transparent window's UI would let

her know how to access local windows. With XICE, when pointing input is sent to remote windows, the UI for a transparent *blocking window* is shown on the personal device (Figure 2:20). The blocking window's UI is shown on handhelds, laptops, and any other personal devices that have a single source for pointing input.

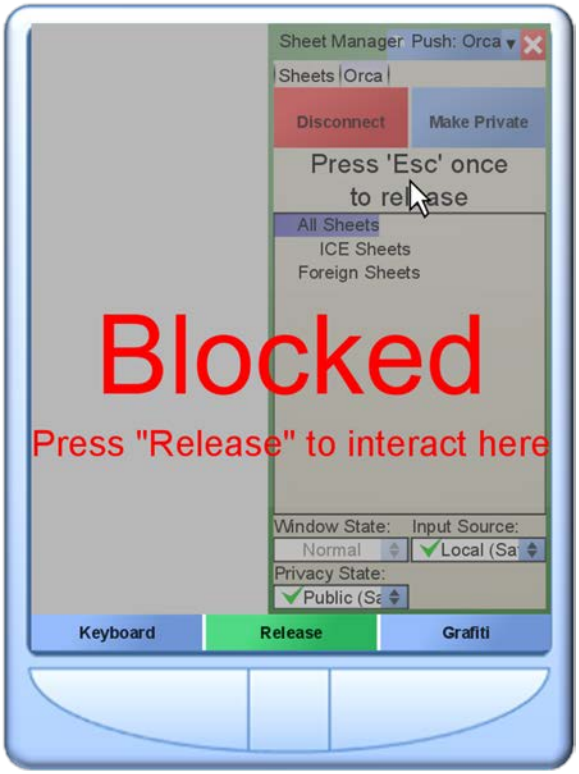


Figure 2:20 – The UI for blocking windows informs the user that input has been redirected from local windows to remote windows, and offers instructions for returning interaction to local windows on the personal device. The software buttons along the bottom cannot be interacted with directly: the user must use the physical buttons below them.

2.6.2.1.2 Independent Pointing Input

A straightforward way of supplying pointing input to local and remote windows is to provide a dedicated source of input on the personal device for each type of window. For example, similar to Oprea et al.'s device [47], a compact, inexpensive optical mouse sensor could be mounted on the back of a personal device. A prototype, called the MousePuter, has

been created using a Sony VAIO UX and the core hardware from an optical mouse (Figure 2:21).

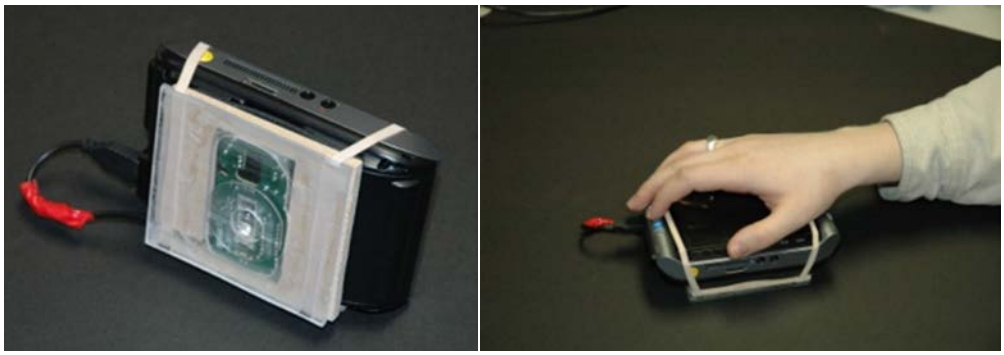


Figure 2:21 – MousePuter prototype.

The personal device can then be used like a mouse. The optical mouse on the underside of the personal device provides pointing input for remote windows; direct interaction with the personal device's screen supplies pointing input to local windows. Input redirection becomes unnecessary, eliminating the need for a blocking window. This solution works well for handheld personal devices, and the user can interact naturally with any display server that does not supply input or that the user does not trust.

2.6.2.2 Text Input

In addition to providing pointing input to remote windows, the personal device should provide text input. The personal device could provide a full physical keyboard, a small physical keyboard, a soft keyboard shown on the personal device, or a soft keyboard shown on the annexed display.

2.6.2.2.1 Full Physical Keyboard on the Personal Device

A personal device such as a laptop may provide a physical keyboard that requires little visual attention. A user may type with the keyboard while watching the annexed display. The

interaction with this particular device setup is natural, because people use tactile and visual feedback when entering text.

2.6.2.2.2 Small Keyboard or Soft Keyboard on the Personal Device

If the keyboard is small or soft, the user will typically spend most of her time looking at the keyboard while entering text. Without looking, touch-typing accurately is difficult in the case of the small keyboard, and nearly impossible with the soft keyboard. Consequently, she will want to regularly look at the screen to ensure she is entering text correctly. In such circumstances, she must frequently switch visual attention between the personal device and the display server to confirm text entered.

If, however, the text input area on the remote window is duplicated on the personal device, the user can confidently enter text by watching only the personal device. Sharp et al. [59] implement a form of this type of text entry. Unfortunately, in their solution, the local window shows a small portion of a pixel-by-pixel copy of the remote window's UI that does not properly fit the personal device's screen. The shown pixels are from the area of the window's UI directly surrounding the mouse cursor. As a result, the user must regularly move the mouse to keep the entered text within the personal device's screen. To compensate for this, the local window must be able to adjust its UI for the personal device's screen. If just the text entry widget were replicated, it would have to adjust as well.

XICE allows windows to adjust their UI to the personal device's screen. XICE clones the focused remote window's presentation tree to a new local window, as shown in Figure 2:22.

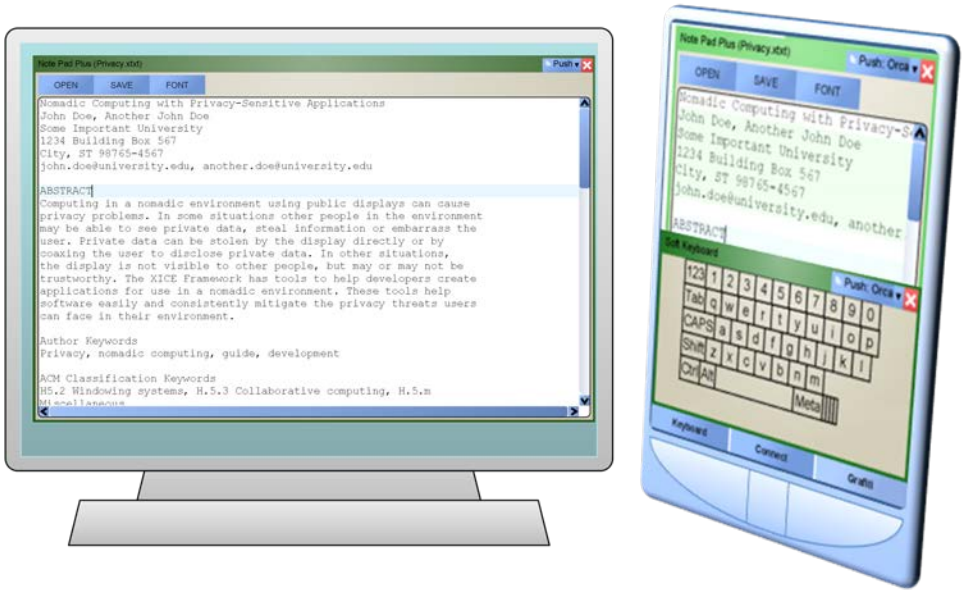


Figure 2:22 – Using the personal device's soft keyboard to provide text input for a remote window. The remote window is copied to a local window so the user can immediately see the entered text on the rendered UI.

The clone is created when a widget on the remote window requests text focus. Through model-view-controller (MVC) design, both the local window's presentation tree and the remote window's presentation tree share the same model. The XICE architecture makes cloning presentation trees simple and efficient without necessitating application intervention. XICE automatically ties each cloned widget to the original widget's model, rendering the cloning process trivial. Each clone may adapt its appearance according to the screen's parameters and the input available on the personal device.

If you are typing into an application using the soft keyboard and a message comes in from your spouse, then that message should be shown outside of the soft keyboard. The user should make an explicit command to change over to the messaging application so they can respond to the incoming message, rather than making an inadvertent reply that could accidentally appear on the annexed screen in the currently-being-edited window.

2.6.2.2.3 Soft Keyboard on the Annexed Display

The personal device could show a soft keyboard on the annexed display. The soft keyboard is easy to create and position near where the user is already looking. However, the display server can perform malicious acts with the soft keyboard. For instance, if the display server is blocked from supplying text input directly, the display server may still commandeer the keyboard and surreptitiously move it around the screen or rearrange the characters on the keyboard; as a user attempts to enter standard text, she might inadvertently send malicious text to the application. Consequently, XICE does not show a soft keyboard on an annexed screen unless the user completely trusts the display.

2.7 XICE Toolkit and Nomadic Experience

The XICE windowing toolkit has several major components that enable users to annex display servers safely and confidently. XICE is designed around the idea that users push UIs to display servers. The toolkit facilitates annexing devices and enables developers to seamlessly write code that operates in all three nomadic situations. This section will describe how XICE implements each of the nomadic situations: personal device alone, annexed screen only, and annexed screen and input. Regardless of the nomadic situation, applications must be able to build presentation trees, create windows, render each presentation tree on a window, and receive user input.

An application must request that a "space" create a window. A *space* is software on the personal device that represents the display area on which the window's UI will be rendered. A space tracks the screens, their sizes in VIC, and their relative locations. The space manages the size, position, and z-order of any windows rendered on that space. Spaces also perform the work necessary to show a window's UI on a screen. A *local space* creates local windows and

represents the personal device's screen resources, whereas a *remote space* creates remote windows and represents an annexed display server's screen resources. Local spaces direct windows to render their UI's on the personal device, while remote spaces direct windows to serialize their UIs to the display server.

To receive user input, a window must have an event source. An *event source* is software that tracks and directs input from a machine. This input could be from a single hardware device, from multiple hardware devices, or from multiple users on a single machine. (With multiple devices, each device is identified using a unique ID, similar to Multi-Pointer X [26]). The space automatically registers each new window with an appropriate event source. The event source uses the space to determine which window receives dispatched input (based on the size, position, z-order, and text focus of the windows' UIs). After the event source sends input to a window, the window dispatches the input to the presentation tree.

A *local event source* is an event source that dispatches user input from the personal device's hardware. A *remote event source* dispatches user input from a display server's hardware. A space may supply either type of event source; a window is assigned a single event source that is either local or remote. As XICE connects to a display server, its space is assigned a default event source based on user preferences: a local event source is assigned when the user does not want the annexed device to supply input, and a remote event source is assigned when the user wants the annexed device to supply input. When the space creates a window, the default event source is attached to the window.

Applications always execute on the personal device; they are never transmitted to a display server. Only the UI is serialized to a display server. As the user changes an application's nomadic situation, specific components used by the application may change (e.g., an application

may swap a local event source for a remote event source or vice-versa), but the overall UI architecture remains the same. The general software organization for all windows within an application is shown in Figure 2:23.

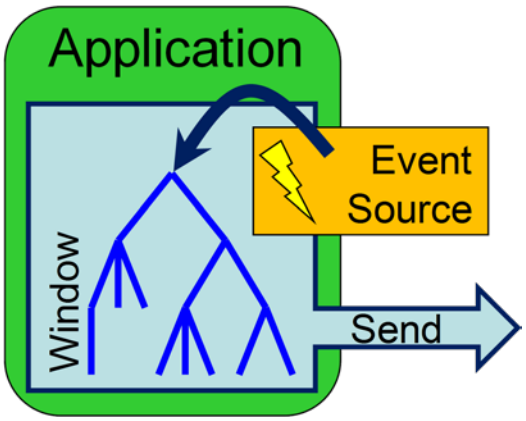


Figure 2:23 – Generic software organization for any application window. The window stores the presentation tree and dispatches events from the event source to the tree. The window serializes the tree or renders the UI.

All pointing input within XICE is stored and transmitted in VICs. For instance, a mouse location is stored in VIC's relative to the display space, and dispatched according to which window is directly under that mouse location. Button-click states are transmitted with the mouse movements and whenever the button-click state changes. Keyboard input is not related to VICs so keyboard input is dispatched to whichever widget currently has input focus.

Developers do not need to consider whether a space, a window, or an event source is local or remote. Developers only need to create presentation trees and assign them to windows.

2.7.1 Personal Device Alone

With XICE, users point, select, and execute as usual. When all input and output is on the personal device, self-contained interaction does not differ on personal devices such as laptops and PDAs. However, core application design and application startup processes diverge from existing systems.

A user must be able to launch an application on her personal device. The XICE framework, as implemented, provides an *application launch dialog* (Figure 2:24), and users select an application from its UI.

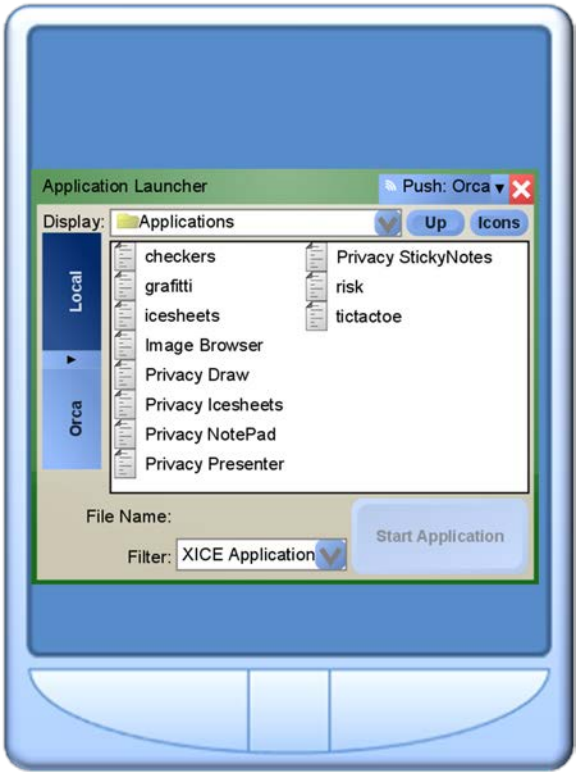


Figure 2:24 – Application launch dialog rendered on the personal device.

After the user chooses an application on the personal device, XICE starts the application and provides it the local space, so the application can show local window UIs. The software is organized in accordance with to the representation in Figure 2:25.

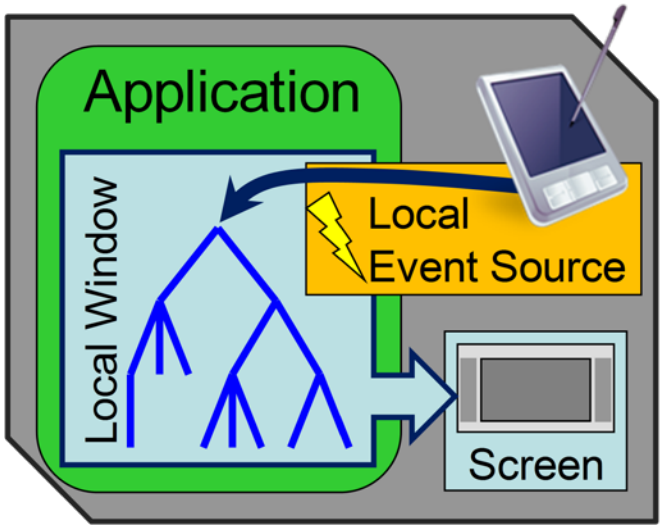


Figure 2:25 – Personal device alone software organization.

In the software organization for the personal device alone, the local space creates a local window and assigns the local event source to that window. The window then renders its presentation tree.

2.7.2 Annexing Display Servers

The process of annexing a display server should be as straightforward as possible, particularly for frequently used display servers, such as the personal desktop. In windowing toolkits, common window management functions are typically available in the corner of the title bar. XICE employs a similar approach and adds new functions for pushing a window to another screen. To reduce clutter and increase understandability, two buttons are presented to the user: the standard “close” button with one other (a drop-down provides access to other common functions). By default, a “Push” button is shown in the upper right corner of window UIs on the personal device (Figure 2:26). The user simply clicks the “Push” button to establish a connection with a display server and push that window’s UI to the display server.

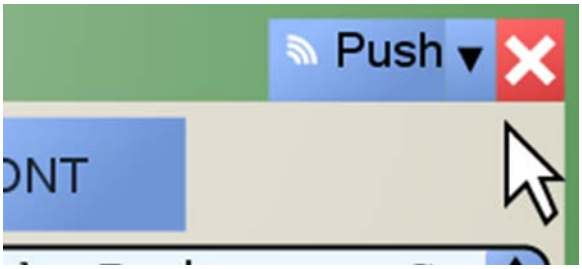


Figure 2:26 – The "Push" button initiates the process of annexing a display server. More options are available through a drop-down menu (inverted triangle).

More specifically, after a user clicks the “Push” button, a connection dialog’s UI is shown (Figure 2:27). The dialog lists display servers which the personal device has previously annexed. A configuration file is associated with each of those display servers. The user simply selects a display server’s configuration file, and the current window’s UI is pushed to that display server.

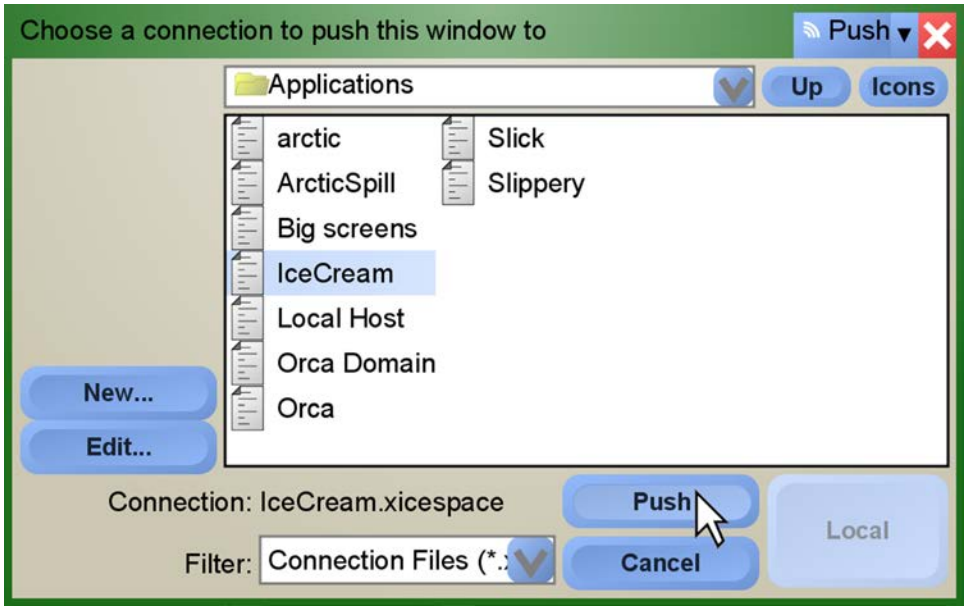


Figure 2:27 – XICE connection dialog UI. The user is pushing a window’s UI to the computer she named IceCream.

Each configuration file contains the following information:

- The name of the display server (as assigned by the user)
- The domain name or IP address of the appropriate XICE display server

- Whether the display is trusted to show sensitive data (default: distrusted)
- Whether the display server's input devices are trusted (default: distrusted)
- Where input comes from (default: local event source)

These configuration files are intended to identify display servers the user connects to regularly, like a desktop or a home television. In such situations, the user has a specific trust setting meant for each respective display server. Saving these settings in a file simplifies the annexation process for subsequent connections to a particular display server.

People may also annex display servers they have never used before. To annex a new display server, the user clicks the "Push" button, requests a new connection, and enters the domain name or IP address of the display server. The connection dialog clearly shows a "New…" button (Figure 2:27) users can click to initiate a connection with a new display server. The user must be made aware of the display server's domain name or IP address; otherwise, she cannot connect to the display server. XICE can show that information via a dialog on the display server, or the display's owner can post that information on a physical sign near the display server.

Editing a configuration file is inconvenient for users. XICE provides the dialog in Figure 2:28 for editing the properties stored in the display's configuration file. The green check marks represent the default, safe configuration for those two properties.

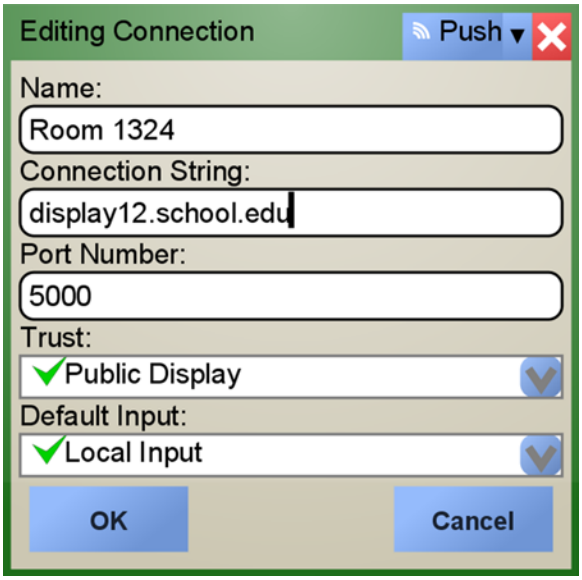


Figure 2:28 – XICE connection properties dialog. The "Default Input" drop-down box is used to select which device input comes from.

When the user connects to a display server, the XICE toolkit provides an additional dialog on the personal device to allow the user to manage the annexed screen. This dialog lists the windows on the annexed screen and provides configuration options for those windows and for the display server. One such option (shown in Figure 2:29) allows the user to change the hardware input source.

Figure 2:29 – An option on the personal device for changing the hardware input source.

The connection process can be accelerated with broadcast techniques such as Bonjour [3] and location services [67]. These techniques filter results based on the personal device's wireless access point or Global Positioning System (GPS) location, enabling a user to easily and accurately select a nearby display server. Such connection facilitators could be used by XICE but are not essential to this discussion.

Once a display server has been selected, the scene-graph on the current local window is attached to a remote window. Remote windows' UIs show the "Pull Back" option (Figure 2:30) in place of the original "Push" button. The "Pull Back" button allows a user to quickly remove individual scene-graphs from the display server and closes the respective remote window.

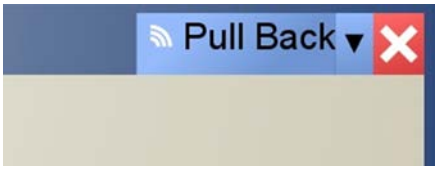


Figure 2:30 – A window's UI that has been pushed from a personal device to a display server has a "Pull Back" button that enables the user to easily remove that window's UI.

After establishing a connection, pushing scene-graphs to the annexed display server is simple. The "Push" buttons for all local windows change to "Push: *Connection*", where "*Connection*" is the name of the most recently connected display server. For example, clicking the "Push: Orca" button in Figure 2:31 sends the scene-graph to the machine the user named Orca.

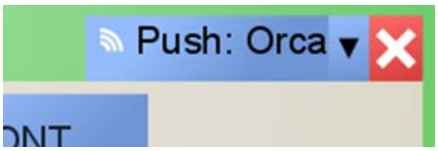


Figure 2:31 – After a connection is made, local windows prepare to push their UIs to the annexed display server.

After a scene-graph is attached to a remote window, the scene-graph is serialized to the annexed display server. Then, the application executes normally.

Some applications may need to know if a user has pushed their scene-graphs to an annexed display. In particular, widgets may need that information so they can adapt their appearance based on the display server or context. For example, to protect email addresses in a scene-graph from being shown on a public display—where other people or the display server

might steal them—the *To* and *From* address boxes would remove those addresses from the scene-graph (described in more detail in the Protecting Distributed Applications section). To protect sensitive data, applications must be informed that a scene-graph has been pushed to a distrusted display server.

XICE notifies each presentation tree when it has been moved to a different window. When the user pushes a scene-graph, XICE creates the remote window, moves the presentation tree to the remote window, closes the local window, and then serializes the scene-graph to the annexed display server. Before serialization, all widgets within the tree are sent a *recontext event* to notify them that the scene-graph has been attached to a new window. The recontext event provides each widget an opportunity to change its appearance according to the window's context. Most widgets ignore the event and pass it to all child nodes. However, widgets that adapt to context can alter the presentation tree. After all widgets have processed the recontext event, serialization proceeds normally.

In addition to pushing a window's UI from the personal device to the annexed screen, a user may launch an application on the personal device directly from the display server. The user clicks on the display server's desktop and the personal device shows the application launch dialog's UI (Figure 2:24) on the display server. The dialog's UI lists all the applications installed on the personal device; any applications that might be installed on the display server are not listed. The user selects an application that launches on his personal device, and XICE supplies the application with the remote space for creating windows. However, launching applications directly from a display server has security and privacy implications (Protecting Distributed Applications: section 2.8).

2.7.2.1 Annexed Screen Only

If a display server does not provide input devices, or if a user distrusts the display server's input devices, then the user is in the annexed screen only situation. Figure 2:32 illustrates the software organization for a remote window with input from the personal device.

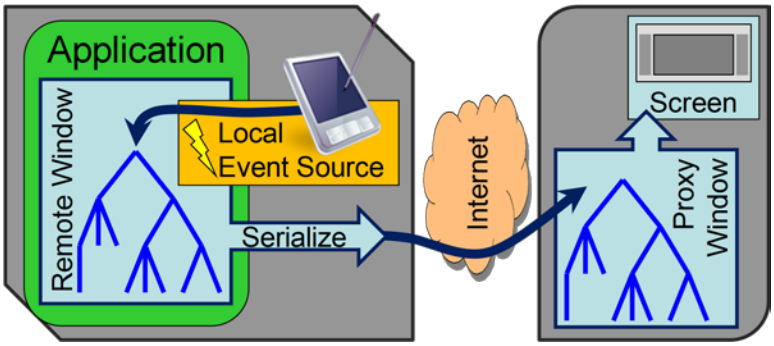


Figure 2:32 – Annexed screen only software organization.

The user needs to provide pointing and text input to remote windows using the personal device. The remote space supplies a local event source to attach to new remote windows. The local event source tracks the cursor's position and shape, and transmits that information to the display server for presentation to the user (Figure 2:33).

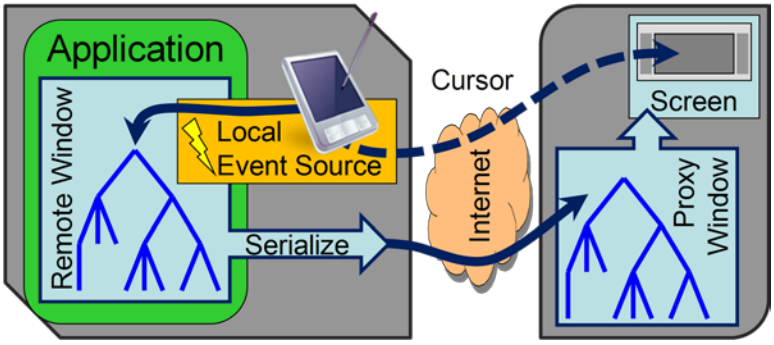


Figure 2:33 – Remote cursor echo.

2.7.2.2 Annexed Screen and Input

In the richest nomadic situation, a user interacts with data directly using the display server's input devices. The user could annex a display server, then set the personal device down

or place it in her pocket. Although the personal device would continue to run her applications, she would interact exclusively via annexed devices.

To annex a screen and its input devices, the user changes the configuration settings to accept annexed input (Figure 2:28 and Figure 2:29). After this change, a remote event source is attached to each new remote window on that remote space. Remote windows render on the annexed display, input is processed on the personal device, and scene-graph changes are serialized to the display server. The annexed screen and input software organization is illustrated in Figure 2:34.

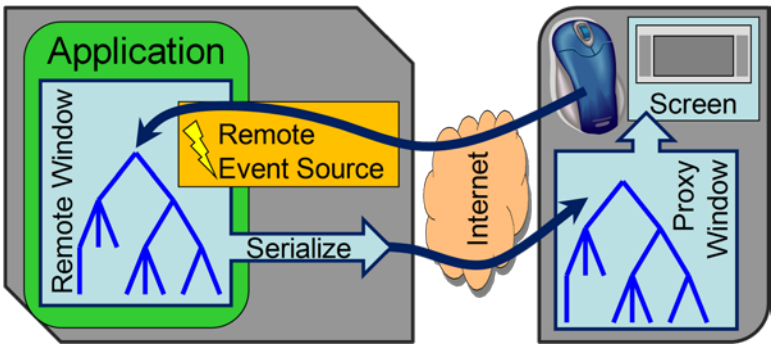


Figure 2:34 – Annexed screen and input software organization.

2.7.2.3 Annexed Input

Another potential situation is if the user just wants to accept richer input for their personal device. For example, the user may want to annex a keyboard and/or mouse so that he can have a richer typing experience or finer pointing ability within his applications. In this configuration, the display server's input is routed across the network to the personal device and then transmitted to the user's applications.

To establish such a connection, the user is no longer pushing a window to a display server. Instead, he is grabbing input from a display server. He must establish such a connection

through a channel other than the “Push” option from Figure 2:26, but XICE’s software can easily handle such an organization. This software organization is illustrated in Figure 2:35.

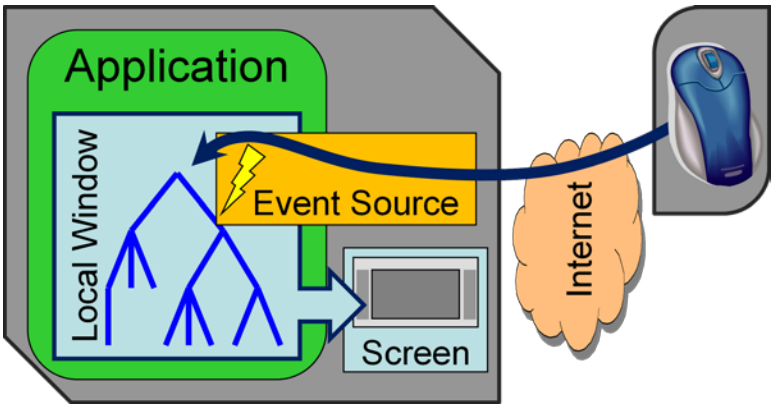


Figure 2:35 – Annexed input software organization.

2.8 Protecting Distributed Applications

One major reason a user may enter into the annexed screen only situation—and then have to provide all input from the personal device—is that she may not trust the annexed display server to provide input. As mentioned earlier, some significant security risks are introduced by distributing a UI to annexed screens. Four threats specific to UI distribution are: stolen input, false input, stolen output, and false output.

With XICE, the personal device performs application processing to protect a user’s sensitive code and data; code and data are never distributed to the display server, so the display cannot directly steal that data. User input from the local event source is never routed through the display server, so the display server cannot redirect user input to different widgets or change the input the user supplies.

XICE has built-in provisions for privacy. The details of the levels of privacy protection XICE supplies are unimportant for this discussion. What is salient is that display servers may be assigned a *privacy state* of *public* (distrusted) or *private* (trusted). The user specifies whether the

display server is distrusted or trusted in the display server configuration file. Input from a display server may also be considered distrusted or trusted, and this setting is independent of the privacy state.

Most distributed UI solutions do not provide applications with information about the privacy state of prospective displays. If an application does not have access to the privacy state of a display, the application cannot take appropriate protective action. Consequently, each privacy-aware application must have its own way of detecting the privacy state of a display, usually by allowing the user to input that state directly to the application. Symbiotic Displays [9] takes this approach, allowing the user to specify the privacy state of a display server directly to the application.

Instead of having a per-application privacy-aware solution, the windowing toolkit should track the privacy state of the display, provide that information to applications, and inform applications of any privacy state changes. Applications, on the other hand, must inform the toolkit of sensitive input and output. By making the privacy state available and identifying sensitive input and output, the toolkit and applications can proactively protect sensitive data.

XICE provides the privacy state in the device context passed to each widget when its presentation tree is attached to a window or when an event is dispatched to the presentation tree. The most important event for privacy is the recontext event. To protect sensitive data, applications that respond to the recontext event may change their UI before distributing it to a public display server. In addition, applications may tag presentation trees or sections of presentation trees as sensitive, and the XICE toolkit can protect those trees or sub-trees from potentially malicious display servers.

2.8.1 Stolen and False Input

Stolen input occurs when the display server overtly steals sensitive input (e.g., usernames and passwords) directly from the user. False input happens when the display server impersonates user input to applications. False input usually happens in an attempt to get the user to expose sensitive data (e.g., email addresses) to the display server so that it can steal that data.

To protect against stolen input and false input, XICE automatically configures any new connection to block input from a display server. If user input from the display server is blocked, the display server cannot supply false input to the user's applications. Also, the display server is less likely to obtain sensitive input directly from the user.

When the user elects to accept display server input (the annexed screen and input situation), XICE encourages her to distrust that input. So, a display server may supply input that the personal device dispatches, but the input is tagged—per her settings—as either distrusted or trusted. This tagging allows an application to identify what the user considers unsafe input and take appropriate action with respect to that input.

2.8.1.1 Distrusted Input

Most input a user enters is not sensitive and does not affect sensitive data. In these situations, the user can seamlessly supply input using annexed input devices.

When annexed input affects data the software knows is sensitive, the software should ensure that the user explicitly confirms that action on the personal device before implementing changes. For example, XICE supplies the dropdown menu in Figure 2:36 on all remote window UIs on public display servers. If the user clicks the "Show Private Data" option, the window could expose all of the user's sensitive data on that UI. However, because input to the "Show Private Data" option is distrusted, XICE explicitly confirms the action on the personal device

prior to exposure of any sensitive data. XICE's event handler for that menu option checks the trust tag on the mouse input, discovers that the input is distrusted, and shows a confirmation dialog on the personal device. The user must confirm the menu click on her personal device before the window will explicitly show the user's sensitive data on that public screen.

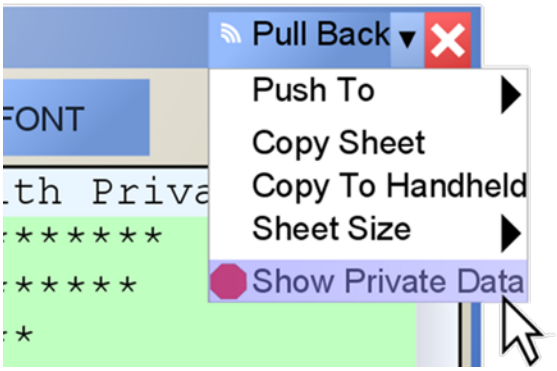


Figure 2:36 – Options such as "Show Private Data" can be exploited by malicious display servers. XICE mitigates such acts by explicitly confirming privacy-affected commands on the personal device.

The confirmation dialog also alerts the user when the display server surreptitiously attempts something malicious. In addition, when a widget expects sensitive input (e.g., usernames and passwords), the widget can check the trust level of the input source; the sensitive input can then be requested on the personal device, preventing the display server from accessing sensitive data. Both of these features further protect the user against stolen and false input.

If an application supplies an option similar to the "Show Private Data" option, the application must also enable the user to confirm that option on the personal device; applications can easily detect when input is distrusted, so they can properly confirm such actions.

The ability to launch applications on the personal device directly from the display server is a known security threat. Unbeknownst to the user, the display server might launch a susceptible application to exploit its vulnerability. On the personal device, XICE automatically

requires the user to confirm application launch requests originating from distrusted display servers.

2.8.1.2 Trusted Input

If a user completely trusts a display server (like she would at a personal desktop), then none of XICE's security measures are enforced. Users have a seamless experience with fully trusted annexed display servers.

2.8.2 Stolen Output

Any data shown on a display server—including sensitive data like usernames, email addresses, and phone numbers—can be stolen by that display server and potentially used maliciously. The easiest solution is for users to never annex any display servers, but that would unnecessarily limit users to the resources on their respective personal devices. In addition, a trustworthy screen may be available in a public place. For instance, a screen in a conference room might be trusted because the company properly maintains the display server, but a presenter may not want sensitive company data to be divulged to an audience of vendors.

To protect sensitive data, applications must be aware of public environments. The user should account for the public nature of an environment by setting the display server's privacy state to public. When the application is aware the display server is public, the application can appropriately protect the user's sensitive data. XICE stores a device's privacy state as a property of the device's space. The local space is always private because it represents the personal device, but a remote space may be public or private.

In XICE, the user can change a remote space's privacy state at any time. For instance, if a user is at her private desktop and a coworker approaches to discuss some work, the user may change the desktop to public, protecting her sensitive data from the visiting coworker. When the

coworker leaves, she can change her desktop back to private. XICE sends a recontext event when the privacy state of a display changes, informing all windows and widgets of the change.

Applications that use sensitive data may need to proactively protect that data. The developer of an email application may design it to ensure that each email is initially shown on a private display. To implement this functionality, the email application checks the space's privacy state before creating an email window; if the space is public, the application shows the email window on the personal device instead of on the public display server. If the user chooses to push an email to a public display, the presentation tree receives a recontext event that contains the privacy state for the new remote window. The "To" and "From" widgets receive the recontext event and can then, like Symbiotic Displays [9], protect embedded email addresses.

A simple way to protect sensitive data is to overlay black rectangles. Privacy Blinders [64] uses this approach to protect sensitive data. Even if viewers cannot see the sensitive data, the display server still has access to the information, since both the sensitive data and the overlaid rectangles are sent to the display server. Widgets that protect sensitive data must scrub that data from the presentation tree before the tree is serialized to a public display server. The recontext event affords widgets opportunity to remove sensitive data from the presentation tree and ensures that no sensitive data is inadvertently sent to a public display.

XICE provides some automatic systems to help developers protect sensitive data. For instance, file dialogs may reveal sensitive data (e.g., file names or file system structure) outside of an application's control. Therefore, file dialogs should not show on a public display. However, expecting each developer using the file dialog to properly implement this restriction is not practical. Instead, the file dialog widget should automatically ensure that the file dialog only shows on private displays.

To guarantee that a particular dialog only appears on private displays, XICE requires that the dialog be tagged with the *private only* Java annotation. The UI for the dialog is represented in a presentation tree that is rendered on a display, so the root of the presentation tree must have the private only tag. XICE is designed so that public spaces check for the private only tag and automatically redirect a tagged presentation tree to a new window on the personal device. The private only annotation is only effective on the root widgets in a presentation tree. Applying that annotation anywhere else is not protected.

If developers create a subclass of a dialog that is tagged private only, the dialog remains private only. Because XICE automatically redirects these dialogs to the personal device, developers can use the original or subclass dialogs without implementing redirection code or explicitly protecting those dialogs within the application. So the user's private file system is protected consistently across applications.

When a file dialog is opened on a public space and consequently redirected to the personal device, the user needs to shift her visual focus from the annexed screen to the personal device. However, if she expects the dialog to be rendered on the public display she may be confused. To minimize this confusion, XICE automatically renders a *heads-up dialog* on the public display whenever a private only dialog is redirected. The UI for the heads-up dialog instructs the user to look at her personal device (Figure 2:37).

Figure 2:37 – File dialogs are not sent to public displays. Instead, a *heads-up* dialog rendered on the public screen directs users to the file dialog's UI on the personal device.

In the annexed screen only situation, input on the personal device is typically directed at remote windows. If a user performs an action on a remote window's UI which creates a dialog that is redirected to a local window, then input is also automatically redirected to local windows. When the dialog closes, XICE automatically redirects input back to the remote windows.

The heads-up dialog is likely to work best in situations where a user's action caused a redirected dialog to appear. If an application attempts to show a private window independently of user action and the heads-up dialog appears, then the other users at the display space may become confused as each tries to figure out if he is the owner of that dialog. Such software design decisions are discouraged.

2.8.3 False Output

Display servers could overtly falsify the user's application output. *Falsifying output* includes any effort to coax users to expose sensitive data that the display server can then steal. For example, selecting the "Show Private Data" menu option in Figure 2:36 causes an

application to expose the user's sensitive data. A malicious display server might expect the user to click the "Copy Sheet" option, but wants the user to click the "Show Private Data" option. To accomplish this, the display server would swap the text of the two menu options, so the user thinks she is clicking "Copy Sheet" when, in reality, the personal device interprets that click as "Show Private Data".

A similar situation could occur when a user interacts with the application launch dialog on a malicious display server. The display server might rearrange application names or reposition the mouse cursor to coax the user into launching applications that the display server could then exploit.

To mitigate the false output problem, critical inputs or outputs are shown only on private devices. If the annexed device is public, XICE renders a confirmation dialog on the personal device, and a heads-up dialog rendered on the display server lets the user know she needs to enter input on the personal device. By double-checking critical actions, XICE discourages a display server from falsifying output and prevents the display server from surreptitiously stealing the user's sensitive data.

2.9 Benefits of the XICE Protocol

XICE is a windowing toolkit that offers a seamless nomadic experience and helps mitigate or solve a wide array of problems related to creating a nomadic computing environment. In particular, XICE is network-based to provide easy connection to a variety of display servers and to accept input from them. The network-based protocol also allows multiple users to annex a display server simultaneously. Unlike with VGA connections, XICE users may show their personal applications on the same shared display to compare information.

Scene-graphs defined in VIC allow XICE to distribute UIs to display servers in myriad viewing situations that include multiple sizes, resolutions, and viewing distances. Using the scene-graph increases the UI's ability to adapt to window size, screen dimensions, and screen resolutions; reduces personal device CPU requirements; and lengthens battery life on the personal device by offloading the rendering burden to the display server.

XICE encourages development and implementation of a stable display server platform. This stability results from a simple communication and rendering framework. One key benefit of a stable platform is that users and developers can trust that their application UIs will have the same appearance, regardless of the device annexed. Another benefit is that users do not need to reconfigure their personal devices to deal with each new display server's installation.

If a user chooses to accept input from a display server, she will have an interactive experience similar to her desktop. However, display servers might not supply input devices, or the user may distrust input from the display server, so XICE encourages users to annex only a display server's output. Consequently, the personal device should be used for all input to the user's applications. When the user provides input on the personal device, XICE can smoothly transition between local windows and remote windows. Particular devices, such as the MousePuter, can leverage XICE to naturally transition between local and remote windows by supplying dedicated input hardware for each type of window.

XICE has several features that protect sensitive data from malicious displays. First, core application processing is on the user's personal device, so a malicious display server cannot steal or alter that data. Second, annexing input is discouraged. Blocking display server input ensures that the input comes from a trusted source: the personal device. Third, accepting but distrusting input from the display server allows applications to require the user to confirm actions that

include or affect sensitive data. Fourth, XICE sends applications the display server's privacy state so that private data in the application's output can be protected. Fifth, XICE automatically prevents sensitive dialogs, such as file dialogs, from rendering on public displays.

2.10 Developer Experience

A key factor that influenced the design of XICE is how effective it is for developers to learn and use. For example, in section 2.5.3, this article mentioned the developer confusion in relation to DVA.

There have been roughly a dozen developers who have used XICE to create applications. The developers in the lab pick up XICE relatively quickly. The overall scene-graph structure takes little time to understand and start using. It takes only a few hours to master the overall usage of XICE, from building scene graphs to showing new windows, especially since developers do not have to deal with a distributed application unless they choose to.

There have been several dozen applications built using the XICE windowing toolkit and framework. Some of these test specific techniques while others pursue research avenues independently of XICE. Some of the key programs implemented in XICE are a spreadsheet application (called IceSheets), a text editor, a simple drawing application, and presentation software. These applications have varying levels of sophistication, as can be observed in the screen shots shown in Figure 2:38. IceSheets is designed for research into new avenues with spreadsheets and represents more than six hundred hours of developer effort. Most of that effort was spent implementing the various mathematical functions for the spreadsheet. Little effort went into UI development. In Figure 2:9, the user has "spilled" the IceSheets application and is panning it to another position. Other programs are games, such as Checkers, tic-tac-toe, or Risk, which are for training developers or pushing the graphical bounds of the toolkit. Checkers

represents about 40 hours of developer effort; tic-tac-toe about 8; and Risk—which includes some automated game-playing capabilities—about 100.

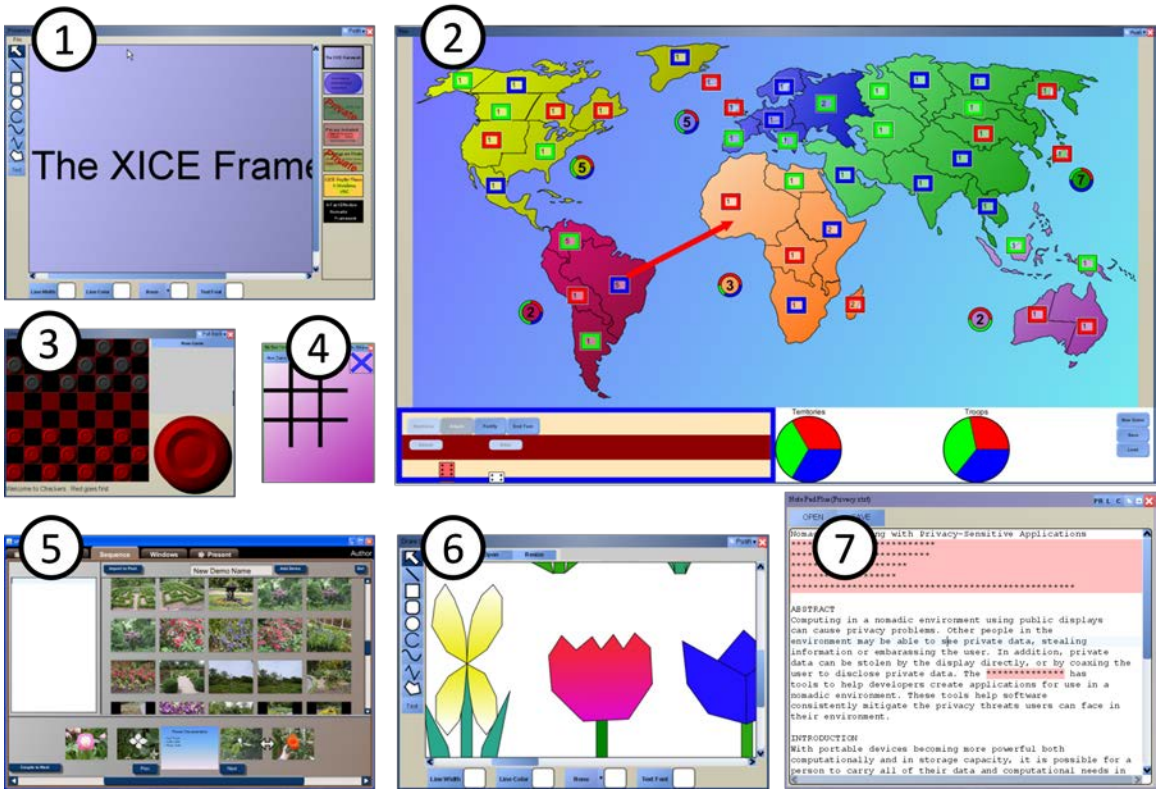


Figure 2:38 – Various applications implemented with XICE. 1) A presentation tool. 2) Risk™. 3) Checkers. 4) Tic-Tac-Toe. 5) A more sophisticated presentation tool. 6) Drawing application. 7) Privacy-aware text editor.

When the privacy toolkit was added to XICE, IceSheets was already complete, but an experienced developer retrofitted IceSheets to support privacy by hiding private columns and rows, and graying out private cells. This retrofitting consumed about 8 hours of developer time, including the time it took to alter the data format to store the custom values necessary to track which columns, rows, and cells contained private data. The data issues are not XICE related.

2.11 Limitations of the XICE Protocol

There are some limitations to the XICE implementation. Some of these are limitations of developer resources; others are a limitation of the technology itself. This section will discuss several of these important limitations.

2.11.1 Backward Compatibility

A key limitation of XICE is that all applications must be rewritten for the XICE platform. When developing this framework, the authors considered this limitation and chose to forgo backward compatibility for several reasons.

First, although RDP supports network transmission of WPF scene-graphs, this fact is not widely known or easily discoverable. In addition, the source code for RDP is not available for augmentation to handle the aspects of multiple input devices, multiple focused windows (one per connected user), input redirection, and privacy that the authors need.

Second, coercing RDP or X11 into Java and then to support multiple input devices, multiple focused windows, input redirection, scene-graphs, and privacy is beyond the resources of the authors, especially when trying to provide timely research. It was easier to build the framework from scratch.

Third, separating the display spaces made for a natural addition of privacy because privacy may be attached to a display server.

Fourth, the paradigm for nomadic application development is fundamentally different from how legacy applications were built. With this new paradigm, a single application may simultaneously show separate windows on different displays and even display servers, with each window getting input from different sources. Legacy applications do not understand operating in this kind of environment. These applications are designed under the assumption of a single,

unified display space where a single user interacts. Although backward compatibility would be nice for these applications, the authors think that many of these applications need to be rewritten simply because of the paradigm shift to a more flexible environment.

For example, consider the desktop and Windows Mobile versions of Microsoft Outlook: neither version handles swapping between a tiny screen and a large screen, let alone rendering on both screens at once. The desktop version is only usable on an annexed large screen, while the mobile version can only show small windows on an annexed screen. The desktop version of outlook is pointless if it is useless on a small screen. The mobile version gains no benefit from annexing a screen. A new version or Outlook that can simultaneously handle both a large and a small screen should be implemented. This new version should incorporate privacy-aware interaction: for example, show a window for privately selecting contacts via the smartphone while allowing the user to type up a new email on an annexed screen.

Fifth, smartphones released in recent years (e.g., the iPhone [2] and Android [21]) have shown that people are willing to rewrite applications for different devices, especially if rich toolkits are available for these devices. Although Android uses the Java syntax and a byte language, it is not backward compatible with existing Java applications, especially because the Android has a different rendering framework. Other technologies have also shown that developers are willing, if not eager, to rewrite their applications to adopt the new technologies. This happened with the advent of HTML and again with AJAX, especially when rich toolkits became available for these technologies.

But, regardless of these reasons to not be backward compatible, backward compatibility should be incorporated in the future, especially because users have existing applications on laptops that should also fit into the ecosystem afforded by XICE.

2.11.2 Rendering Performance/Capabilities

In some situations, damage/repaint is faster than a scene-graph. This happens when the scene-graph takes longer to serialize than to render the pixels and serialize the frame buffer. For instance, scatter plots with hundreds of thousands of points or complicated CAD diagrams may exhibit this property.

Pixel painting and photo editing are also not efficiently supported by XICE. To implement these, for each change made to the image, the client device must render to a new pixel buffer and serialize that buffer to the display server.

If a damage/repaint option is integrated into XICE, both of these problems may be addressed.

2.11.3 Collaboration

Copy & Paste between application windows supplied by different personal devices is not supported. For example, when two people share a display space, the first person cannot copy an appointment to the other person's calendar.

Part of this limitation is caused by the perspective of the authors that XICE should inherently distrust display servers. Consequently, a display server might not be trustworthy to either broker a connection between the two users or to transmit data from the first user to the second. The authors have not focused on this problem, but hope to spur thought in this direction as users need more seamless collaboration via shared display spaces.

2.11.4 Interaction Distance

XICE is not intended for large network distances: hosting an application in New York and interacting with it in California is not likely to produce a favorable user experience. Technologies such as HTML, AJAX, etc. provide a better experience over large distances

because they transmit code to the web browser for execution. With code executing at the browser, the UI is more responsive.

However, with XICE, the user's personal device is in the room with the display server. So, the physical and network distances are short. Transmitting code is not necessary to gain the needed interactive richness and responsiveness.

2.11.5 Animating Graphical Primitives

A common feature of modern scene-graph toolkits is animation of graphical primitives. Unfortunately, XICE does not support such animation except through application-level implementations (i.e., an application developer must create an animation loop to update the discrete position of shapes). So far, with a low latency in the network, these animations are nearly seamless. However, serializing animations to the display server may have great performance benefits.

Animations are a natural extension of scene-graphs, as WPF has shown. An animation engine could easily be incorporated into the XICE scene-graph architecture and animation instructions serialized to the display server for execution. Many animation frameworks exist that could be used as a model for a XICE animation framework.

2.11.6 Video

As mentioned in section 2.5.1, XICE has no video support. This support was not incorporated because the XICE rendering engine is implemented in Java and JMF [49] has poor video support, especially transforms, transparency, and graphical overlays to the rendered video.

If the XICE rendering engine were a platform-specific rendering engine (e.g. WPF or Cocoa [4]) which supports deep integration of video with other graphical primitives, then support for simple video rendering and manipulation (e.g. play, pause, fast forward) is straightforward.

The XICE scene-graph would be transmitted to the platform-specific rendering engine, which would interpret that scene-graph accordingly.

2.11.7 3D Graphical Primitives

Additionally, XICE does not provide facilities for 3D graphics, such as might be used in gaming or visualization scenarios. The framework was designed with 3D graphics in mind as a potential addition, but this was not included in the prototype. Incorporating work similar to blue-c's distributed 3D scene-graph [40] appears to be straightforward.

2.12 Future Work

In the immediate future, researchers and developers can design or re-implement widgets to be compatible with the XICE windowing toolkit, integrate and test multiple personal devices and display technologies, and examine numerous collaborative work environments.

Through XICE, input and output can be easily annexed, so widgets will need to be redesigned to handle varying input sources and output sizes. Input devices will range from the common mouse and keyboard to pens, styluses, fingers, lasers, game controllers, gestures, and eye tracking. Hard-coding widgets to meet every possible combination of input device and output device is excessively complicated; therefore, widgets and software need to be designed to handle a more abstracted version of input that can be stored and transmitted in a common way. The exact form of input/output storage and the precise mechanism of input/output transfer also need to be determined.

XICE does not transmit any data other than scene graphs and device input, but interaction with devices in the environment may require transmitting other kinds of data. For example, an environment may have a printer with advanced features that the user wants to print through. The user should be able to annex that printer and send documents to it (probably using PostScript

[23]) and use that printer's advanced features using a UI that is familiar to the user (i.e., matches the user's look and feel on her personal device, but supplied by the printer, possibly using XICE). Speakeasy [18] is one such approach for combining nearby resources in flexible, powerful ways. Device Ensembles [61] lays out a good taxonomy of where these design decisions need to take placed. XICE is an example of a technology that fits in the "Application" (input and output redirection) and "Data Format" (standardized scene-graph and hardware input) groups in the Device Ensembles taxonomy. Such research should be explored in conjunction with the XICE interaction model.

XICE is designed to support multiple users with a variety of input devices and display server configurations. Exploration of scenarios and implications in the collaborative realm, with well-designed user studies, would be appropriate.

2.13 REFERENCES

1. Adobe Systems, Adobe Flash, http://get.adobe.com/flashplayer/, 1996, accessed June 2010.
2. Apple Computer, iPhone, http://www.apple.com/iphone/, accessed June 2010.
3. Apple Computer. Bonjour, http://www.apple.com/support/bonjour/, accessed June 2010.
4. Apple Computer. Cocoa, http://developer.apple.com/technologies/mac/cocoa.htmh, accessed July 2010.
5. Argue, R. *Advanced multi-display configuration and connectivity*. MS Thesis, Dalhousie University, August 2007.
6. Bartels Media GmbH, MaxiVista, http://www.maxivista.com/, accessed January 2011.
7. Baudisch, P., Cutrell, E., Hinckley, K., and Gruen, R. "Mouse Ether: Accelerating the Acquisition of Targets Across Multi-monitor Displays." In *CHI '04: CHI '04 extended abstracts on Human factors in computing systems*, pages 1379-1382, New York, NY, USA, 2004 ACMPress.

8. Bederson, B. B., Grosjean, J., Meyer, J., “Toolkit Design for Interactive Structured Graphics.” *Software Engineering*, IEEE 2004, 535-546.

9. Berger, S.; Kjeldsen, R.; Narayanaswami, C.; Pinhanez, C.; Podlaseck, M.; and Raghunath, M. 2005 “Using Symbiotic Displays to View Sensitive Information in Public.” In *Third IEEE International Conference on Pervasive Computing and Communications* (PerCom 2005) IEEE Computer Society, 139-148.

10. Bharat, K., Cardelli, L. “Migratory Applications,” In *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg 1997, pp 131-148.

11. Biehl, J. T., Baker, W. T., Bailey, B. P., Tan, D. S., Inkpen, K. M., and Czerwinski, M. 2008. “Impromptu: a new interaction framework for supporting collaboration in multiple display environments and its field evaluation for co-located software development.” In *Proceeding of the Twenty-Sixth Annual SIGCHI Conference on Human Factors in Computing Systems* (Florence, Italy, April 05 - 10, 2008). CHI '08. ACM, New York, NY, 939-948.

12. Bier, E. A., Stone, M. C., Pier, K., Buxton, W., and DeRose, T. D. “Toolglass and Magic Lenses: the See-Through Interface.” *Computer Graphics and Interactive Techniques (SIGGRAPH '93)*, ACM (1993), pp 73-80.

13. Canonical Ltd., Ubuntu 10.10, http://www.ubuntu.com/, accessed January 2011.

14. Chapuis, O., and Roussel, N., “Metisse is not a 3D Desktop!” *User Interface Software and Technology (UIST ’05)*, ACM (2005), pp. 13-22.

15. Cisco Systems, Inc. WebEx, http://www.webex.com/, 1997, accessed June 2010.

16. Citrix Systems, Inc. Citrix Online, http://www.citrixonline.com/, 1997, accessed June 2010.

17. Edwards, W. K., Hudson, S. E., Marinacci, J., Rodenstein, R., Rodriguez, T., and Smith, I. “Systematic Output Modification in a 2D User Interface Toolkit”, *User Interface Software and Technology (UIST ’97)*, ACM (1997), pp 151-158.

18. Edwards, W. K., Newman, M. W., Sedivy, J., Smith, T., and Izadi, S. 2002. “Challenge: recombinant computing and the speakeasy approach." In *Proceedings of the 8th Annual international Conference on Mobile Computing and Networking* (Atlanta, Georgia, USA, September 23 - 28, 2002). MobiCom '02. ACM, New York, NY, 279-286.

19. Equalizer Graphics, http://www.equalizergraphics.com/, 2008, accessed January 2011.

20. Flanagan, D. 2006 *JavaScript: the Definitive Guide.* O'Reilly Media, Inc.

21. Google Inc., Android, http://www.android.com/, accessed August 2010.

22. Gosling, J., Joy, B., Steele, G., and Bracha, G. *Java Language Specification, Second Edition: the Java Series*. 2nd. Addison-Wesley Longman Publishing Co., Inc. 2000.

23. Gosling, J., Rosenthal, D., and Arden, M. The NeWS Book: An Introduction to the Networked Extensible Window System, Sun Microsystems (1989).

24. GKS (Graphical Kernel System), ANS X3.124-1985 ANSI, Dec. 1984.

25. Howard, M. and LeBlanc, D. *Writing Secure Code, Second Edition*, Microsoft Press, 2003.

26. Hutterer, P., and Thomas, B. H. "Groupware support in the windowing system." In *AUIC'07: Proceedings of the eight Australasian conference on User interface*, pages 39-46, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

27. International Business Machines Corp., nmon performance: Nigel's Monitor, http://www.ibm.com/developerworks/aix/library/au-analyze_aix/, accessed January 2011.

28. Izadi, S., Brignull, H., Rodden, T., Rogers, Y., and Underwood, M. "Dynamo: a public interactive surface supporting the cooperative sharing and exchange of media." *User Interface Software and Technology* (UIST '03), ACM 2006, 159-168.

29. Jiang, H., Wigdor, D., Forlines, C., Borkin, M., Kauffmann, J., and Shen, C. 2008. "LivOlay: interactive ad-hoc registration and overlapping of applications for collaborative visual exploration." In *Proceeding of the Twenty-Sixth Annual SIGCHI Conference on Human Factors in Computing Systems* (Florence, Italy, April 05 - 10, 2008). CHI '08. ACM, New York, NY, 1357-1360.

30. Johanson, B., Fox, A., and Winograd, T. 2002. "The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms." *IEEE Pervasive Computing* 1, 2 (Apr. 2002), 67-74.

31. Liu, Z. "Lacome: a Cross-Platform Multi-User Collaboration System for a Shared Large Display," Computer Science, University of British Columbia, 2007. http://hdl.handle.net/2429/378.

32. Microsoft Corporation, Kinect, http://www.xbox.com/en-US/kinect, accessed January 2011.

33. Microsoft Corporation, logman, http://technet.microsoft.com/en-us/library/bb490956.aspx, accessed January 2011.

34. Microsoft Corporation, .NET Framework http://www.microsoft.com/net/, accessed June 2010.

35. Microsoft Corporation. Network Projectors. Microsoft Corporation, http://msdn.microsoft.com/en-us/library/aa934598.aspx, accessed June 2010.

36. Microsoft Corporation, Silverlight, http://www.microsoft.com/silverlight/, accessed June 2010.

37. Microsoft Corporation, Windows Phone 7 Series, http://www.windowsphone7.com/, accessed June 2010.

38. Microsoft Corporation, Visual C#, http://msdn.microsoft.com/en-us/vcsharp/default.aspx, 2000.

39. Myers, B. A., "Using Handhelds and PCs Together," *Communications of the ACM*, 44(11), ACM (Nov 2001), pp 34-41.

40. Naef, M., Lamboray, E., Staadt, O., and Gross, M. 2003. "The blue-c distributed scene graph." In *Proceedings of the workshop on Virtual environments 2003* (EGVE '03). ACM, New York, NY, USA, 125-133.

41. Nichols, J., Myers, B. A., and Rothrock, B. 2006. "UNIFORM: automatically generating consistent remote control user interfaces." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada, April 22 - 27, 2006). R. Grinter, T. Rodden, P. Aoki, E. Cutrell, R. Jeffries, and G. Olson, Eds. CHI '06. ACM, New York, NY, 611-620.

42. Nintendo Co., Ltd., Wii, http://www.wii.com/, accessed January 2011.

43. Olsen, D. R., "Interacting in Chaos" *interactions*, ACM (1999), pp 42-54.

44. Olsen, D. R., Clement, J., and Pace, A. "Spilling: Expanding hand held interaction to touch table displays." In *Proceedings of TABLETOP '07*, IEEE Computer Society. Washington, DC, USA, 2007, pages 163–170.

45. Olsen, D. R., Hudson, S. E., Verratti, T., Heiner, J. M., and Phelps, M., “Implementing Interface Attachments Based on Surface Representations”, *Human Factors in Computing Systems (CHI ’99)*, ACM (1999), pp 191-198.

46. Olsen, D. R., Nielsen, S. T., and Parslow, D. “Join and capture: a model for nomadic interaction.” *User Interface Software and Technology* (UIST '01). ACM 2001, 131-140.

47. Oprea, A.; Balfanz, D.; Durfee, G.; Smetters, D.K., "Securing a remote terminal application with a mobile trusted device,” *Computer Security Applications Conference, 2004. 20th Annual* , vol., no., pp. 438-447, 6-10 Dec. 2004

48. Oracle Corporation, Java Documents on DataOutputStream, Oracle Corporation, http://download.oracle.com/javase/6/docs/api/java/io/DataOutputStream.html, accessed January 2011.

49. Oracle Corporation, Java Media Framework, Oracle Corporation, http://java.sun.com/javase/technologies/desktop/media/jmf/, accessed July 2010.

50. Oracle Corporation, OpenJDK, Oracle Corporation, http://openjdk.java.net/, accessed January 2011.

51. Paek, T., Agrawala, M., Basu, S., Drucker, S., Kristjansson, T., Logan, R., Toyama, K., and Wilson, A. “Toward universal mobile interaction for shared displays.” *Computer Supported Cooperative Work* (CSCW '04), ACM 2004, 266-269.

52. Perlin, K. and Fox, D. 1993. “Pad: an alternative approach to the computer interface.” In *Proceedings of the 20th Annual Conference on Computer Graphics and interactive Techniques* (Anaheim, CA, August 02 - 06, 1993). SIGGRAPH '93. ACM, New York, NY, 57-64

53. Petzold, C. Applications = Code + Markup: A Guide to the Microsoft Windows Presentation Foundation, Microsoft Press 2006.

54. Pierce, J. S., and Mahaney., H. E. “Opportunistic Annexing for Handheld Devices: Opportunities and Challenges.” In *Proceedings of HCIC* (HCIC ‘04) 2004.

55. RealVNC Ltd., RealVNC, http://realvnc.com/, accessed January 2011.

56. Richardson, T., Stafford-Fraser, Q., Wood, K. R., Hopper, A., "Virtual Network Computing," *IEEE Internet Computing*, Vol. 2, No. 1, 1998.

57. Scheifler, R. W. and Gettys, J. "The X Window System," *ACM Transactions on Graphics*, vol 5(2), (April 1986), pp 79-109.

58. Sharp, R., Madhavapeddy, A., Want, R., and Pering, T. 2008. "Enhancing web browsing security on public terminals using mobile composition." In *Proceeding of the 6th international Conference on Mobile Systems, Applications, and Services* (Breckenridge, CO, USA, June 17 - 20, 2008). MobiSys '08. ACM, New York, NY, 94-105.

59. Sharp, R., Scott, J., and Beresford, A. R. 2006, "Secure Mobile Computing via Public Terminals." In *Proceedings of the International Conference on Pervasive Computing* (PerCom 2006). IEEE Computer Society, 238-253.

60. Shen, C., Vernier, F. D., Forlines, C., and Ringel, R., "DiamondSpin: An Extensible Toolkit for Around-the-Table Interaction", *Human Factors in Computing Systems (CHI '04)*, ACM (2004), pp 167-174.

61. Schilit, B. N. and Sengupta, U. 2004. "Device Ensembles." *Computer* 37, 12 (Dec. 2004), 56-64.

62. Shuey, D.; Bailey, D.; Morrissey, T.P., "PHIGS: A Standard, Dynamic, Interactive Graphics Interface," *Computer Graphics and Applications, IEEE*, Vol. 6, No. 8, (Aug. 1986) pp 50-57.

63. Synergy, http://synergy-foss.org/, accessed January 2011.

64. Tarasewich, P., Gong, J., and Conlan, R. 2006. "Protecting private data in public." In *CHI '06 Extended Abstracts on Human Factors in Computing Systems* (CHI 2006). ACM Press, 1409-1414.

65. Tan, D. S., Meyers, B., and Czerwinski, M. 2004. "WinCuts: manipulating arbitrary window regions for more effective use of screen space." In *CHI '04 Extended Abstracts on Human Factors in Computing Systems* (Vienna, Austria, April 24 - 29, 2004). CHI '04. ACM, New York, NY, 1525-1528.

66. TightVNC Group, TightVNC, http://tightvnc.com/, accessed January 2011.

67. Thota, C. *Programming MapPoint in .NET*, O'Reilly Media, Inc., 2005.

68. Tritsch, B. Microsoft Windows Server 2003 Terminal Services, Microsoft Press 2003.

69. Want, R., Perins, T., Danneels, G., Kumar, M., Sundar, M., and J. Light. “The Personal Server: Changing the way we think about Ubiquitous Computing.” *Ubiquitous Computing* (UbiComp ‘02), Springer Verlag, (2002)

70. Yuan, F. Windows Graphics Programming: Win32 GDI and DirectDraw, Prentice Hall, 2000.

71. Yue, C. and Wang, H. 2009. “SessionMagnifier: a simple approach to secure and convenient kiosk browsing.” In *Proceedings of the 11th international Conference on Ubiquitous Computing* (Orlando, Florida, USA, September 30 - October 03, 2009). Ubicomp '09. ACM, New York, NY, 125-134.

Chapter 3 Privacy-Aware Shared UI Toolkit for Nomadic Environments

Arthur, Richard B. and Olsen, Dan R. Privacy-aware shared UI toolkit for nomadic environments. Software: Practice and Experience. May 2011. John Wiley & Sons, Ltd. DOI=10.1002/spe.1085

ABSTRACT

As computing becomes more nomadic, privacy becomes a greater concern. People use portable devices to annex displays in their environments so they can share information with other people. However, private information such as usernames, email addresses, and folder names are shown on foreign displays. Also, foreign keyboards can be used to enter in passwords generating a significant privacy and security risk. Because nomadic users' sensitive data is constantly at risk for exploitation via the UI toolkit, a solution for protecting user privacy must include that toolkit. This paper introduces the XICE framework—a windowing toolkit that provides easy display annexing and includes a robust privacy framework to help protect users and their data. This paper discusses the exploits that annexing external devices introduces and how XICE mitigates or eliminates those threats safely and naturally for both users and developers.

3.1 INTRODUCTION

Computing is increasingly nomadic. Nomadic computing allows people to collaborate with each other in a wide variety of places and situations. A key feature for a nomadic user is the ability to have his own data, settings, and applications available wherever he is located.

A user could carry his data, settings, and applications via a *personal device* (e.g. laptop) so that he can interact with his data anywhere. However, portable devices tend to have constraints on size and weight and consequently on processing power and interactive richness. If

the user can use his personal device to annex resources (e.g. projectors or desktop monitors) in his environment then he can overcome many of the limits of his personal device.

Normally annexation is performed via a VGA or DVI cable. Digital connections via wireless networks are more versatile, because the annexed devices may additionally supply input hardware and support multiple users. This paper contends that such network devices will replace VGA or DVI. The network-enabled, annexable computers are called *shared devices* and typically provide a screen but may also supply input devices such as keyboards and mice. Some of the common devices users might share are illustrated in Figure 3:1. Notice that some shared devices are public and not trusted while some shared devices are private and trusted. User Interface (UI) software must be able to tell the difference and present itself accordingly.

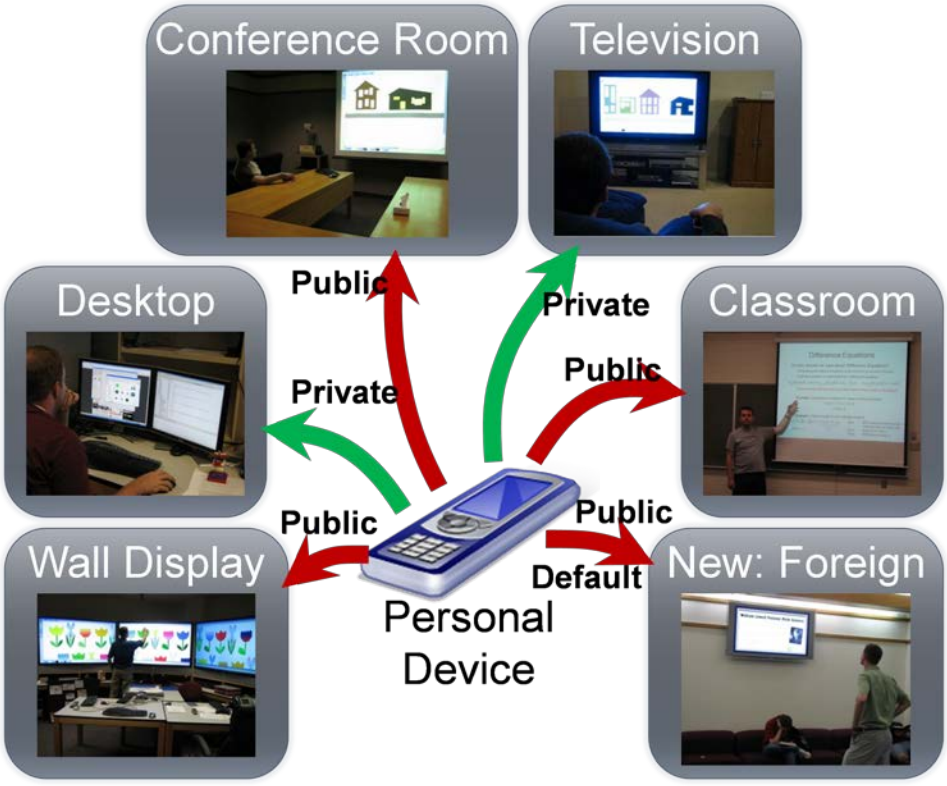


Figure 3:1 – Shared devices that a nomadic user could annex. These devices are located in either public or private environments.

Key to annexing shared devices is distributing the UI from an application executing on the user's personal device to a shared device. There are several existing technologies that may be used to distribute a UI, such as X-Windows (X11) [1], Remote Desktop Protocol (RDP) [2], or Virtual Network Computing (VNC) [3]. These protocols transmit the output from the executing computer to the shared device (sometimes called a *display server*) and transmit input from the shared device back to the executing computer.

This network UI architecture introduces several privacy-related challenges. If a user annexes a shared device, that device has several potential attack vectors: stolen output, stolen input, false output, and false input. The shared device may have *crimeware* [4] installed, which could steal the output from the user's applications including any sensitive data shown in those applications (e.g. email addresses or login names). If the user enters his username and password through the annexed input hardware, the shared device may steal those credentials. In addition, the shared device may falsify output or input in an attempt to compromise the user's device or expose his sensitive information.

Fortunately, because the user has a portable trusted personal device, there are some new privacy-related opportunities. The personal device may be used to view and input *sensitive data* (any data which should not be viewed by arbitrary users), while the shared device may be used to display *open data* (data that can be viewed by anyone). Most current privacy-aware software operates exclusively on one screen which it must treat as a *public screen* (viewable by anyone) or *private screen* (viewable by trusted parties). Allowing software to simultaneously operate on a public and a private screen opens many new issues for privacy-aware user interface tools.

In addition, the input from a shared device may be trusted or distrusted. *Trusted input* is input that the user (or someone the user trusts) has control over to prevent malware from stealing

input or providing false input. *Distrusted input* is input from some foreign device which the user does not control. The term ‘distrusted input’ may synonymously be considered ‘untrusted input.’ This paper uses the term ‘distrusted input’ throughout.

Each device a user interacts with has a different *privacy state*—a combination of whether the display is public or private and whether the input is trusted or distrusted. With nomadic interaction, the portable device is assumed to be private and trusted. The user’s desktop machine at work or home would be a shared device that is private and trusted. However, annexing a coworker’s desktop display may be public and trusted; the user does not want IMs to appear on the coworker’s display. In a conference room at the user’s work the display is public, so sensitive data should not be shown, but any other open data should be shown. Because the shared device is controlled by the user’s work, its input is trusted so the user can confidently access or expose sensitive data as necessary. When presenting at another institution, however, the shared device is public and distrusted, so sensitive data should not be shown or altered. Shared devices in other locations such as restaurants or mall kiosks would also be public and distrusted.

Even though input may be distrusted the user is not precluded from accepting input from the annexed device. For instance, a user may choose to annex a foreign display so he can type up an email consisting of open data because typing via a physical keyboard is faster than via his mobile device’s soft keyboard. If a user chooses to accept input from a distrusted device then his software must actively protect him from potentially malicious activity. For example, the software must be aware of any input that may attempt to alter or expose sensitive data. Continuing with the email example, the user should pick the recipients via his mobile device, and the email addresses should be blocked on the public display (although the contact name may be shown). A malicious display server could attempt to expose the user’s contact list on the public display so

that it could steal the available email addresses. The contact list exposure instruction should be confirmed in a non-exploitable way—for example via a dialog on the personal device—so that the email application knows the user initiated the contact list exposure. A worse violation would be if a malicious display could generate and send spam emails via the user's email application. Consequently, the "send email" command should similarly be confirmed.

To enable privacy-aware development, application developers must be able to write software which changes its output and can filter any input based on the shared device's privacy state. X11 and RDP inform software when the UI is rendered on a shared device, while VNC does not. However, none of these protocols attaches a privacy state to the network connection; all shared devices are treated as private and trusted. RDP allows for excluding the shared device's input, but accepting and distrusting input is not possible. A shared UI protocol must be able to identify the privacy state of a shared device so shared applications may alter their output and filter the input.

When using existing UI distribution technologies, developers have two major problems: tracking privacy state and augmenting an application's UI. Developers must independently add code to identify and track the privacy state of the shared device. Additionally, developers must make many privacy-related decisions with regard to application output. Without rigorous software development standards, these decisions may not be implemented consistently, leading to inadvertent privacy leaks.

An application-specific privacy state creates an inconsistent interface for users. If a user has more than one privacy-aware application executing on his personal device, then having different privacy management tools in each application can lead to more privacy leaks (e.g. the

user changes the privacy state of one application but neglects to change the privacy state for the other applications).

This paper introduces the XICE Windowing Toolkit (eXtending Interactive Computing Everywhere pronounced “zice”) [5]. XICE has been built to allow any network-connected device to annex (via Wi-Fi, Bluetooth, etc.) interactive resources (e.g. screens, keyboards or mice) on other network-connected devices. XICE includes a straightforward, robust privacy framework that allows a user to specify whether a display/environment is public or private and whether he trusts its input hardware. The privacy framework also allows developers to write code that takes advantage of the user’s privacy specifications. Providing a user-friendly solution for securing sensitive data in nomadic environments requires an interactive, understandable model and a UI toolkit that simplifies privacy-aware UI development.

This paper deals exclusively with the privacy issues inherent in a shared UI windowing toolkit, and not the broader privacy problem of cryptography, secure protocols, or online website privacy settings (e.g. facebook [6] or MySpace [7]). This paper and the XICE toolkit do not address these issues. This paper does introduce a toolkit which helps protect the user and software from potential exploits inherent in sharing a window to a foreign display server.

Although the connection to a display server could be encrypted (e.g. using public/private key-pair encryption) XICE does not enforce this. Instead XICE assumes that any data transmitted out of the personal device is stolen, regardless of whether the display server or environment is trustworthy. For this reason, only UI output is sent to the display server and never any data files, and XICE’s privacy framework is intended only to affect application output.

3.2 Privacy Threat Analysis

As people work nomadically, they interact with foreign devices. To protect sensitive data, the safest choice is to never annex devices belonging to others. However, users are then limited to resources on their personal devices. People should be able to use external devices, even suspect ones, in ways that protect their data. Malicious machines can take advantage of users in four ways: stolen output, stolen input, false input, and false output. These exploits, plus the potential for public embarrassment, comprise the *five key problems* inherent in nomadic computing.

3.2.1 Stolen Output

Any information shown on a shared device can be stolen by that device. At worst, a malicious device could steal data like user credentials. For example, if an email application has trouble authenticating with its email server, the application may show a dialog similar to the one in Figure 3:2. This dialog shows the user's email address, email server, and password length. The shared device now has enough information to perform a brute-force attack on the user's email account. An annoying, but less injurious consequence might be that the shared device could scrape the email address and spam it.

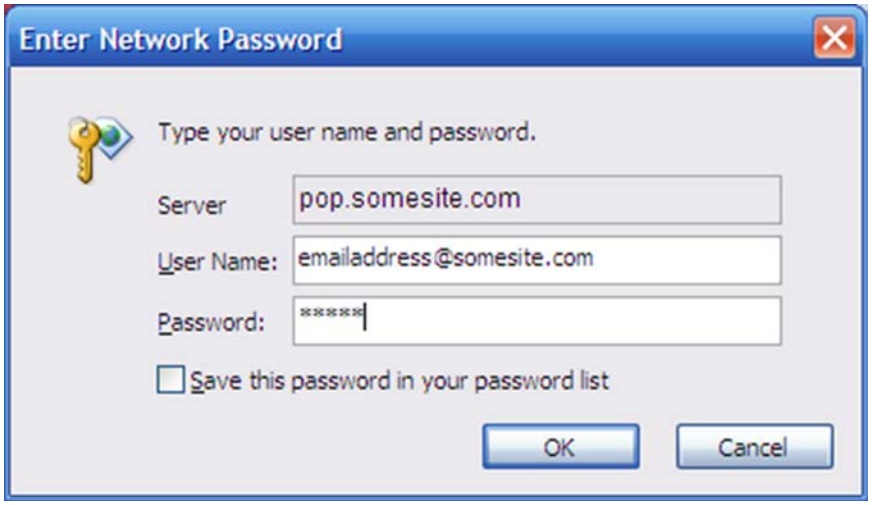


Figure 3:2 – An inappropriate dialog for a public display—it shows private information. The display now has the user's email server, email address, and password length.

To protect against stolen output, sensitive data should not be shown on a public screen. At minimum, if a window that typically contains sensitive data is shown on a public screen, the sensitive data it contains should not be transmitted to the shared device. It is better if that sensitive data is redirected to a private device.

To facilitate protecting sensitive data, the UI toolkit will need to identify public screens and provide that identification to applications. Each screen must be tagged either public or private. Applications can then use that state to prevent sensitive data from showing on public screens.

Identifying a public screen must be within the user's control. The shared device cannot specify the privacy state because the shared device could easily provide a false privacy state. Instead, the private state must be specified through the personal device, preferably by the user.

3.2.2 Stolen Input

If a shared device is used to provide input, the device can steal that input. For instance, many people use internet cafes or hotel computers to check email. These machines could easily harbor key loggers that steal usernames and passwords.

If an application is unaware that the input is distrusted, the application will request input through the shared device, regardless of whether sensitive data is included. For instance, if a web browser encounters a page that has login boxes, those boxes will show on the shared display. Then, users will likely enter their credentials directly via the shared device's input hardware, compromising sensitive data. Applications must be aware of public displays and any widgets that request sensitive data (e.g. password boxes). Requests on public devices for sensitive data may then be denied. The sensitive data request and input binding can be moved to the secure personal device. The user must be notified via the public device that sensitive data is requested on the personal device. Users need software to intuitively guide them with privacy and security [8].

3.2.3 False Input

A shared device can control any application without the user's consent. For example, if a shared device has malicious software that is familiar with a user's application, the shared device could expose sensitive data by sending input to the application's window causing changes to the user's privacy settings. The "Show Private Data" menu option in Figure 3:3 causes an application to expose all of the sensitive data within the window. To exploit this menu option, the shared device could send a mouse click event to expose the menu, and then another click to "Show Private Data".

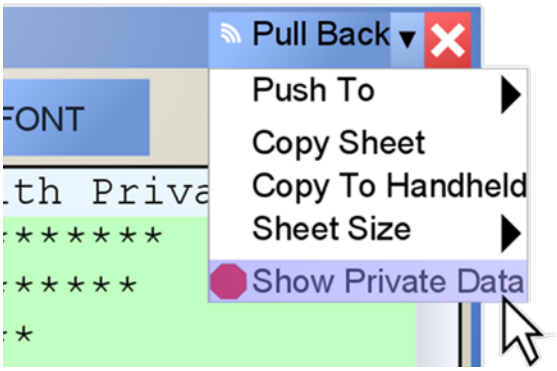


Figure 3:3 – Options such as "Show Private Data" can be exploited by a display that falsifies input.

One way to prevent the shared device from supplying malicious input is to categorically deny the device's input. However, input from a shared device could be much richer than input on a personal device. For example, a physical QWERTY keyboard on a shared display is much easier to work with than the soft keyboard on a personal device. The user must be able to use the input on the shared device if he chooses.

Another way to prevent false input is to remove all widgets that affect sensitive data from a window. However, such prevention limits what the user can do. For instance, a user may trust the environment for some of his sensitive data. When the "Show Private Data" option (Figure 3:3) is attached to the window it affects, the user knows which window he is exposing when he selects that option, because he is exposing the window the menu option is attached to. If that option is not shown on the public display then there must be an abstraction on the personal device that he can use to expose that window. The user may then inadvertently use the wrong widget and expose the wrong window.

A better way to prevent false input is to securely confirm sensitive input. For example, if the user clicks on the "Show Private Data" option, a dialog could be shown on the personal device to confirm that selection. Through the trusted input on the personal device, the confirmation dialog ensures that the user wants to expose a particular window's sensitive data. If

the display supplied malicious input to try and expose that window, the user would be alerted to that request by the unexpected appearance of that confirmation dialog on the personal device. By confirming actions that affect sensitive data, the display is discouraged from supplying false input.

3.2.4 False Output

Shared devices can overtly change a window's output in an effort to conceal malicious activity or to coax users into exposing sensitive data. As an example of concealing malicious activity, if no confirmation dialog is provided in response to false input, the malicious display could click the "Show Private Data" option, and then immediately send mouse clicks to undo that option. The device may only present the last concealed version of the window so the user is unaware that his privacy has been compromised.

Providing convincing false output to coax the user into exposing sensitive data is difficult, but a malicious display may resort to such behavior when the user does not allow input from the shared device. For example, when the user provides all input from his personal device, the display only knows when the mouse moves or when the UI changes. If the malicious device accurately predicts a user's click on the "Copy Sheet" option in Figure 3:3, the shared device could swap the graphical output of that option with the "Show Private Data" option. The user sees an altered view (on the right of Figure 3:4). The personal device will interpret the user's click as "Show Private Data," because the personal device has a model of the graphical output sent to the shared device (on the left of Figure 3:4). The personal device automatically trusts this input from itself; consequently, the shared device tricks the user into inadvertently exposing sensitive data.

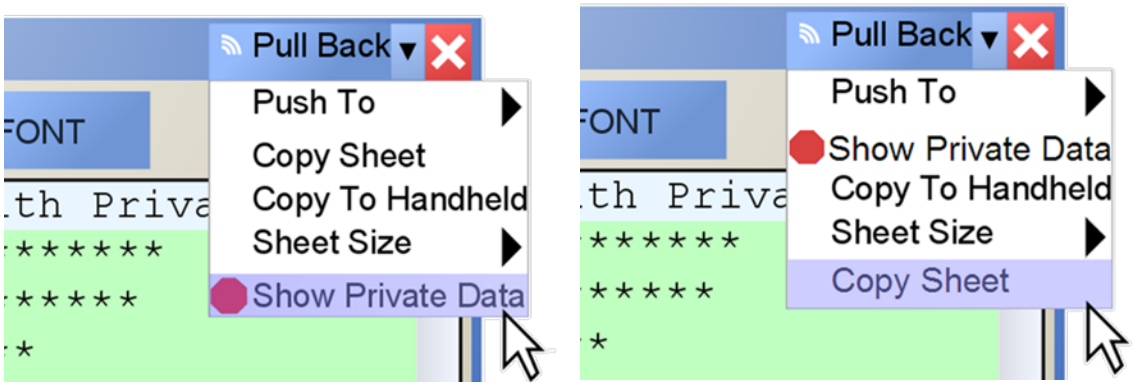


Figure 3:4 – A shared device could swap the "Copy Sheet" and "Show Private Data" options. The image on the left is what the personal device transmits to the shared device. The image on the right is what the shared device shows.

False output cannot be prevented, but it can be detected. If a user performs an action that affects sensitive data, that action should be confirmed on the personal device. Confirmation prevents exposure of sensitive data to a public device and enables the user to discover the malicious display. Since the user does not expect a confirmation dialog, its appearance alerts him to the display's malicious activity.

3.2.5 Embarrassment

In collaborative situations, users bring data to share with other people using shared displays. Even if sensitive data—as identified to applications—are protected, *embarrassing data*—relating to the social appearance of the user—may be exposed. Disclosure of either data type is unacceptable. Consequently, for the remainder of the paper both are considered sensitive.

One situation in which a user's embarassing data may be exposed is when he receives an instant message on a shared display. Such messages are typically benign, but a message like the one in Figure 3:5 might be embarrassing. If the icon in that message were offensive, the social situation could be disastrous.

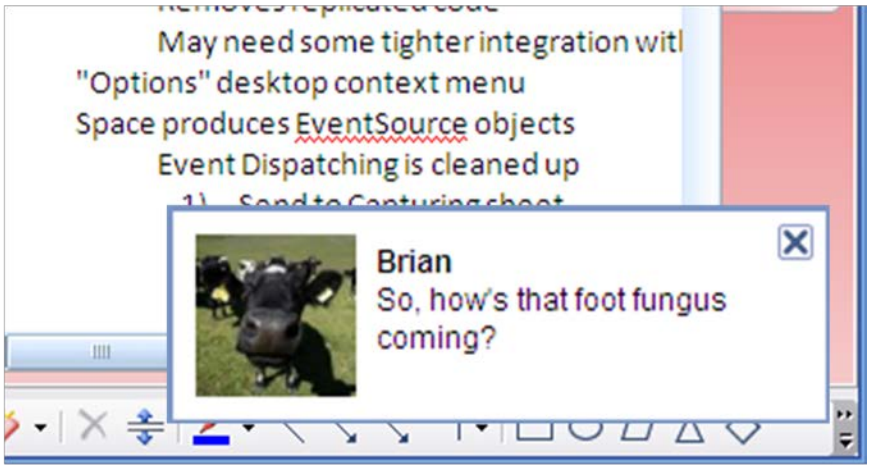


Figure 3:5 – An embarrassing Instant Message shown during a collaborative session.

Also, many email applications show notifications of an email's arrival. The process through which these notifications appear is similar to instant messages.

Instant messaging clients are somewhat intelligent with regard to new messages; the latest versions of these clients will not show notifications when the user's active window is full-screen. Not showing notifications is useful when he plays a full-screen game or gives a presentation. But, if he demonstrates software in a typical window, the shared display is treated as private and the notifications appear. Another indication of the flaw of using window state to determine privacy state is when the user views a full-screen application on a private display (such as at his desktop), the notifications do not appear. The software misinterprets the full-screen application as a public situation whereas the user is in a private situation where he wants to receive notifications. However, if applications are aware of the user-specified privacy state, the applications can intelligently choose when and where to show notifications.

When a user opens files, he sees dialogs like the one in Figure 3:6. If he makes a presentation to IBM, he might not want IBM employees to notice presentations for Intel, Apple, and Microsoft.

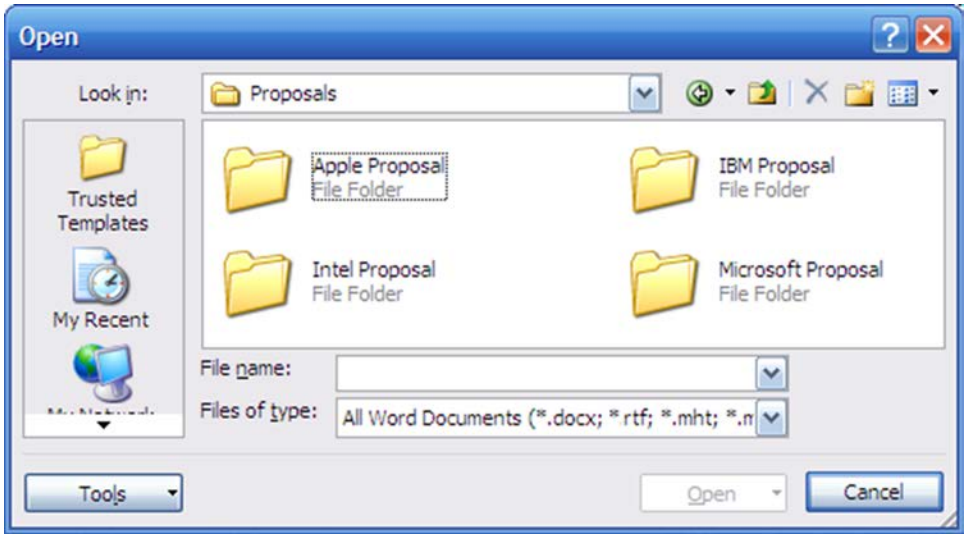


Figure 3:6 – A file dialog with sensitive folder names.

To prevent embarrassment, potentially sensitive data should not be shown on public devices. Privacy-aware applications can make appropriate privacy decisions when using public screens instead of assuming like instant message applications do. When annexing a public screen, applications should show potentially sensitive data on the personal device. When annexing a private screen such as one's own desktop, applications should show sensitive data on that private screen.

3.3 Solution Requirements

The authors envision new environments and styles of interaction that will emerge as users become more nomadic and computers change to facilitate such users. The core challenge for nomadic computing is to carry a portable computer but annex shared devices and safely gain significant resources. An extreme, but potentially common, situation is a user carrying a handheld device to execute his software, but interacting with that software on a much larger screen (e.g. wall-sized). The annexed resources must provide more screen space or input than the user is carrying; otherwise the user is unlikely to annex those resources because they do not offer any additional power.

The user should have consistent personal settings on any annexed machine. This consistency means that his personal settings and applications should be available wherever he is. The user must be able to carry his data, settings, and applications with him so that these pieces are uniform everywhere.

Furnishing data, settings, and applications to an annexed machine exposes the user to the five key problems. The data, settings, and applications should be available without being exposed, which requires trusted processing wherever the user is located. A personal device should process the user's data, settings, and applications.

When annexing foreign devices, users may frequently interact with sensitive data. Rather than exposing sensitive data on public screens, users need privacy-aware software to help protect their sensitive data. Users could restrict themselves to web-based applications and use kiosks for all computing needs. However, those who interact exclusively with kiosks are exposed to the five key problems. For example, a user must supply his username and password to the kiosk each time he visits a personalized website. Not only can the kiosk steal those credentials, it can swipe sensitive data (e.g. email addresses) on those personalized sites. A user needs a trusted device through which he can safely view and enter sensitive information, regardless of whether the environment is private or trustworthy.

A privacy-aware application is more capable of protecting the user's sensitive data when it is informed of the privacy state for a shared device. The application must be able to show non-sensitive data on a public screen and simultaneously show sensitive data on the personal device's screen. Splitting the UI between the annexed device and the personal device has the potential to considerably enhance the user experience and greatly reduce the attack surface a shared device may exploit, because malicious shared devices can only steal sensitive data from application

outputs rather than from data within the user's files. Applications with sensitive data which are aware of public screens can better protect the user's sensitive data.

The privacy behavior should be consistent and universal through all applications on the user's device. An inconsistent experience creates more privacy failures and user frustration. The Mac [9] has shown that to gain consistency across applications the toolkit must provide the consistency rather than relying on end developers. Thus, to gain consistent privacy-aware software, the privacy-aware decisions must be moved to the windowing toolkit as much as reasonably possible.

The nature of the nomadic computing environment necessitates windowing toolkits and applications that are privacy-aware. This must be dealt with before deploying a Dynamo- [10] or Personal-Server-like [11] environment. With a privacy-aware shared UI toolkit and applications, the user's software becomes aware of the privacy state he applies to his current environment and the software can augment itself accordingly.

3.3.1 Coding Privacy

The granularity at which to integrate shared UI privacy into an application is an application-specific decision. Different applications and users will have varying needs for protection. For example, a word processor may treat entire documents as sensitive or open, while a spreadsheet may treat individual cells as sensitive or open.

The best granularity for privacy is unknown. There is a continual tradeoff between privacy violation and over-protection. Reducing barriers makes software easier to use and more transparent, but also increases the opportunity for violations and surprised and angry users. Creating more barriers increases the protection for users, but frequently makes users frustrated as they must navigate these barriers and make decisions. Developers need the freedom to

experiment with their software designs so they can create the best software for their users. As is shown in this section, the question of when to prevent sensitive data from appearing on an annexed screen is broad and has numerous potential solutions.

Currently, there is little to no windowing-toolkit-wide shared UI privacy protection available, so applications must make these decisions independently. However, with a good toolkit which provides system-wide privacy information, developers have greater freedom to explore the privacy-aware shared UI software space.

What follows are five examples of privacy granulation issues. This section discusses each option and proposes several theoretical approaches. None of these approaches has been tested, and each approach has varying levels of difficulty for both users and developers. The purpose of this section (3.3.1) is to show why it is necessary to have a windowing toolkit which makes these options possible.

3.3.1.1 Word Processor

A word processor could have privacy integrated at different levels within the application, from fine-grained to coarse-grained. A word processor could be designed to protect individual characters, words, sentences, paragraphs, pages, or even the entire document. Embedded images may require protection as well, which could be at the image level (block the whole image) or pixel level (block out someone's face using a rectangular or irregular region).

Somehow the sensitive data must be specified. With finer-grained protection, the software may require the user to specify each piece of sensitive data. For example, the user must highlight text and mark it "sensitive." On the other hand, developers may produce software which automatically detects and protects potentially sensitive information. For example, it may discover and protect email addresses, phone numbers, social security numbers (SSN), and

financial information. The sensitive data would not appear on a public screen unless specified by the user.

On the other hand, maybe it is easier for the user to specify what data is open rather than what data is sensitive. A developer may create a word processor which protects all data by default, unless the user specifies otherwise. However the user specifies the open data, the sensitive data must be protected from appearing on public displays.

When the software protects characters, words, or images, it must provide feedback to the user that such information is protected. The software may highlight that information in some way while the user is working in a private environment.

When the user moves to a public environment, the sensitive data must be protected while potentially giving feedback about its existence. The word processing application may hide all sensitive data and not show that it even exists. Or the software may show gray rectangles where the sensitive data is located. “Greeking” sensitive text may be preferred; in such cases “lorem ipsum” text may be inserted. A more radical case may be to collapse the sensitive data and show a note down the side of the page pointing at it.

3.3.1.2 Application-Independent Privacy Awareness

There are many widgets that have privacy issues independently of the content. For example, showing the file dialog in Figure 3:6 on a public display is unacceptable. Content-assist menus which show recently-used words may need to be blocked. Menus within the application that show the names of recently accessed documents may need protecting. Launching an email application to send a document may be more appropriate on a private display. Clearly some dialogs, menus, menu items, or even applications need to be protected.

Properly protecting the user's privacy requires that the user, developer, or software detect *privacy boundaries*—clear differentiation between open data and sensitive data. For example, specific data within a document may be considered sensitive, so the boundary between the surrounding open data and the sensitive data must be respected. Another clear privacy boundary is shown in the following example. Two people are discussing a document on a shared display space. When the user who owns the document opens the file dialog to choose another document to view, that user has crossed a privacy boundary. He thinks about the file system differently from his word processor, and the ways of protecting the file system are different from protecting data in the word processor. Detecting such boundaries in the UI design process and then protecting them in the implementation is important.

3.3.1.3 Spreadsheet

A spreadsheet has many of the same privacy-related design decisions as the word processor. Protection could be performed on the level of characters within cells, cells themselves, rows and columns, or even the entire document. For the user, marking individual rows or columns may be less tedious than marking individual characters, but the user may find any marking tedious. Developers may add code which automatically protects data based on the format. For example, it could protect all financial information, email addresses, and phone numbers from appearing.

Some shared UI privacy-related decisions are different than a word processor. For example, a user may not want to share individual employee salaries, but may be willing to share aggregate personnel costs. A developer may write an algorithm that treats financial information as sensitive, except in cases where an aggregation (e.g. sum or average) is used on some minimum number of items (e.g. 5 or more). Graphs may also be treated as sensitive if they

express data from cells marked sensitive, unless the data is sufficiently abstracted from the original values. Clearly these decisions are not the same as the decisions for a word processor application.

Whether each of these spreadsheet-application-specific ideas is good or bad or even appropriate is unknown. But a good toolkit will allow developers and users to explore these decisions themselves.

Other similar issues are how to indicate at a private display that sensitive data is protected if shown when connected to a public display and protecting some dialogs, menus, menu items, and applications.

3.3.1.4 Email Manager

An email application is likely to contain sensitive data especially because the data is generated by other users. Expecting other users to properly tag data as sensitive may be foolhardy. But requiring a receiving user to individually tag information within each email message will likely frustrate him. More proactive protection tools must be necessary. For example, the email application may actively prevent the email list and individual emails from showing on public displays.

Email is frequently shared with other users. For example, in a collaborative situation users might show email messages to each other. The email application itself will appear on the personal device, and the user can elect to share individual emails.

When an email is shown on a shared device, email addresses in the ‘To’ and ‘From’ fields represent another privacy boundary and should not be sent to the shared device, otherwise the addresses could be stolen. Two new situations arise when sensitive data is blocked from appearing on a public device. First, the user may want to privately access those email addresses;

this scenario is called the *reviewing situation*, because the user may need to review sensitive data without exposing it on public screens. Second, the user may want to explicitly show those email addresses anyway because he trusts the audience; this scenario is the *field override situation*.

3.3.1.5 Web Browser

A web browser offers some interesting privacy-related challenges. One such case is when a user visits his bank's website. He does not want his sensitive data to show on a public display. Or consider a user who visits a news site which requires a user id and password to view articles. The pages within the site may be considered open while the login page should be sensitive.

Web standards may change so that sites can specify pages or content that is sensitive, but until that point browsers have several approaches for privacy-aware shared UI designs. These approaches include protecting data at the window level, the page level, widget level, or at the content level.

The browser may protect a specific window the user has created, regardless of the tabs shown within that window. Consequently, when the user shares that window on a public screen the content shown in each of that window's tabs are also shared. If the user chooses to share a window containing a sensitive page, the URL for that page may be blocked from appearing in the address bar so that others in the room have a more difficult time finding and accessing that site.

Alternatively, the browser may protect individual pages from showing, requiring the user to explicitly authorize each page for public screens. Or the browser may be designed so that the user must explicitly authorize specific domains, sub-domains (e.g. for blogs, or sub-sites that may contain sensitive information), and possibly URL folders. As the user navigates away from an authorized domain/sub-domain/folder, the user must re-authorize. If the user logs in to a site,

the rest of that site may be considered sensitive unless specified otherwise. A site that is accessed through HTTP may be considered open while a site that is accessed through HTTPS is considered sensitive. Or developers may come up with a more manageable page-level privacy boundary.

Content within the page may need to be protected. For example, login boxes on a web page should probably not be available on a public display, but still available to the user (a variant of the reviewing situation). After logging in, easily-identified sensitive data (e.g. email addresses) within a page may be discovered and protected by the browser, even though the rest of the page is considered open.

Assistance from the browser may also contain sensitive information. A privacy-aware browser may be designed to hide content-assist drop-downs, recent history, or bookmarks, while still showing most menus. Or, if the user marks certain pages or sites as sensitive, those specific pages may be blocked from appearing in content-assist drop-downs, the history, and bookmarks. There are clearly a wide variety of shared UI privacy issues that affect browsing and the features expressed by the browser.

3.3.1.6 Instant Messenger

As discussed in section 3.2.5, instant messaging software can present several problems. It is unlikely that users will tag individual messages or content within the messages as sensitive. So the software must treat all such messages as potentially sensitive. When the user is at a private screen all his messages should appear, but when in a public environment his messages should not appear publicly.

Similarly, the contact list should not be shown on a public display unless the user explicitly chooses to share that list. On the other hand, the user may choose to share a specific conversation thread to a public screen.

3.3.2 XICE Privacy-Aware Strategy

Specifying where privacy boundaries should be placed is a difficult problem. Clearly the shared UI privacy issues in a word processor are not the same as the issues in a spreadsheet, email manager, web browser, or instant messenger. Each of these tools should have consistent mechanisms for allowing users to share a window, protect sensitive data, review sensitive data, and show sensitive data anyway. Therefore, the XICE toolkit is designed to incorporate many privacy-aware shared UI decisions for both users and developers.

The XICE toolkit provides important facets for privacy-aware software design. One facet is that the toolkit provides information to applications about how the user feels about an annexed display space (the privacy state). Another is that the toolkit provides a set of tools available to developers from public/private windows to fine-grained widgets that can help the developer protect the user's sensitive data on public screens and from distrusted input.

Critical to implementing effective privacy-aware applications is lowering the overhead required for developers to implement privacy awareness. If developers make fewer decisions when implementing privacy awareness, but can more consistently achieve their desired results, then users will also have a better experience. Currently, developers must start from scratch when building privacy-aware applications, so moving many decisions into the toolkit is beneficial.

Considering that no such windowing toolkit exists (as is shown in section 3.4) the XICE toolkit is compared against the current state of no options for developers (other than build-from-scratch), no information for applications (no system-wide privacy state), and no control for users.

For example, when writing an email application, the developer may want to declare new emails as sensitive and show them only on a private display. Previously these decisions were difficult but with the XICE framework they are straightforward.

In summary, the overall protection of nomadic user privacy has *five parts* that the toolkit and applications must implement. First, the toolkit must be able to distribute the UI between the personal device and the display, allowing the application to independently render to both displays, and giving the user a consistent, richer experience. Second, the toolkit must allow the user to identify an annexed display's privacy state, and the toolkit must inform applications of that privacy state. Third, applications must avoid showing sensitive data on public devices, while still showing open data. Fourth, applications should either not allow sensitive input from distrusted devices or confirm such input on the personal device. Finally, users must be able to resolve the reviewing situation and field override situations by explicitly overriding privacy controls.

3.4 Prior work

The solution for protecting a nomadic user's privacy must incorporate the five parts enumerated in the preceding section. A wide variety of privacy-aware applications have been developed, each with novel aspects in how they protect sensitive data, but few are implemented in the toolkit or incorporate UI distribution. A generalized solution that can be used simultaneously by a wide variety of applications is preferred.

Examples of privacy-aware applications include PrivateBits [12] and Privacy Blinders [13]. PrivateBits has novel tools for blocking sensitive data contained in a web browser. For example, PrivateBits filters the browsing history and auto-complete hints when the browser is in public mode. Privacy Blinders renders moveable black rectangles over the top of sensitive

information within a web page. The source web server tagged that information as sensitive. But neither approach incorporates UI distribution to shared devices, so input cannot be controlled, nor can they resolve the reviewing or field override situations. If the user were to use X11 or RDP to annex shared devices, Privacy Blinders fails to prevent sending sensitive data to the shared device. Because the black rectangles are rendered after rendering the sensitive data, the user sees the information as being blocked but the display server sees the rendering calls for the sensitive data.

Berry, Bartam, and Booth (BBB) [14] have developed an approach that uses simple UI distribution. Their tool integrates with Microsoft Word, Excel, and Internet Explorer and exposes a replicated, public view via VGA. Users tag information as sensitive using the application-provided highlighter tool, and then the public view blocks that sensitive data. In addition, file dialogs are prevented from showing on the public display. However, the public view only annexes as much visual space as the personal device has. Consequently, the shared device provides no additional screen space. The applications are also not informed of the privacy state and cannot take action to protect the user's sensitive data.

Symbiotic Displays [15] is a wrist-watch application that can annex a display to show emails. This approach effectively distributes the UI across two displays with different privacy states. It also supplies an interesting resolution to the reviewing situation by allowing the user to select sensitive words (which are blurred) on the shared display and view the words on the wrist-watch's screen. Unfortunately, although other people in the room cannot see any sensitive data, all of the sensitive data is transmitted to the shared device. The shared device is the computer that makes a decision as to how to block the sensitive data. Consequently, this approach is still vulnerable to stolen and false output and input.

Oprea et al. have a solution for safely interacting with personal data on a shared device [16]. The personal data is stored and processed on a remote computer, and the personal device establishes a connection between the shared device and the remote computer, as illustrated in Figure 3:7. The personal device is augmented with an optical mouse to supply all pointing input. Unfortunately, this solution only protects against false input and some stolen input (users may still use the shared device's keyboard to enter credentials), but not stolen output or embarrassment. In addition, because it relies on the Internet and VNC, displays without Internet access cannot be annexed and large distances slow the interactive experience.

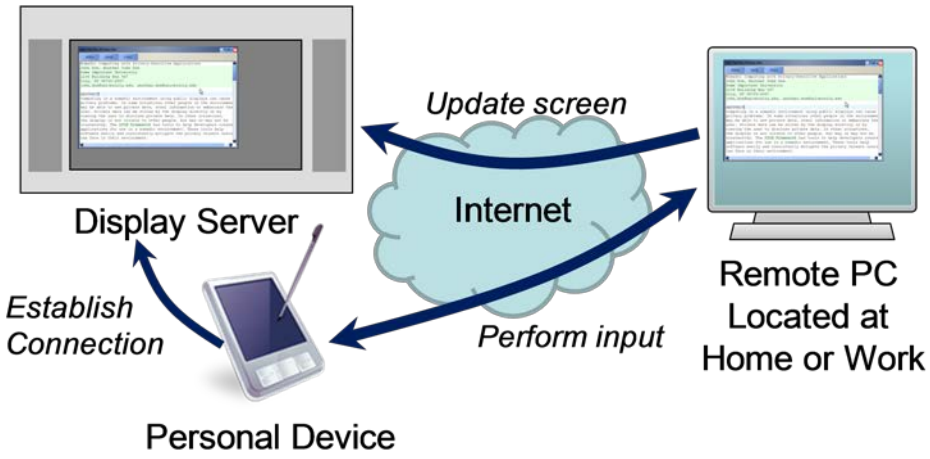


Figure 3:7 – Oprea's annexing solution. The personal device brokers a connection between a VNC display server and a remote PC, and then supplies all pointing input.

Sharp et al. provide a similar solution to Oprea, except that Sharp protects against stolen output [17]. Oprea shows everything the remote computer renders while Sharp blurs all text before transmitting the UI to the shared device, including all non-sensitive text. To view any of the rendered text, the personal device shows a complete, scrollable, un-blurred view of what the remote machine renders. However, most text within an application is not sensitive (open) and is useful if shown on the shared device. In this case, applications are not privacy-aware so the toolkit cannot intelligently block only sensitive data.

Mobile Composition [4] provides a web-based privacy solution that splits web pages across two machines. The shared device shows all of the user's open data, but areas of a page containing sensitive data are tagged as sensitive by the website designers and are encrypted such that only the personal device may decrypt them. In this way, Mobile Composition protects against the four problems of stolen and false output and input. However, this solution does not support applications outside of the web and relies on the web for all processing, which prevents the user from interacting with software when Internet access is unavailable.

SessionMagnifier [18] takes a similar approach to Mobile Composition, except that SessionMagnifier uses the PDA as a proxy web server for the shared display. Then SessionMagnifier can filter information on a web page. However, this approach suffers from the same problem as Mobile Composition—SessionMagnifier only operates with web applications.

3.5 A Privacy-Aware UI Toolkit

Nomadic computing necessitates UI distribution from personal devices to shared devices. For the goals outlined in this paper, a significant failing of most existing privacy-aware technologies is that they do not distribute the UI from a personal device to a shared device.

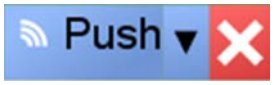
XICE was developed using Java 1.6 to create a seamless nomadic computing environment. XICE handles the details of annexing shared devices and distributing application UIs to them. The graphical output of each application window is abstracted as a scene-graph (a tree-structure display list) so that windows can be rendered on any shared device without regard to display size or resolution. A scene-graph is sometimes called a presentation tree. Scene-graphs have been studied for several decades in tools such as Graphics Kernel System (GKS) [19] or Programmer's Hierarchical Interactive Graphics System (PHIGS) [20]. Rather than using the traditional damage-repaint rendering model an application builds a tree structure of drawing

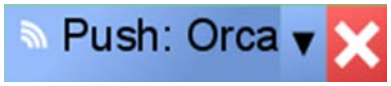
commands and hands that tree over to a rendering framework which renders the tree independently of the application. Toolkits such as Silverlight [21], Windows Presentation Foundation (WPF) [22], and JavaFX [23] are popularizing scene-graphs as integral components in GUI development for desktop applications. As measured in prior research, the scene-graph also dramatically reduces the computational burden on the personal device relative to RDP, VNC, and X11, making the protocol more suitable for small personal devices [5]. Via XICE, users can annex shared devices in airports, restaurants, personal offices, living rooms, etc., and push windows to them. In addition, XICE has the appropriate hooks to make privacy a first-class citizen.

3.5.1 Windowing Toolkit Interaction

Users working nomadically take personal devices to shared devices and annex them. The personal devices run XICE, and the shared devices have the hardware and software necessary to establish network connections and to process XICE rendering commands. Users can push application windows to the shared devices.

The windowing toolkit must enable users to quickly push windows to an annexed display. In the top-right of every window is a drop-down button which provides access to the toolkit options for that window. This button takes on different appearances—some of which are shown in Figure 3:8—depending on the state of the window. Windows on the personal device show the “Push” option, which, when clicked, starts the process of connecting to a shared device.

(a) 

(b) 

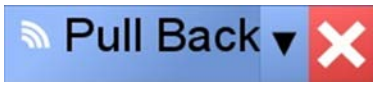
(c) 

Figure 3:8 – Common decorator widgets. The X closes the window, while (a) initiates a connection to a shared device, (b) pushes a window to a currently connected shared device, and (c) pulls back a window shown on a shared device.

After annexing a shared device, all other windows on the personal device change to "Push: *Machine*". All shared windows say "Pull Back" to provide a single click for removing that window from the shared device. Applications continue to execute on the personal device; only the graphical output is transmitted to the shared display.

After a user clicks the "Push" button, the connection dialog in Figure 3:9 is shown listing several configuration files. Each file contains the information necessary to connect to a shared device. A user just picks a previously-used device and the current window is pushed to it.

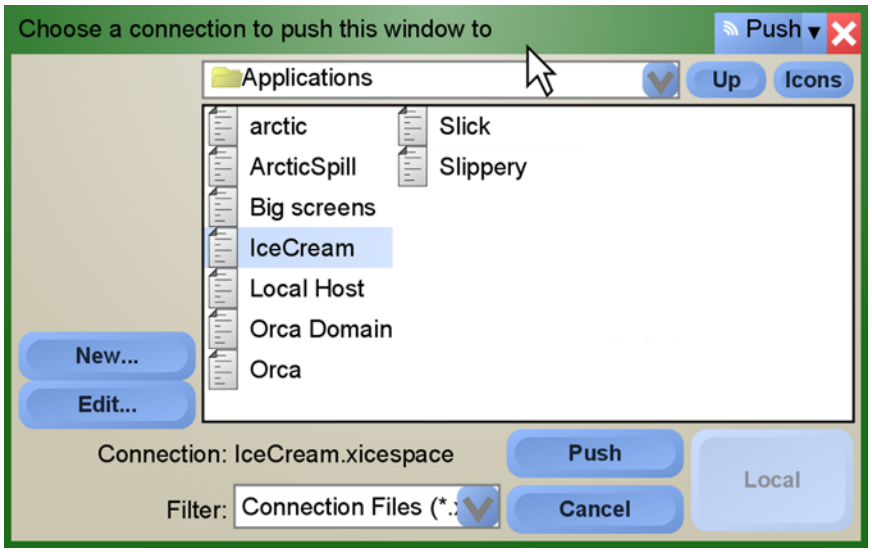


Figure 3:9 – The XICE connection dialog. A user has chosen to push a window to the machine he named IceCream. By default, the configuration files are stored in the "Applications" folder and the view shows only the configuration files by filtering on the file extension.

If the user wants to connect to a new shared device, the user may click the “new” button. Figure 3:10 shows the new connection wizard which walks the user through configuring the connection. Using this wizard configures the following values:

- The shared device’s connection string (domain name or IP address)
- The connection name (e.g. Orca), created automatically or assigned by the user
- The shared device’s trust level (described in section 3.5.2)
- Where input comes from (defaults to the personal device)

The user only needs to enter the connection string and select the screen’s type. The connection name is a unique identifier for the screen generated from the connection string but the user may change it. The three most common types of screens are presented to the user, each with different presets for the display server’s trust level and input sources. These presets are discussed in the next section. Configuration files store all these settings so that regularly used devices, such as desktops or home televisions, do not need to be reconfigured.

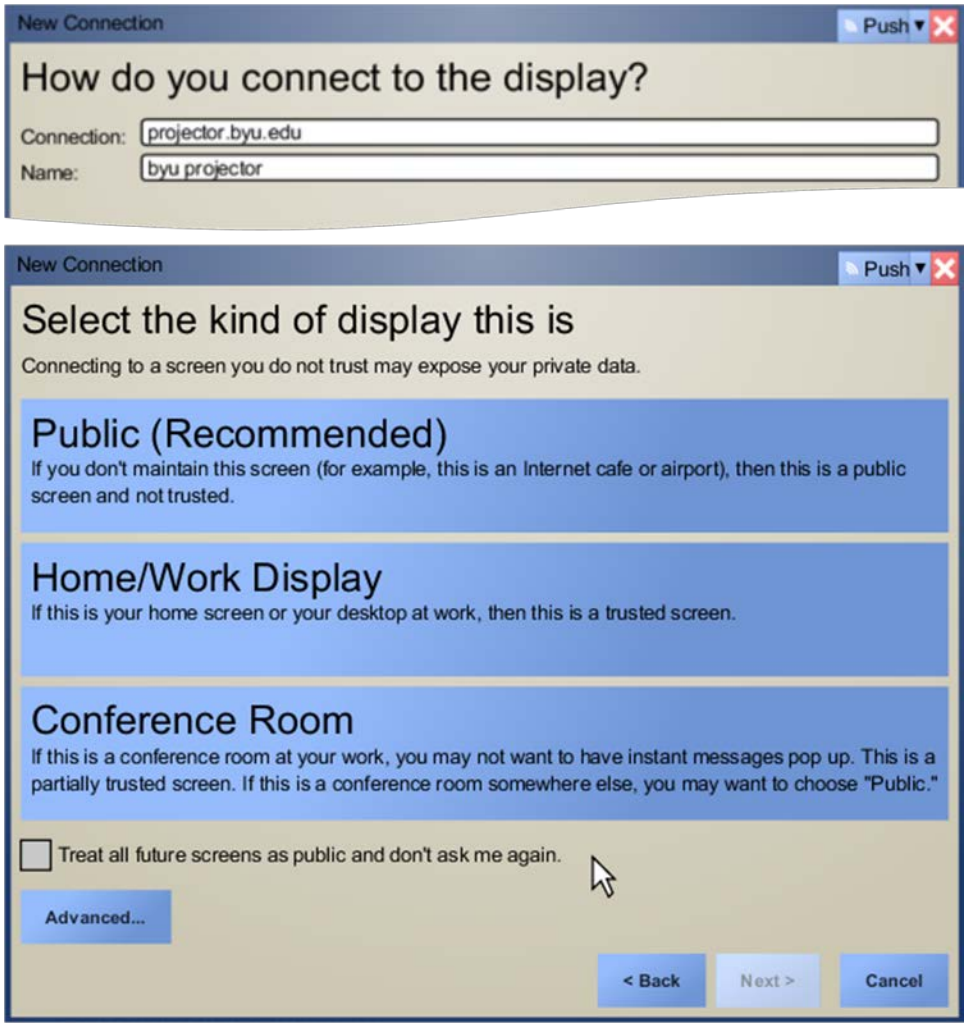


Figure 3:10 – The XICE new/edit connection wizard. The user must supply the connection string and select a trust level for the display.

The process of selecting a device or entering its connection information could be simplified by using local area network broadcasting technologies such as Bonjour [24]. Currently XICE does not use these technologies, but if it did, these technologies would list shared devices near the personal device so that XICE could show only proximally available devices, highlighting those the user has previously annexed. These broadcasting technologies could prevent clutter from the many devices the user has annexed before.

3.5.2 Mitigating the Privacy Problems

When a user first connects to a screen he may choose one of the three types of screens presented in Figure 3:10. This selection results in setting two separate options: the input source for windows shown on that screen and the privacy state for the shared device. Input can either come from the personal device or from the shared device. Input from the personal device is always trusted. However, if the input source is the shared device then the trust level affects that input.

The shared display privacy state may have one of four *trust levels*. When a screen is public, then windows shown on the shared device should only show the user's open data. Private screens should show the user's open data and sensitive data. With respect to shared input, if the input is trusted, then any input from the shared device is accepted and processed normally. But, when the input is distrusted, software may take protective actions. For example, if the user clicks on the "Show Private Data" option in Figure 3:3, then that click is confirmed on the personal device. Most of the time, input is allowed without double-checking—like when typing up an email via a public display—but the input is double-checked when it affects sensitive data.

The four trust levels are delineated in Figure 3:11. The first trust level is *public only*, where the display is public and input is distrusted. The *public with input* trust level is useful when the user trusts the input but the environment is public. A user hosting a discussion in his company-owned conference room can implicitly trust the display server's input to not be malicious, but he may not want sensitive data to appear on the shared display. The public with input trust level protects his sensitive data without needing to confirm input that affects sensitive data. The third trust level is *display only*. The display only option may be used in the situation where the user trusts the audience but not the input. For example, if a lawyer is working with

clients he believes are providing malicious input he may treat the display as public and the input as distrusted. Finally, *full trust* is assigned by the user when the display is private and the input is trusted, such as at his desktop.

Display \ Input	**Distrusted**	**Trusted**
Public	*Public Only* Show open data and treat input as distrusted	*Public with Input* Show open data and treat all input as trusted
Private	*Display Only* Show everything and treat input as distrusted	*Full Trust* Show everything and treat all input as trusted

Figure 3:11 – The four different trust levels. The user can trust or distrust input from public or private displays.

An important feature of XICE is the ability to redirect input—input from the personal device may be used to interact with a UI shown on an annexed display. For example, a user may use the input hardware on his personal device to control a mouse cursor on the annexed display. The display is informed of the cursor movement but none of the mouse clicks. The clicks are dispatched to the widget the cursor is over. After selecting a widget the user can use the input hardware on the personal device to enter text into that widget. This input redirection adds an extra level of security for users selecting the public only and display only trust levels.

When the user selects the type of screen in the step of the connection wizard in Figure 3:10, that selection chooses both the trust level and input source. While each trust level can accept a display server's input (trusted or distrusted), the personal device does not necessarily accept that input by default. The combination of trust level and input source results in eight possible options. Rather than presenting the user with all eight options, the authors opted to provide users with what they believed to be the options best-paired with the three most common situations, hence only three options in Figure 3:10. Figure 3:12 enumerates each option with its trust level and input source settings. The presented options are statically defined and are the most

common options the user is expected to encounter. The user may, at his discretion, click the "Advanced" button and adjust the trust level and input source directly; in such cases all four trust levels are presented to the user along with both input sources.

Room Type	Trust Level	Input Source
Public	Public Only	Personal Device
Home/Work	Full Trust	Display Server
Conference Room	Public with Input	Personal Device

Figure 3:12 – Settings used for each of the options presented in Figure 3:10.

Because the dialog in Figure 3:10 is not actually tied to the XICE protocol but is instead part of the UI presented to the users, other handheld providers could implement these options differently. For example, home consumers could be presented with a dialog that is different from the dialog shown to corporate consumers. Or the dialog's options could be dynamically provided. For example, when the user connects via his employer's wireless network the widget could show him only the work and conference room options with the conference room option listed first. Alternatively, if the user connects via a wireless network he labeled public (e.g. through the Microsoft Windows Network Configuration dialog) then the XICE configuration dialog could present him with just the public only option.

3.5.2.1 Customizing Display Privacy Settings

The trust level is attached to the connection information for the shared device, and is not configured by the shared device but is configured by the user. Different users may apply different privacy states to the same shared device. For example, a user may set his desktop device to full trust, while a visiting coworker might consider the same shared device to be public with input.

A user may want to change the trust level of a shared display. Consider the *visiting student situation.* Suppose a teacher is preparing student grades at her desktop when a student

stops by to review his grades. The teacher would want the full trust desktop display to change to public with input so it will block her sensitive data (like others' grades) from the visiting student.

3.5.2.2 Customizing Window Privacy Settings

Each window shown on a shared device can have a privacy setting that is independent of the device's privacy state. On private screens, the window's privacy setting is ignored and sensitive data is presented. On public devices, *protected windows* protect their sensitive data while *exposed windows* show their sensitive data. By default all windows on public displays are protected windows but the user may change the windows to exposed windows.

This mixed-state view of windows on the public device has some useful situations. Suppose the student in the visiting student situation has submitted an assignment containing sensitive data; the teacher needs to review that assignment with the student. The teacher can open the assignment and change that window to an exposed window using the "Show Private Data" menu option in Figure 3:3. The teacher and the student can review the assignment as though that window is on a private display.

3.5.2.3 Customizing Widget Privacy Settings

In addition, individual widgets may have their own privacy settings. For example, in the visiting student situation, the teacher may want to review the student's grades with him. If each row in the grading spreadsheet has an independent privacy setting, the teacher may change the setting on the row containing that student's grades. Only his grades are presented; other students' grades remain hidden.

3.5.3 Blocking Data on Public Devices

Building a custom toolkit affords opportunities for quick and easy experimentation, solidification of solutions built using the toolkit, and development of custom UI responses to the privacy state of a device [25]. XICE is constructed to accomplish these goals. The rest of section 3.5 focuses on how XICE is implemented to support privacy for both users and developers.

3.5.3.1 Blocking Windows

To show a UI on a display, an application must first build a scene-graph representation of the UI. When the application creates a window the application supplies a scene-graph which XICE renders on the created window. Because a scene-graph may be rendered on one window and a window may render one scene-graph, the two terms are interchangeable.

An application may show five types of windows (scene-graphs). *Plain* windows do not carry sensitive data and can be shown on public or private screens. *Sensitive-data* windows (like emails) may initially be shown on public devices, but the application will protect the sensitive data. *Private-first* windows (like instant messages) might contain sensitive data, so these windows are initially shown on personal devices and users must explicitly choose to show them on public screens. *Private-notify* windows (like file dialogs) function similarly to private-first windows, except a notification is shown on the public screen when the private-notify window is redirected to the personal device. *Private-only* windows (like login dialogs) contain only sensitive data and will never be shown on shared devices. Users and software are not allowed to push private-only windows to public displays.

To create a consistent experience for users XICE uses Java reflection to identify several of these different windows. XICE detects a specific annotation, @PrivateDialog, on the root class of a dialog's scene-graph and inspects its properties. Then XICE can consistently redirect

those windows as necessary. XICE provides the PrivacyHandling enumeration which can be passed to the @PrivateDialog annotation constructor for private-first (First), private-notify (Notify), and private-only (Only) windows. For example, the XICE-supplied FileDialog type is tagged with the @PrivateDialog annotation (circled in Figure 3:13), which defaults to being a private-notify. If the @PrivateDialog annotation is detected then XICE will show that scene-graph on the personal device instead. Plain windows and sensitive-data windows are untagged—the salient difference is in how the application treats the data within the window.

Figure 3:13 – How to annotate a dialog as private-notify.

If the user expects an application to show a window on the shared device, he might not realize the window has been redirected to the personal device. So, when an application launches a private-notify or private-only window and XICE redirects it to the personal device, the toolkit automatically shows the heads-up window in Figure 3:14 to cue the user to look at his personal device.

Figure 3:14 – When showing a file dialog on a public display, the user is given a heads-up dialog instructing him to look at his portable device.

Because the window type may be attached to the window's scene-graph, XICE can enforce the privacy setting for that window wherever that window is used. Otherwise, developers must implement the redirection code everywhere they use those dialogs which may lead to occasional leaks. Instead, by using Java annotations, developers do not have to consider privacy when showing private-first, private-notify, or private-only dialogs. For example, XICE-supplied file dialogs and any subclasses are tagged as private-notify, so the embarrassing situation illustrated in Figure 3:6 will not happen with any file dialog even if the application using the dialog is not privacy-aware.

All five types of windows are created by applications. To show a window, the application must have a reference to the destination display. This reference is represented as a *Space* object within XICE. One Space represents the personal device. Any annexed device is represented with an additional Space.

The Space object has a property that contains the privacy state for the device. Applications can query this value and make decisions independently of XICE. For example, a word processing application may provide an option for always opening documents on private screens. If the option is set, then the software uses code similar to the listing in Figure 3:15 to redirect documents to the personal device's display when the open command is issued via a public screen.

```
public void openFile(String fileName)
{
  Space space = locateSheet().getSpace();
  if(space.getPrivacyState() == PrivacyState.Private)
  {
    //Display is private: show on shared device
  }
  else
  {
    //Display is public: show on the personal display
  }
}
```

Figure 3:15 – Using this code, an application chooses where to show all newly opened documents.

3.5.3.2 Blocking Widgets

In order for a window to appear on a shared device, the scene-graph for that window must be serialized and sent to the shared device. Widgets are embedded within scene-graphs. XICE only serializes the graphical output of the widgets and does not serialize any of the code that controls those widgets. XICE uses a custom text-based serialization framework so that software in other programming languages and frameworks may benefit from XICE.

For sensitive-data windows, developers may protect sensitive data by hiding widgets within the window. For instance, a developer may choose to hide only the username and password boxes on a public device, but the developer wants these boxes to show normally on

private devices. XICE provides the FullPrivacy widget to address this situation. The developer uses code similar to the listing in Figure 3:16 to wrap each widget he wants protected with a FullPrivacy widget, and then widgets containing sensitive data are blocked on public devices, but available on the user's personal device. The FullPrivacy widget ensures that the widget is only seen on private screens; the widget is replaced with a gray rectangle on public screens so that none of the scene-graphs produced by the widget is transmitted to the shared device. This coding technique allows the interactive behavior of any application to be simply wrapped within standard privacy controls.

```
private void makePrivate(ResizableContainer widget)
{
  FullPrivacy container = new FullPrivacy();
  container.setPrivateView(widget);
  this.add(container);
}
```

Figure 3:16 – When getting automatic privacy protection, a developer would add the first two lines in this method.

Suppose a developer has created a simple application for organizing notes. A user can create notes anywhere within the application window, color the notes any way he chooses, and reposition the notes. This application is illustrated in part (a) of Figure 3:17. Further, suppose this developer believes that users may want to designate certain notes as sensitive. A user would select a note then instruct the application to tag that note as sensitive. The developer can then use the FullPrivacy widget to protect the sensitive notes. The note widgets need not consider privacy issues and the FullPrivacy widget knows nothing about notes but protects all interaction inside of it.

a)

b)

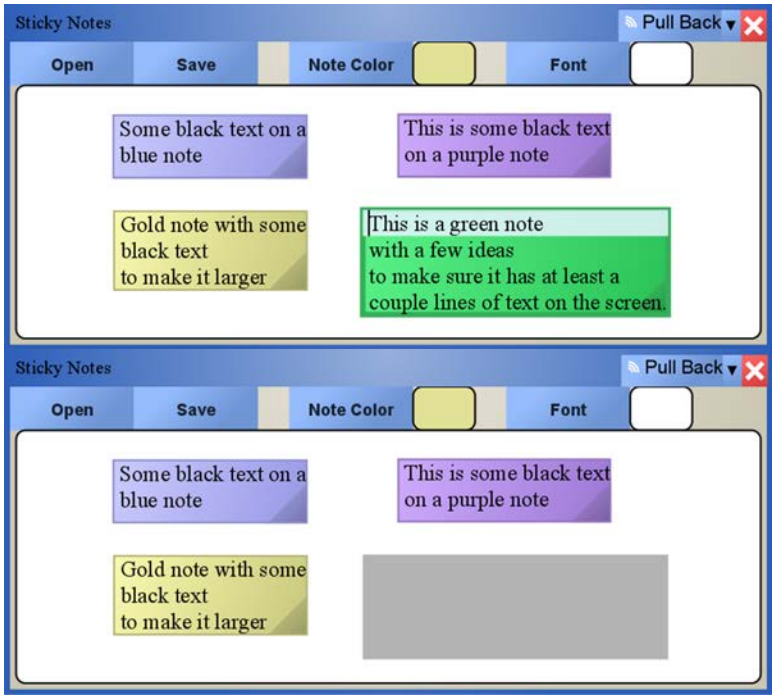


Figure 3:17 – Augmenting an existing application to protect sensitive data. Image (a) is the original application. The bottom-right widget is wrapped with a FullPrivacy widget. Image (b) shows the same window on a public device.

An alternate option to graying out a sensitive widget would be to hide the widget entirely. For some users hiding such widgets may be useful. For example, the act of graying out widgets informs other people in the room that the user does not trust them to see the contents of those widgets. The user may want to have the widget completely hidden instead. On the other hand hiding such widgets makes it difficult for a user to know of the existence of the hidden widget. If he is familiar with the application's interface and data then hiding the widget may be acceptable. But if he is not familiar with the interface and data then he may have trouble addressing the reviewing situation or the field override situation. The relative benefits of both approaches must be weighed by the application developer.

3.5.3.3 Customizing the Public View

Application developers may want to customize sensitive-data handling according to the shared device's privacy state. For instance, a spreadsheet developer might build an application that acts like a normal spreadsheet on a private screen, but hides rows and columns containing financial information when shown on a public screen. Or, an email application developer might show emails in their entirety on a private screen and then block all email addresses embedded within an email when the email is shown on a public screen.

Similar to how the Space supplies its privacy state each window has an independent privacy setting property. Applications can query this information to change their appearance. Code for getting the privacy setting is shown in Figure 3:18.

```
public void visualSetupMethod()
{
  Sheet window = this.locateSheet();
  if(window.getImplicitPrivacyState() ==
       PrivacyState.Private)
  {
    //Setup Private View (show everything)
  }
  else
  {
    //Setup Public View (block sensitive data)
  }
}
```

Figure 3:18 – Template code for a custom visual setup. This code checks on the privacy state of the widget's window.

Any time the privacy setting of a window changes, embedded widgets are notified of that change. For example, when a window is pushed from a private screen to a public screen, a *recontext event* occurs. A recontext event is somewhat similar in purpose to repaint or damage events in more traditional damage/redraw architectures. When the user changes the privacy setting of a window, the recontext event also occurs. The recontext event notifies each widget of

the change, allowing the widget to query the window's changed privacy setting and update the widget's output. Because XICE delays scene-graph serialization until after all widgets have processed the recontext event, the private view of the widget will never be inadvertently serialized to a public screen.

3.5.4 Reviewing Sensitive Data

Blocking sensitive data on applications can become frustrating to users who need frequent access to that data. If a user can access his sensitive data without exposing it on a public screen, as in the reviewing situation, his privacy remains protected.

XICE facilitates reviewing sensitive data on a personal device without exposing that data on shared devices. Reviewing is supported for windows, widgets, and custom reviewing situations.

3.5.4.1 Reviewing Windows

Consider the spreadsheet application mentioned in section 3.5.3.3. If a user opens this application on a kiosk in a public environment, he may want to see broad overviews of the data without exposing individual values. The user could interact with a spreadsheet on a public screen, and sensitive data in the corresponding cells are highlighted on the personal device. Then the user can glance at his personal device to see the actual values without exposing the sensitive data on the shared device. Synchronizing the public screen and personal device spreadsheet views is particularly helpful if the personal device is a handheld which can only show a few columns and rows of data at once. The synchronized view is illustrated in Figure 3:19.

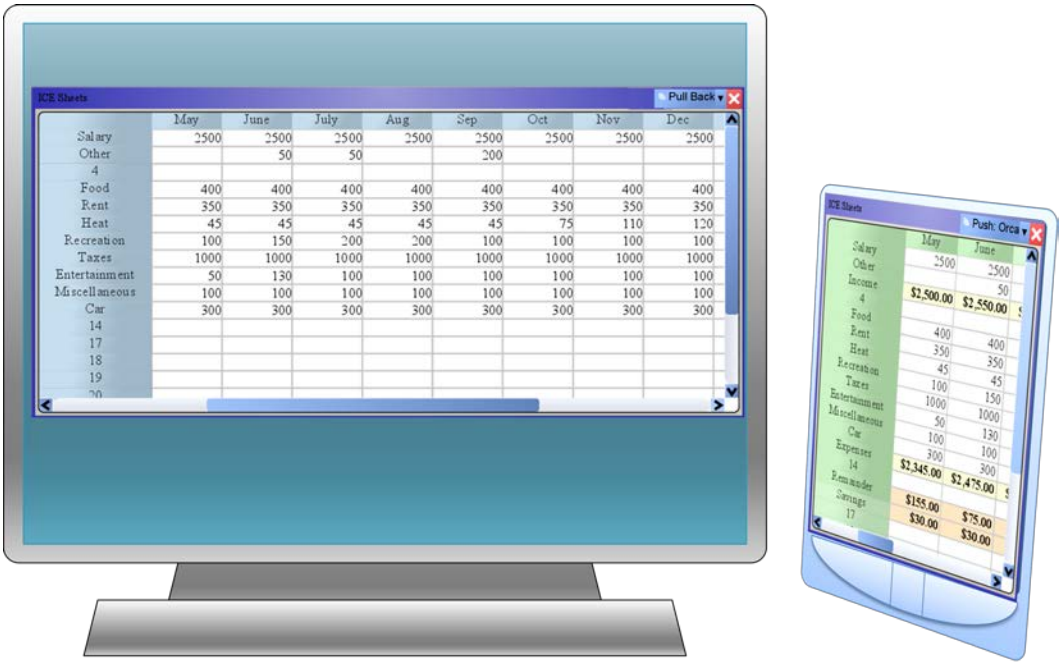


Figure 3:19 – A possible synchronized custom view. The public display (left) blocks sensitive data. The portable device (right) simultaneously shows the same spreadsheet with the sensitive data exposed (the rows highlighted at the bottom and near the top).

To synchronize the two spreadsheet views, one solution would be to put the widget in both scene-graphs. However, if widgets were in multiple scene-graphs, a single widget instance would have to support a potentially unlimited number of contexts (i.e. privacy settings for each window), states, and sub-graphs. To minimize the demand on the widget, simplify the scene-graph representation, and streamline the serialization process, XICE does not allow a widget to be embedded in multiple locations. Instead, widgets built using model-view-controller (MVC) design facilitate the same visual result—two views which share the same model.

To create synchronized views across devices, XICE ensures that all view-controllers (scene-graphs and widgets) can be cloned and that cloned view-controllers share the same model. As XICE clones each widget, only its view-controller is copied; then the copy is pointed at the original widget's model. The cloned scene-graph is rendered on a new window on the personal

device. The original view remains on its window on the shared device and retains its privacy settings.

From the user's perspective cloning windows is straightforward. To create a clone of a protected window on the personal device, the user clicks the "Copy To Handheld" menu option in the title bar of the protected window (Figure 3:20). XICE then clones the scene-graph and renders the clone on a new window on the personal device.

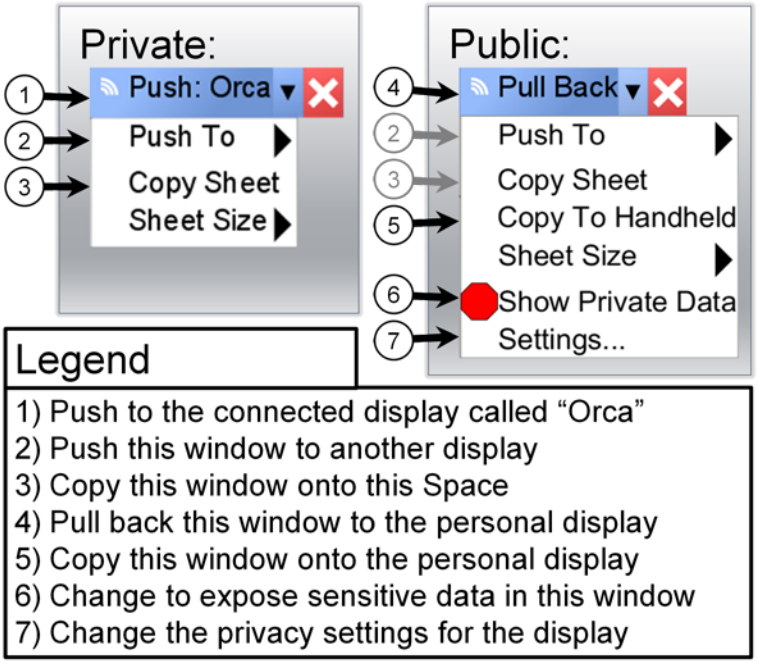


Figure 3:20 – Privacy-related menu options on the title bar of every window. Some options are specific to public displays so users can properly protect their data.

3.5.4.2 Reviewing Widgets

Section 3.5.3.2 mentions blocking a username and password using the FullPrivacy widget, but a user must still be able to enter input to those widgets via his personal device. When annexing a private screen, he can access the username and password normally. On a public screen, he must take a few extra steps. First, he clicks on (or hovers the cursor over) the gray rectangle where the username or password textbox should be. The FullPrivacy widget then

shows the textbox on the personal device (this is illustrated in Figure 3:21). He then clicks the textbox on the personal device and enters his credentials using the personal device's input. This interaction redirection is completely invisible to the application.

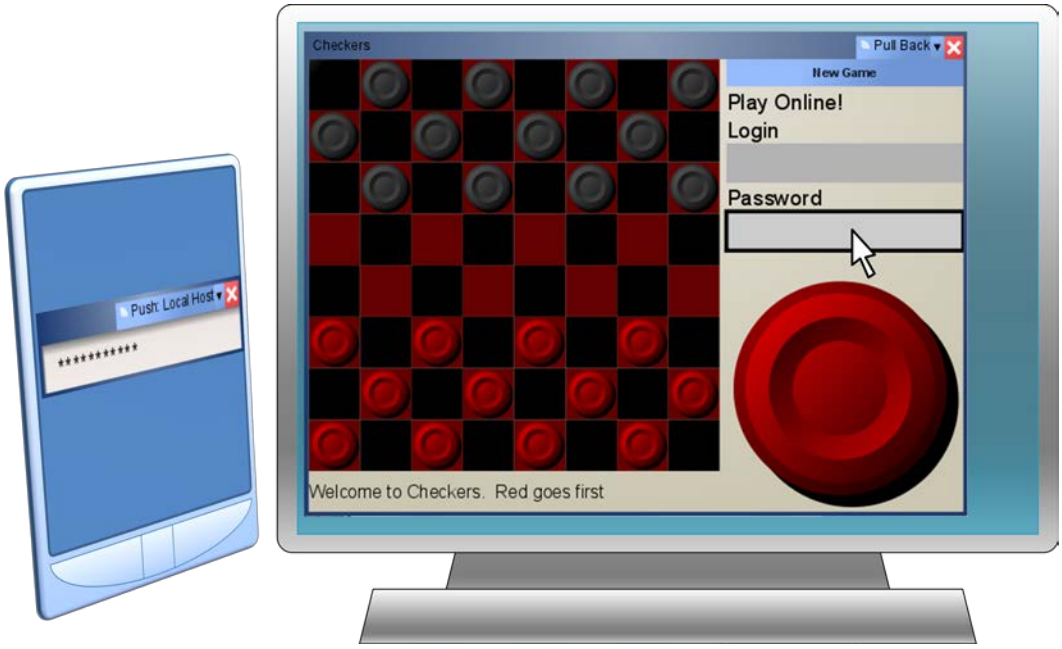


Figure 3:21 – Hovering over a FullPrivacy widget on a public display (right) shows the blocked widget on the personal display (left).

3.5.4.3 Custom Reviewing

If a developer chooses to customize data blocking, XICE provides window and device privacy states, but does not offer much other assistance. Any custom reviewing will require more programming time and effort. The developer may instead decide to rely on XICE's cloning infrastructure (section 3.5.4.1) to resolve the reviewing situation.

3.5.5 Privacy Control

In addition to reviewing sensitive data on the personal device, users may want to show that data on a public screen. Perhaps the user does not consider the data sensitive for a particular

audience. XICE facilitates users showing windows and widgets on public screens, and supplies applications with enough information to implement custom data exposure options.

3.5.5.1 Showing Windows

Whenever a shared device is annexed, a connection management window is shown on the personal device. On this window, a button click causes the privacy state of the Space to change from public to private or vice versa. This functionality addresses the portion of the visiting student situation when the teacher must change her desktop to the public state.

Each window on a public display has the “Show Private Data” menu option, which allows the user to expose all sensitive data on that window. This menu option resolves the email issue raised in section 3.3 by allowing users to show individual emails and all the sensitive data in them on public screens, if they so desire.

3.5.5.2 Showing Widgets

If a developer uses the FullPrivacy widget to block another widget, the user can expose the blocked widget on a public screen. On public screens, when the user clicks on the gray rectangle that replaces the private widget, a “Show Data” option is provided which exposes the blocked widget. When the input is tagged as unsafe, the user must confirm that widget exposure on the personal device. This option remedies the field override situation (mentioned in section 3.3).

3.5.5.3 Custom Showing

If a developer chooses to customize sensitive data blocking, he should also create code for exposing that data. While the developer could rely on the “Show Private Data” menu option to show the sensitive data, this solution is discouraged, because all sensitive data would show on

that window and the user may only want to show a portion of the sensitive data. Instead, the developer should provide custom tools for exposing individual pieces of sensitive data.

For instance, in the visiting student situation, the teacher wants to review only one student's grades. On a public screen, the spreadsheet application could provide a custom context menu to allow the teacher to expose only a single row of data—the row containing that student's grades. The spreadsheet would track exposed cells and protect all other cells containing sensitive data.

3.5.6 Developer Summary

An important feature of XICE is the simplicity it affords developers when creating privacy-sensitive applications. This simplicity surfaces at the toolkit, window, recontext, and widget levels.

First, because overall privacy settings are tracked by the toolkit, developers do not need to add an overall privacy-tracking solution to their application and users have a consistent, central location for specifying the privacy state of a shared display.

Second, if a scene-graph is tagged with a @PrivateDialog annotation, then that scene-graph's privacy behavior are interpreted consistently wherever the scene-graph is used. Consequently a developer's decision about privacy may be made once when the scene-graph for that dialog is designed, but enforced everywhere that scene-graph is used without rethinking about the privacy decision. Even developers who do not consider privacy when developing their applications can benefit. An application that uses the FileDialog scene-graph, or any subclass thereof, will automatically use that dialog in privacy-sensitive ways.

Third, because of the recontext event and independent privacy states for each window, developers can depend on getting and processing the recontext event before the scene-graph is

serialized to the shared device. This ensures that no application-known sensitive data is transmitted to a public screen. If privacy is implemented externally to the UI distribution software then this guarantee is more difficult to ensure.

Finally, developers can wrap widgets with a FullPrivacy widget to protect that widget's contents. If the developer so desires, he may integrate the FullPrivacy widget into the custom widget so that everywhere the custom widget is used its sensitive data is protected.

In addition, developers can create individual widgets which perform custom privacy protections, typically with little effort. For example, a spreadsheet application is available that uses XICE as its rendering framework. This application represents well over 200 hours of development time, most of which involved writing spreadsheet functions. Privacy-aware shared UI features were added to this application after it was fully developed. The new privacy features required less than 8 hours of another developer's time. The second developer was not involved in coding the spreadsheet application so the 8 hours include his time discussing the existing design with the original programmers. However, the second developer was very familiar with XICE's design (he designed XICE) so overhead related to the overall windowing toolkit was low. This time also included adding application-specific storage of "private" tags to the spreadsheet's data file. The incorporated privacy-aware shared UI features allow rows, columns, or cells to be marked sensitive, and then sensitive rows and columns are hidden on public screens while sensitive cells are grayed out. The synchronized view in Figure 3:19 demonstrates this spreadsheet application in use.

3.6 Usability Walkthrough

Evaluating the usability of toolkits is difficult. Part of this difficulty is because the toolkit is a large development effort. User evaluations provide tight data for controlled experiments with

few variables, but toolkits have so many variables that exactly quantifying the effect of each feature on the overall experience is difficult [26]. Therefore, usability experiments are either trivial—collect conclusive data on a small feature—or cannot be generalized—too complex or require too many caveats to be useful. Because a usability experiment on XICE would be too complex to provide useful data, this paper does not include a user study but includes a usability walkthrough.

By doing a walkthrough, one can perceive the burden on a user and characterize the decisions that he must make. In particular, this walkthrough answers questions about when the user encounters the shared UI privacy features and how the user can choose to change privacy settings.

3.6.1 Home/Work Displays

Consider a user who just purchased a new XICE-enabled cell phone and who is connecting it to his home desktop screen for the first time. To connect to his home display he must configure the display using the wizard in Figure 3:10.

He could correctly click the “Home/Work” button, or he could click the “Public” or the “Conference Room” button. Regardless of the choice the user makes, the new connection wizard in Figure 3:10 will not reshow when he connects to that display again. If the user correctly chooses the “Home/Work” button, he will receive no further privacy settings or warnings, which is appropriate because he is using his home screen.

Let us assume that the user picks the “Public” option. He then attempts to open a word processing document via his home display. Because the software is informed that the display server is distrusted, the XICE toolkit redirects the file dialog to the cell phone, similar to Figure 3:14.

It is likely that either now or at some future point (because sensitive dialogs are continually redirected to the personal device) the user will become annoyed and desire to change this setting. On the redirected dialog, as shown in the upper-right of Figure 3:22, XICE provides the "Settings…" button which the user may click to change the privacy settings.

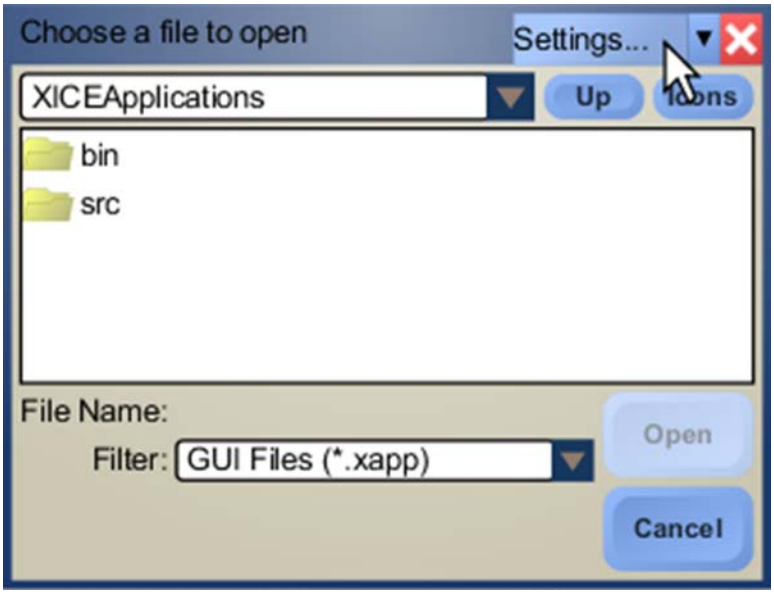


Figure 3:22 – A redirected dialog presents the "Settings…" button which the user may use to access and adjust the display's privacy settings.

Clicking the "Settings…" button shows the configuration wizard in Figure 3:10, again. At this point the user may choose to reconfigure the display as a "Home/Work" display which causes the software to treat the display as full trust. From that time forward, the user will not see any protections for his sensitive data whenever connected to that display.

3.6.2 Corporate Conference Room

When the user first connects to the conference room at his work, he is presented with the configuration wizard. If the user chooses the "Conference Room" option, he can show output on the display server and use the input from the display server. In addition, his sensitive information such as instant messages will not show on the shared display.

As the user discusses data within various applications, he can choose to push windows containing sensitive data to the shared display. He may also choose to show any embedded sensitive data using the “Show Private Data” menu option, or whatever other means the software may provide for overriding privacy settings. For example, if the user has a spreadsheet with sensitive data in it, he may share that window on the conference room screen. The spreadsheet is shown and the sensitive data blocked from appearing. The user can then selectively expose sensitive data within rows, columns, or cells.

If, when configuring the connection, the user instead chooses the “Home/Work” option, then he may be embarrassed by an email or instant message, or may accidentally share sensitive information (e.g. in a spreadsheet). To change the settings so that he is better protected, the user may click the “Settings…” option on any window he shared on display server (option 7 in Figure 3:20).

3.6.3 Foreign Conference Room

The user will likely visit other institutions and may be asked to give a presentation in their conference room. When the user annexes the display server in that conference room, he is shown the configuration wizard.

In the foreign conference room the correct privacy selection for display server is “Public.” Choosing “Public” reduces the chances for the user’s device to be exploited if the display server is malicious (e.g. the maintenance staff is unaware that the display server is infected with malware). The user could choose to accept input from the display server, and the user is still protected from potentially malicious input.

If, on the other hand, the user chooses “Conference Room” and the institution he is at is trustworthy, then the user is unlikely to be exploited. However, he could be exploited if the

display server harbors malicious software. For this reason, the “Conference Room” option includes a description discouraging its use when annexing foreign display servers.

Unfortunately, if the user chooses “Home/Work” for the foreign display, then he may experience privacy violations. Instant messages, file dialogs, or other sensitive data could alert him to his poor choice, and he can subsequently change the privacy settings for that display. After presenting the configuration wizard, the windowing toolkit has attempted to inform a user and help him make good choices about how to connect to a display server. Further explanations may be made available (e.g. through hyperlinks), or the user may be provided with the opportunity to engage in further training to help him understand the privacy system, potential violations, etc. Such a training program may be a help manual, a game, or some other useful, but non-intrusive option. Raja et al have done research showing promising ways to help inform users of the current firewall state and such research should be incorporated in final production dialog designs [27].

Currently people can sense better what kind of environment they are in, but they must be aware of the potential threats as well as opportunities. Some sort of training is necessary to help humans understand how to make appropriate privacy-related choices. When sensing devices improve to the point that the handheld can sense as well as, if not better than, a human then the portable device may be able to assume the responsibility of making these choices. But with current sensing technologies, the toolkit can only present users with options and attempt to quickly inform users of the implications so that users may make informed decisions.

3.6.4 Airplane

When traveling via airplane to another location, the user may encounter a display server embedded in the screen at his seat. Annexing that display server shows the connection wizard yet

again. Although the airline is likely to have a maintenance staff which maintains the display server and keeps it free of malware, the environment is public; other passengers could see what the user is working on.

If the user chooses “Public” or “Conference Room” he will be properly protected. But if he chooses “Home/Work” he will suffer from the same problems he could encounter if he chose “Home/Work” at the foreign conference room.

3.6.5 Foreign Displays

The user will continue to encounter display servers in a variety of environments. Many, if not most, of these display servers will be owned by other institutions or in public environments. If the user, out of convenience, chooses to make each display a “Home/Work” display, his user experience will be smooth in most cases. Unfortunately, he may encounter a malicious display server at some point and the toolkit will not prevent the shared-UI-specific exploits. Thankfully, if the user correctly chooses to make each subsequent display “Public” he will be properly protected by the toolkit and his applications. The configuration wizard in Figure 3:10 provides a checkbox titled “Treat all future screens as public and don’t ask me again” which may be used to smooth the configuration process while keeping the user safe.

3.7 Alternate Implementations

The XICE Framework is not the only way to implement a consistent privacy-aware shared UI framework that resolves the five key problems. This section will explore the requirements for implementing a privacy-aware framework in another windowing toolkit, such as Microsoft Windows, Java Swing, and Magic Lenses [28] or Attachments [29].

For the user, the solution must have two parts. He must be able to specify the privacy state of a shared device and whether he is comfortable showing a particular window. In order for

the developer to support these two features, he must also have access to the privacy state and receive notifications when the privacy state changes. With these four parts in place, developers can create custom privacy-aware presentations for their software.

Personal devices that can annex at least one additional display can support a privacy-aware windowing toolkit. The toolkit must include a suitable UI distribution framework. This distribution could be provided by VNC, X11, RDP, etc. Ideally, the distribution framework would be simple and easy to implement for stand-alone display servers, allowing any personal device from any manufacturer to annex the display server and distribute UIs to it. XICE fills this requirement.

If the UI distribution framework is just a VGA cable, a privacy-aware framework is possible, but properly protecting windows with sensitive data may necessitate some constraints. Imagine a window that opens across two screens—one on the personal screen (private) and the other on the annexed screen (public). If widgets change size or position based on the display's privacy state, providing a consistent and understandable layout for windows that contain sensitive data becomes difficult. Constraining the window to either the public device or personal device is much easier. The windowing toolkit should ensure the window always has a single privacy setting, even if the window is partially on each display.

Once users can specify trust levels for devices and windows, the rendered output of applications needs to be augmented to protect privacy. If applications are privacy-aware, they may augment themselves. However, if an application is not privacy-aware, a well-designed windowing toolkit can aid in protecting user privacy by blocking file dialogs.

Magic Lenses [28] offer a means for augmenting an application's graphical output without the application's direct knowledge. A lens is a virtual peephole or viewport over the

application. Within that viewport, an augmented view of the application is exposed. This augmented view may make lines dotted, squiggly, or a different color, potentially to obscure sensitive data. Any rendering command can be augmented to yield a different appearance for an application.

Attachments [29] allow one application to attach data to the visual output of another application. The application constructing the attachment accepts the original rendering commands of the source application and then adds rendering commands around detected widgets, similar to Magic Lenses. During the attachment process original rendering commands may also be augmented.

Magic Lenses or Attachments could become means for implementing privacy awareness. Either of the tools would be used where a public screen has a full-screen lens that affects all applications shown on that screen. Applications would render themselves normally, and then the privacy lens would obscure potentially sensitive data (e.g. email addresses, proper names, file paths, etc.).

One significant benefit of a lens-based system is that developers would not need to consider privacy when writing applications. As a result, however, developers may be discouraged or prevented from developing novel privacy solutions, in effect, hampering progress in this field. In addition, a side-effect of Magic Lenses and Attachments is that a widget's graphical output is basically the same size regardless of the lens—the lens has a difficult time effectively resizing widgets and drastically changing graphical output within the viewport while maintaining normal user interaction.

Of course, in each of the mentioned alternate systems, input from annexed devices to the personal device must also be treated according to the trust level of the device. Input would need

to be tagged with the input source's trust level, so that applications can make appropriate decisions, like allowing, blocking, or confirming input on the personal device when input affects sensitive data.

3.8 Concluding Remarks

The proposed XICE privacy framework resolves the major issues involved in allowing users to annex shared devices. XICE demonstrably protects users from the five key privacy problems of stolen output, stolen input, false input, false output, and embarrassment. Vital to the success of the proposed solution is a simple content control system and information about whether a screen is private or public and its input is trusted or distrusted. Using this privacy information, the windowing toolkit and any applications running on it can prevent sensitive data from showing on public displays (by altering their appearance), discourage users from entering private information on public devices, and mitigate the effects of actively malicious devices that falsify user input or graphical output.

Users can manage their privacy needs using the privacy-aware framework, XICE. In particular, a foreign device is treated as public and distrusted by default. The user can change, as needed, the privacy state for that device, for windows shown on that device, and for privacy-aware widgets within those windows.

XICE provides a flexible, powerful privacy framework to developers. This framework requires less overhead for creating privacy-aware applications than building such applications via existing UI distribution frameworks. In particular, using annotations and the FullPrivacy widget allow developers to protect sensitive dialogs and widgets by designing that protection into the widget instead of implementing the protection code everywhere the dialogs or widgets are used.

Implementing privacy awareness does not need to be constrained to XICE, but should be part of any windowing toolkit that annexes shared devices. As people become increasingly mobile, the need to annex devices will grow, privacy will become an even greater concern, and a privacy-aware toolkit will become a necessity.

3.9 REFERENCES

1. Scheifler RW, Gettys J. The X Window System. *ACM Transactions on Graphics* 1986; **5**(2):79-109. DOI: http://doi.acm.org/10.1145/22949.24053.
2. Tritsch B. *Microsoft Windows Server 2003 Terminal Services*. Microsoft Press, 2003.
3. Richardson T, Stafford-Fraser Q, Wood KR, Hopper A, Virtual Network Computing. *IEEE Internet Computing* 1998; **2**(1). DOI: http://doi.acm.org/10.1145/22949.24053.
4. Sharp R, Madhavapeddy A, Want R, Pering T. Enhancing web browsing security on public terminals using mobile composition. In *Proceeding of the 6th international Conference on Mobile Systems, Applications, and Services* (*MobiSys '08*), Breckenridge, CO, U.S.A., 17–20 June 2008. ACM: New York, NY, 94-105. DOI: http://doi.acm.org/10.1145/1378600.1378612.
5. Arthur R, Olsen D.R. XICE Windowing Toolkit: Seamless Display Annexation. *ACM Transactions on Computer-Human Interaction* (*ToCHI*) 2011.
6. Facebook, Available at: http://www.facebook.com/ [October 2010].
7. MySpace, MySpace, Inc., Available at: http://www.myspace.com/ [October 2010].
8. Howard M, LeBlanc D. *Writing Secure Code* (2nd edn). Microsoft Press, 2003.
9. Apple Computer. Macintosh. Apple Computer, Available at: http://www.apple.com/mac/ [October 2010].
10. Izadi S, Brignull H, Rodden T, Rogers Y, Underwood M. Dynamo: A public interactive surface supporting the cooperative sharing and exchange of media. *User Interface Software and Technology* (*UIST '03*), ACM: New York, 2006; 159-168. DOI: http://doi.acm.org/10.1145/964696.964714.

11. Want R, Pering T, Danneels G, Kumar M, Sundar M, Light J. The Personal Server: Changing the way we think about Ubiquitous Computing. *Ubiquitous Computing* (*UbiComp '02*). Springer: Berlin, 2002; 223-230. DOI: http://dx.doi.org/10.1007/3-540-45809-3_15.

12. Hawkey K, Inkpen KM. PrivateBits: managing visual privacy in web browsers. In *Proceedings of Graphics Interface 2007* (*GI 2007*). ACM Press: New York, 2007; 215-223. DOI: http://doi.acm.org/10.1145/1268517.1268553.

13. Tarasewich P, Gong J, Conlan R. Protecting private data in public. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems* (*CHI 2006*). ACM Press: New York, 2005; 1409-1414. DOI: http://doi.acm.org/10.1145/1125451.1125711.

14. Berry L, Bartram L, Booth KS. Role-based control of shared application views. In *Proceedings of the 18th Annual ACM Symposium on User interface Software and Technology* (*UIST 2005*). ACM Press: New York, 2005; 23-32. DOI: http://doi.acm.org/10.1145/1095034.1095039.

15. Berger S, Kjeldsen R, Narayanaswami C, Pinhanez C, Podlaseck M, and Raghunath M. Using Symbiotic Displays to View Sensitive data in Public. In *Third IEEE International Conference on Pervasive Computing and Communications* (*PerCom 2005*) IEEE Computer Society: Silver Spring, MD, 2005; 139-148. DOI: 10.1109/PERCOM.2005.52.

16. Oprea A, Balfanz D, Durfee G, Smetters DK. Securing a remote terminal application with a mobile trusted device. *20th Annual Computer Security Applications Conference* (*ACSAC '04*), 6-10 December 2004*;* 438-447, DOI: http://doi.ieeecomputersociety.org/10.1109/CSAC.2004.33.

17. Sharp R, Scott J, Beresford AR. Secure Mobile Computing via Public Terminals. In *Proceedings of the International Conference on Pervasive Computing* (*PerCom 2006*). IEEE Computer Society: Silver Spring, MD, 2006; 238-253. DOI: http://dx.doi.org/10.1007/11748625_15.

18. Yue C, Wang H. SessionMagnifier: a simple approach to secure and convenient kiosk browsing. In *Proceedings of the 11th international Conference on Ubiquitous Computing* (*Ubicomp '09*). Orlando, FL, U.S.A., 30 September-3 October 2009. ACM: New York, NY, 125-134. DOI: http://doi.acm.org/10.1145/1620545.1620566.

19. GKS (Graphical Kernel System), ANS X3.124-1985 ANSI, December 1984.

20. Shuey D, Bailey D, Morrissey TP. PHIGS: A Standard, Dynamic, Interactive Graphics Interface. *Computer Graphics and Applications* 1986; **6**(8): 50-57.

21. Microsoft Corporation, Silverlight, Available at: http://www.microsoft.com/silverlight/, accessed June 2010.

22. Petzold, C. *Applications = Code + Markup: A Guide to the Microsoft Windows Presentation Foundation*, Microsoft Press 2006.

23. Oracle Corporation. JavaFX. Available at: http://www.javafx.com/ [February 2010].

24. Apple Computer. Bonjour. Available at: http://www.apple.com/support/bonjour/ [October 2010].

25. Bederson BB, Grosjean J, Meyer J. Toolkit Design for Interactive Structured Graphics. *IEEE Transactions on Software Engineering* 2004; **30**(8):535-546. DOI: http://dx.doi.org/10.1109/TSE.2004.44.

26. Olsen DR. Evaluating user interface systems research. In *Proceedings of the 20th Annual ACM Symposium on User interface Software and Technology* (*UIST '07*), Newport, RI, U.S.A., 7-10 October 2007. ACM: New York, NY, 2007; 251-258. DOI: http://doi.acm.org/10.1145/1294211.1294256.

27. Raja F, Hawkey K, Beznosov K. Towards improving mental models of personal firewall users. *Proceedings of the 27th international conference extended abstracts on Human Factors in Computing Systems* (*CHI EA '09*). ACM: New York, NY, 2009; 4633-4638. DOI: http://doi.acm.org/10.1145/1520340.1520712

28. Bier EA, Stone MC, Pier K, Buxton W, DeRose TD. Toolglass and magic lenses: the see-through interface. *Proceedings of the 20th Annual Conference on Computer Graphics and interactive Techniques* (*SIGGRAPH 1993*). ACM Press, 1993; 73-80. DOI: http://doi.acm.org/10.1145/166117.166126.

29. Olsen DR, Hudson SE, Verratti T, Heiner JM, Phelps M. Implementing interface attachments based on surface representations. *Proceedings of the SIGCHI Conference on Human Factors*

in Computing Systems. (*CHI 1999*). ACM Press: 191-198. DOI: http://doi.acm.org/10.1145/302979.303038.

Chapter 4 SPICE: Lightweight, Media-rich, Screen Annexation

Richard B. Arthur, Kelv Cutler, Mitchell K. Harris, Dan R. Olsen, Jr.

Brigham Young University

3361 TMCB, Provo, UT, 84602-6576, USA

startether@startether.com, kelvcutler@gmail.com, mitchell.k.harris@gmail.com, olsen@cs.byu.edu

ABSTRACT

People need to be able to annex screen space at any time, with any device, anywhere screens are located. This paper introduces SPICE which is a simple protocol for annexing screens and sharing content. SPICE allows devices such as smartphones, laptops, or tablets, to annex a wide variety of display configurations from single projectors to large multi-screen setups. This paper demonstrates the power of SPICE with two applications. The first shows a user using his smartphone to give a media-rich, five screen presentation. The second lets multiple users simultaneously annex screen space and collaboratively share and discuss windows.

4.1 INTRODUCTION

Nomadic users require access to their data, applications, and settings anywhere, at any time. To improve the experience beyond small portable devices, users need to be able to annex additional screen space regardless of the device they are carrying, be it laptop, tablet, or smartphone.

This paper introduces the SPICE (Spaces for Interactive Computing Everywhere) protocol which allows anyone with a portable device to annex additional screen space at any time, regardless of the device he or she is carrying.

Consider a botanist who carries around a smartphone as his only portable computing device. Five screens are available to present information in his classroom. The botanist employs those screens to give a presentation containing pictures of various plants, audio clips from related radio shows, and videos of

plant development. All media and controls are provided by his smartphone. This situation is illustrated in Figure 4:1, and this paper will demonstrate how the SPICE protocol enables such a presentation.

Figure 4:1—Multi-screen presentation. The presentation software, controls, and all media (images, audio, and video) are supplied by the smartphone.

Multiple people must be able to simultaneously share information on the same screen space and discuss that data. As shown in Figure 4:2, SPICE enables collaborative interaction on large display spaces. In this figure, multiple people are each sharing windows on a unified display space. The windows show content from any window the user chooses from his personal device. The window may be positioned anywhere on the unified display space. As can be seen, the three users interleave their windows seamlessly on this display space.

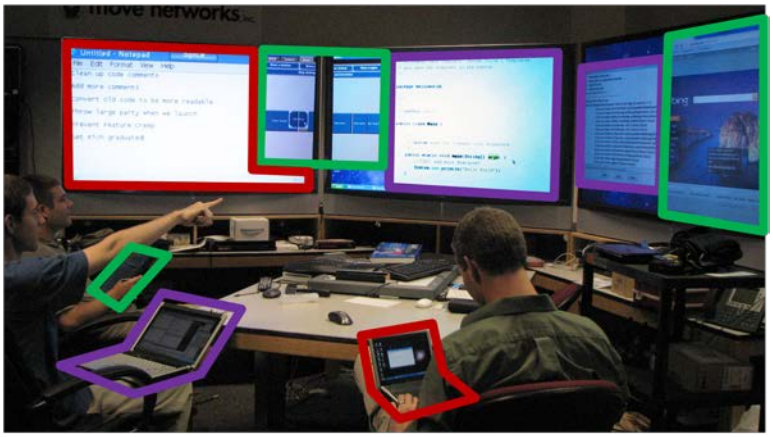


Figure 4:2—Collaborative group meeting. The windows and their source computer are outlined in the same color.

The general organization of SPICE follows the design illustrated in Figure 4:3. Users carry around a *personal device* which may be a laptop, tablet, smartphone, or some other portable computer. The personal device is used to annex a *display server*, which is a computer running a simple protocol but with significant computing resources (RAM, CPU, and screen space). The display server provides the output for the media the personal device transmits to it.

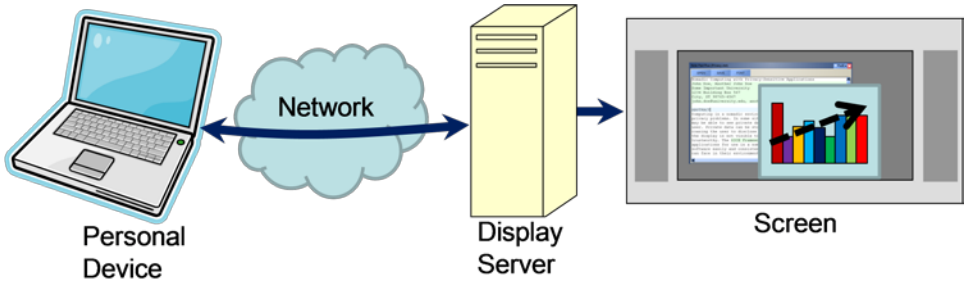


Figure 4:3—Hardware organization for annexing output. The personal device connects to a display server via the wireless network and the display server renders the supplied media.

Nomadic users carry around personal devices while room owners supply display servers for users to annex. Consequently, to gain a ubiquitous nomadic experience the system design must consider the needs of both nomadic users and room owners.

4.1.1 Nomadic Users

A nomadic user travels from place to place and frequently interacts with his data at those destinations. To more effectively support a nomadic user in a variety of locations, the user's experience should be familiar, easy to access, portable, powerful, protected, and collaborative.

For interactions to be *familiar*, the user should be able to use software he already knows at any location. In addition, once a user learns a piece of software for performing some action he should not be required to learn new software to perform the same action at a different location.

A user must be able to *easily access* his data, applications, and settings. He should be able to interact with this data at any time.

To make his data accessible at any time, the user must have a lightweight *portable* computer with which to access his data.

On the other hand the user should have *powerful* computing at his fingertips. He should have media-rich user interfaces (UIs) for his applications.

Less noticed, but often more important, is *protecting* the user's data. The user's data, applications, and settings should not be exposed to viruses or other malware. The user's sensitive data should not be made available for other software to steal. This protection should be seamless so that the user is not unnecessarily interrupted by security measures and can interact confidently with his data, anywhere.

Finally, a user travels to other locations so that he may *collaborate* with other people. Multiple users should be able to simultaneously share information with each other using the same display space.

4.1.2 Room Owners

For users to annex displays anywhere, the annexing protocol must be ubiquitous. To gain ubiquity, room owners must install the display server. However, the room owners must benefit from installing display servers. A large institution may want to improve collaboration among its workers so it will install display servers. A restaurant may want to encourage business lunches. A homeowner may want to install a display server in his television so he can interact with data while on his couch, or install a multi-screen desktop which he can use to expand his phone.

But the display server can present a significant barrier if maintenance is costly. This cost includes the software necessary for user interaction and costs for protecting the display server from malicious software (malware). Users who annex a display server might infect it. Such infection might be intentional—the user is malicious—or accidental—the user's personal device is already infected and the malware is attempting to spread. Although a display server may have exploitable flaws (e.g. buffer

overruns), the protocol for annexing display spaces must not have a design that encourages malware to spread (i.e. readily accept and execute software supplied by the other party). The hardware and software costs to keep it running smoothly must be kept low.

For example, the restaurant owner would prefer to have a solution that is as low-maintenance as many Wi-Fi options. An owner would want to install the device and then let it run with almost no maintenance or direct interaction so he can amortize the costs through food sales. If he must install additional software to meet his customer's needs (e.g. word processors, spreadsheet applications, etc.), then his costs can increase significantly. Maintenance and protection from viruses and malware increase as well. A malware infection that spreads to his customers could be devastating.

To achieve a low-maintenance solution, the display server would need a solid operating system, few (if any) applications installed, and rare updates. Consequently, the annexation protocol must be simple enough for any manufacturer to consistently implement it [43].

If, on the other hand, a display server technology requires significant maintenance, then only institutions that can expend those resources can install such display servers. The homeowner would not install it nor would the restaurant owner. Fewer display servers means less ubiquity for users.

4.1.3 Summary

In summary, the requirements for a successful nomadic experience are two-fold. For users, interactions must be familiar, easy to access, portable, powerful, protected, and collaborative. For room owners, the experience must be low-cost and low-maintenance. Ultimately, the protocol should afford the variety of connections illustrated in Figure 4:4. This figure shows how a variety of personal devices can ultimately use a variety of display space configurations, each via the same standard network protocol.

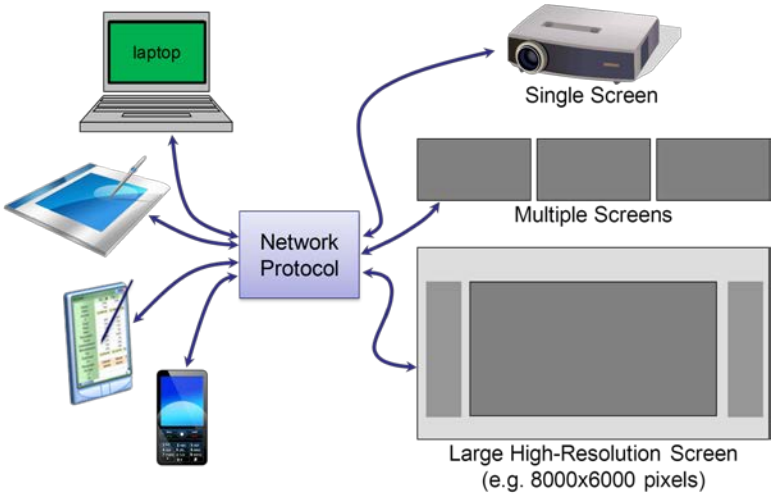


Figure 4:4—SPICE ecosystem. The devices on the left can annex the screen configurations on the right.

4.2 Prior Work

There are many different approaches which allow a user to annex screen space in his environment. These approaches include transmit data, transmit code, transmit the UI, use the web, and brokered connections.

4.2.1 Transmit Data

The user could arrive at a display server and transmit his data to that display for interaction. He could carry that data with him (i.e. with a thumb drive) or transmit the data from a remote location (i.e. a corporate server or website). Dynamo [28] relies on users carrying thumb drives, which is a highly portable approach. Tools such as Windows Roaming User Profiles [30], iRoom [30], Augmented Surface [42], UbiTable [47], MultiSpace [18], MultiBrowsing [25], i-LAND [48], and Roomware [40] each store the user's data on the network and transmit it to a display server for interaction. Tools such as Dropbox [17] or Window Live Sync [32] synchronize data among the user's machines.

Regardless of the approach to delivering data to the display server, the user relies on the display server to have the software necessary to open and augment his data. If the software is not there, the user

cannot view or alter his data. If the software is the wrong version, the user may not be able to open the data on the display server or future machines. For example, a user trying to open a Microsoft Word 2003 document may encounter issues in several ways. If the display server does not have Microsoft Office installed then he cannot alter the document. If the version is 2002, he may not be able to open his document. If the version is 2007, he may not be able to navigate its user interface. Opposite from the user, the room owner has increased costs to install Microsoft Office on that display server and perform regular maintenance which may require purchasing upgrades.

In addition, viruses are easily spread to the display server and other users via this approach. Ultimately, this solution is not effective for nomadic users or room owners.

4.2.2 Transmit Code

A user could carry his data and software to display servers. The software would be transmitted to the display server for execution and the data would be transmitted for alteration. U3-enabled [52] USB drives take this approach.

Unfortunately, U3 software can only run on Windows machines. Using a byte-code language such as Flash [1], Java [22], or Silverlight [31] might make the software cross-platform compatible, but the transmit-code approach has three major failings: speed, consistency, and security.

Applications are typically much larger than the data files they manipulate. This can be seen by examining the size of installation folders on a typical computer. Transmitting the executable and its support files would significantly slow the user experience as the user waits for the application to start up.

Byte-code languages are complex pieces of software which are frequently inconsistent from machine to machine. The volume of tests needed to ensure consistency for all possible uses of the framework are daunting. Consequently, companies producing byte-code frameworks can only target a few operating systems or platforms. Limiting platforms does not encourage wider protocol acceptance.

Most importantly, the display server is given complete control over the user's data and software, which allows the display server to steal or infect the user's data.

4.2.3 Transmit the UI

The user must maintain control over his data, applications, and settings. The only way for him to do this is to ensure that his personal device is the only machine that executes his applications. This personal device could be carried by the user into the room or could be located remotely.

If the personal device is in the room, the user could annex a display server via a technology such as X-Windows (X11) [57], Virtual Network Computing (VNC) [56], or Remote Desktop Protocol (RDP) [68]. If the personal device is located remotely, the personal device could use VNC, X11, or RDP, or use a third-party technology such as GoToMyPC [14] or WebEx [13]. In most cases, the user would enter connection information on the display server to connect to the personal device and bring the UI to the display server.

Regrettably, establishing connections via a display server is an insecure approach. Establishing a connection requires the user to expose his username and password to the foreign display server. After establishing the connection, the display server has complete control over the personal device. Ostensibly the display server will transmit input the user provides. But in reality, the display server can open any application, navigate to any website, or install any software including malware. The display server is not constrained to show the output of the user's personal device, so the malicious activity can remain completely transparent to the user.

If, on the other hand, the user establishes connections from his personal device and never the display server, then his username and password are protected. With a sufficiently flexible application sharing framework, the user can launch and share only the applications he chooses which reduces his

exposure to arbitrary application exploits. In addition, if the display server is treated as an output-only resource, the personal device is protected from potentially malicious input.

Some research environments use VNC (or some variant thereof) to let users collaborate on a shared space. Relying on VNC is a simple, flexible option for creating nomadic environments. Technologies that rely on VNC or some variant include WeSpace [55], IMPROMPTU [10], Lacome [28], WinCuts [49] and research by Wallace et al [53]. Unfortunately, these technologies require a trusted, centralized service, often do not restrict the input coming from the display server, and require more maintenance and setup than homeowners may be willing to invest.

4.2.4 Use the Web

A common approach for nomadic interaction is to use the web or, more colloquially, "live in the cloud." The user travels from place to place logging in to websites wherever he goes. In this paradigm the browser is a key piece of the display server and web sites become the user's personal device. Living in the cloud makes his data portable and his applications remain fairly consistent.

Similar to living in the cloud is using a device like Personal Server [11] where the user's personal device is running the web server. The Personal Server automatically annexes a display server and presents content supplied by the Personal Server.

In essence, using the web is a specialized form of transmitting a UI. Consequently, it carries the same problems as transmitting the UI. For example, the display server can steal the user's credentials, or if those credentials are temporary, it could surreptitiously browse through a website after a user has logged in and steal sensitive information.

In addition, HTML can reduce interaction richness and application familiarity. The only way to provide rich interaction via HTML is to transmit code, be it JavaScript [20] or plug-ins like Flash, Java, or Silverlight. Regrettably, transmitting code re-introduces the problems listed in section 4.2.2. In

addition, if the application requires a plug-in which the display server does not have, then interaction may become impossible.

Display server owners are unlikely to open their displays to plug-ins or JavaScript simply because those routes expose the display server to malware. Browsers are also complicated applications that require a lot of maintenance. These applications are regularly patched to combat exploits and to meet ever-growing standards. Clearly relying exclusively on the web is not sufficiently rich and secure for users, and is not secure enough for display owners.

4.2.5 Brokered Connections

Some research is exploring using a combination of a display server, personal device, and remote machine. The personal device is used to establish a connection between the display server and a remote machine. This is an attempt to protect some of the user's sensitive data (e.g. credentials) from exploitation. This research includes work by Oprea et al [47] and Sharp et al [59].

Mobile Composition [58] extends Oprea's and Sharp's research by splitting a user interface between a web browser and the user's cell phone. The web server and web browser are augmented to support encrypting sensitive data. The web server encrypts sections of the HTML so that only the user's cell phone may decrypt it. The browser is augmented to recognize the encrypted sections of the HTML and forward them to the user's cell phone.

Mobile Composition, Oprea, and Sharp each illustrate an excellent direction for future research. By carrying a personal, trusted computer, the user can have greater control over his credentials. By splitting the UI across machines, like Mobile Composition does, an application can act large on an annexed screen but still protect the user's sensitive data by showing that data on the small trusted device. Unfortunately, these systems rely on the Internet which has significant latency and reduced interactivity.

If the personal device were to supply all the content shared on the display server then the network distance to the display server is short, latency reduces, and interactivity can increase.

The XICE framework implements a brokered approach to screen annexation [6]. Small devices such as cellphones can implement this simple framework to give users a portable, easily accessed, powerful, protected, environment for collaborating. Unfortunately, this framework requires that applications be rewritten to target the XICE windowing toolkit, so it does not provide familiarity for applications the user may already be familiar with. The SPICE Protocol is designed similarly to the XICE framework, and provides UI distribution [6] and a moderate level of privacy [5] for legacy applications.

4.3 The SPICE Protocol

Ultimately, the user needs a way to annex additional display space. This could be via a physical connection or a network connection. In either case, a machine must be available to receive that connection. Because a network connection is more flexible and can support multiple users, the authors advocate for a network protocol.

The network protocol requires software to be available on the display server to interpret commands sent via the protocol. Such a requirement also means that the user must be able to expect any display server to have the appropriate software pre-installed which re-introduces the problem of the user expecting certain software to be pre-installed on the annexable machine. However, the authors argue that expecting a single piece of software is more reasonable than expecting a variety of applications. It is preferable to have a simple protocol which can support a wide variety of devices and applications consistently and easily.

The underlying protocol must be simple so that a wide variety of personal devices and display servers can implement the protocol and so that display servers will require little maintenance. If the

required software is too complex, then the maintenance problems that were illustrated in sections 4.2.1 and 4.2.4 resurface.

This protocol should also be designed to protect both the user's privacy and the display owner's security. As a result, the protocol should not transmit code or user data at all, and connections should be established to any display server without sharing credentials.

The shared visual output must be consistent from machine to machine. Transitions to different media must be immediate. And the user must be able to share the output from existing applications on the shared space.

SPICE meets these protocol requirements through a particular interactive paradigm. The user carries a trusted personal device which executes applications and can annex display servers as output-only resources. The annexing protocol is simple so that conceivably a wide variety of personal devices and display servers can consistently implement it. The protocol is output-only to reduce the space for exploitation. No code is transmitted and no input is accepted. The UI transmission (section 4.2.3) is via a pixel-based scene-graph. The protocol is network-based so it can support multiple simultaneous users. Legacy applications may be shared directly with the display server, while applications that use the SPICE protocol can create more powerful experiences.

SPICE was designed to accomplish the distinct interactions depicted in Figure 4:1 and Figure 4:2. The details of the implementation of the multi-screen presenter and collaborative workspace are discussed in section 4.4. To produce these interactions, the display server is designed to be "dumb and powerful". In other words, the display server implements a simple protocol and has plenty of hard drive, RAM, CPU, and possibly GPU resources.

On the other hand, small devices such as smartphones have limited RAM, processing power, and network bandwidth. One portable device used when developing SPICE was an HTC Tilt [22] (aka HTC

TyTn II), which has a 400 MHz processor, 256 MB of RAM, Wi-Fi access, and runs Windows Mobile 6 [33]. Although rated as 802.11g, the Tilt's radio has a top speed of 5 Mbps which is one tenth the 'g' maximum of 54 Mbps. The authors also used laptops and netbooks for testing, but the Tilt demonstrates the device constraints SPICE supports.

To balance between a low-power device and the high-power display server, SPICE uses a scene-graph to represent an application's UI and caches the scene-graph and the media on the display server so that the personal device only needs to transmit rendering instructions and media once. If a window is required to repaint itself (e.g. in response to a z-order change) the display server uses the scene-graph to repaint that window without a round-trip to the personal device.

The basic render-able data types that SPICE supports are images, audio, video, and streamed images. Images, audio, and video use web standards for their data types. *Streamed images* are basically VNC clients and servers embedded in the scene-graph (as is described in section 4.3.4).

Because SPICE is cross-platform and consistent, it meets the user and room owner needs. For the user, a portable device gives familiar, portable, easy access to data. Annexing display servers allows portable devices to act powerful and collaborative. An output-only protocol also protects the personal device from many potential exploits.

On the other hand, the output-only protocol allows room owners to purchase and install a display server with the confidence that it is unlikely to require maintenance.

This section will proceed by first describing the frameworks used to implement SPICE, then how display-server-side rendering occurs, how media is cached, and finally how streamed images operate.

4.3.1 Implementation Frameworks

SPICE is designed to be a cross-platform protocol. As such it does not use any language- or toolkit-specific remote method invocation (RMI) protocols such as Java RMI [27] or COM [29].

Because the display server cannot initiate connections to the personal device (as explained in detail in section 4.3.3) standard web service protocols are not sufficient, so SPICE does not use them. Instead, SPICE uses a custom, text-based, message-passing protocol with syntax similar to JSON [15]. SPICE messages are compact, well structured, and cross-platform compatible.

To help ensure that SPICE is cross-platform the client and server implementations are available in both Java [22] and C# [53]. Either client version may communicate with either server version, each sending SPICE messages. The C# display server supports rendering video via Windows Presentation Foundation (WPF) [53]. The Java server does not support rendering video. As demonstrated by the two client and two server implementations, the consistent, text-based messages with few operations yield a protocol that is easy to implement on other platforms.

The underlying transport mechanism is HTTP which is a client-to-server message framework. However, as is discussed in section 4.3.3, the display server frequently has messages destined for specific client machines. As a result, the display server queues up messages for each client and delivers the messages any time the client initiates an HTTP GET request.

To track each client, SPICE uses URLs with an embedded session ID. When a client connects for the first time, the server allocates a session object to track that client and redirects the client to the updated URL. The session tracks data about the client including all cached media and pending messages. The session is abandoned if the client does not make an HTTP request within two minutes from the last request.

4.3.2 Rendering a Scene-Graph

The SPICE scene-graph is inspired by other scene-graph technologies such as WPF, Jazz [8], Picolo [8], and XNA [35]. The scene-graph removes the damage-redraw cycle of typical rendering paradigms. The elimination of this cycle frequently reduces network traffic (and thus client-side

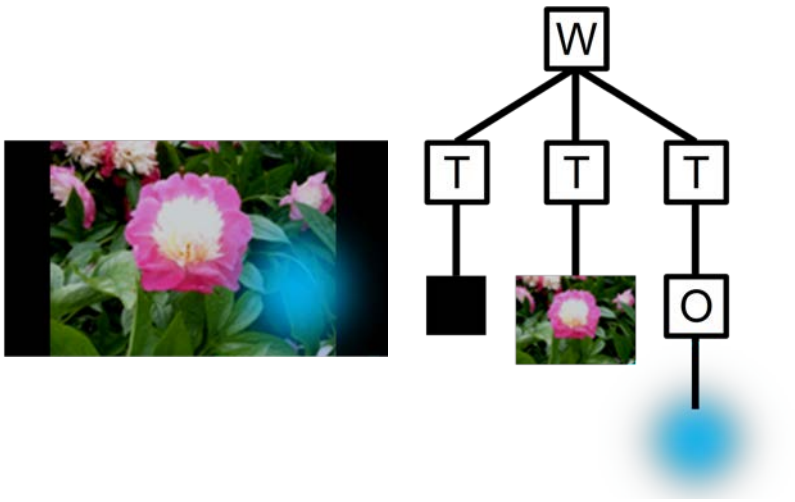
resource requirements) because the display server retains a copy of the scene-graph [5]. Retaining the scene-graph allows the display server to render from that graph instead of querying the personal device any time pixels need to be re-rendered.

The SPICE scene-graph has two broad node types: leaf and container. The leaf nodes—AudioNode, ImageNode, and VideoNode—point at content to render. The container nodes—Container, Clipper, Transformer, and Transparent—can hold zero or more nodes, may be nested, and affect how the child nodes are rendered.

The ImageNode, AudioNode, and VideoNode types each reference a piece of media to render. The AudioNode and VideoNode types also include properties for affecting the playback of their referenced media: whether it is playing, its start position, play speed, volume, and whether it is muted. Advanced manipulation of audio or video such as warping or video editing is not supported.

The Clipper node performs a rectangular clip on rendered children. The Transformer node holds an affine transform that is performed on all child nodes. And the Transparent node renders all child nodes as a single transparent layer.

Figure 4:5 shows an example scene-graph for showing a slide from the botanist's presentation. Part (a) shows the rendered slide while part (b) is the scene-graph for that slide. To gesture at important information, the botanist may, on his phone, tap on regions of the image thumbnail appearing on his phone and a fuzzy blue dot shows up on the annexed display and then fades away. The window has three Transformer nodes under it. The first Transformer node stretches a 2-pixel by 2-pixel black image to fill the entire window. The second Transformer holds the image of the flower and centers and scales it to fit the window. The third Transformer holds a Transparent node (represented with an 'O' for "opacity"). The Transparent node is usually configured to hide the fuzzy blue dot but is dynamically changed to show the dot when the thumbnail is tapped.



(a) (b)

Figure 4:5—Scene-graph. Part (a) shows what is rendered while part (b) is the graph. Transformers position the background, image, and blue highlight dot. The blue dot is in a transparency node (O) that can show or hide the dot.

SPICE separates the media from the nodes rendering the media. For instance, none of the leaf nodes in Figure 4:5 part (b) actually holds an image, but instead hold a reference to an image. This reference is akin to an HTML “IMG” tag pointing to an image instead of embedding it.

Media is referenced with either a URL or a unique identifier. A URL lets the display server retrieve that image from another web server while the unique identifier lets the display server retrieve the media from the personal device.

A few steps are required to transmit media from the client to the display server, as outlined in Figure 4:6. First, the personal device transmits the desired scene-graph to the display server. Then, when the display server is ready to receive the media, the display server returns a message requesting that the personal device post that media to a specific URL (this message is returned in response to a subsequent HTTP GET or POST request). The personal device uses an HTTP POST to send the requested media to the specified URL. Finally, the display server can render that media.

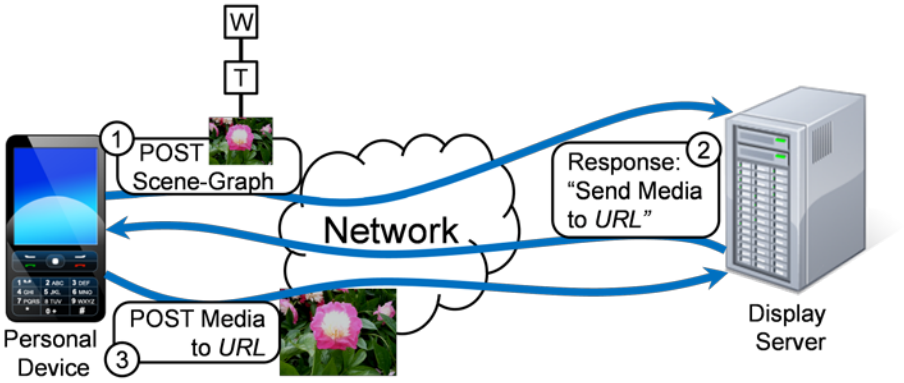


Figure 4:6—Transmitting a scene-graph with media. 1) The scene-graph is transmitted. 2) The display server requests the media. 3) The personal device transmits the media.

Without the media, the scene-graph fits within a 1Kbyte network packet, so transmitting a new scene-graph is instantaneous. Alterations to the scene-graph (such as changing the opacity value on the Transparent node or the affine transform on a Transformer node) are smaller still. On the local network, such changes can be transmitted quickly.

Regrettably, a 5 Mbps transmission speed over HTTP limits the size of transmitted files. An HTTP connection typically times out after 20 seconds. Consequently, the client cannot transmit a multi-megabyte files (e.g. video) without a potential timeout. To compensate for these timeouts, SPICE supports transmitting media in chunks. The personal device chooses what size of chunk is appropriate for its connection speed and transmits the total file size and the location of the current chunk in the HTTP header.

4.3.3 Media Caching

The botanist's pictures are at least 1MB in size. Transmitting each image only when the botanist advances the slide takes about two seconds at 5Mbps. However, the slide transitions need to be instantaneous.

SPICE speeds up these transitions by using two approaches: a scene-graph and preloading media. A scene-graph reduces the personal device's network load when rendering windows and speeds up transmission of a UI by separating out the media. Multiple scene-graphs can reference the same media which helps reduce the transmission needs. But, to gain instant transitions, SPICE allows the client device to explicitly instruct the display server to cache media before it is needed. This second approach is also called *preloading media*.

As shown in Figure 4:7, preloading media to the display server follows an approach similar to transmitting the media for a scene-graph. The only difference is that a "store this media" message is sent to the display server instead of a scene-graph. The "store this media" message contains only the media's identifier (URL or unique identifier). The display server can request the media when it is ready to receive it.

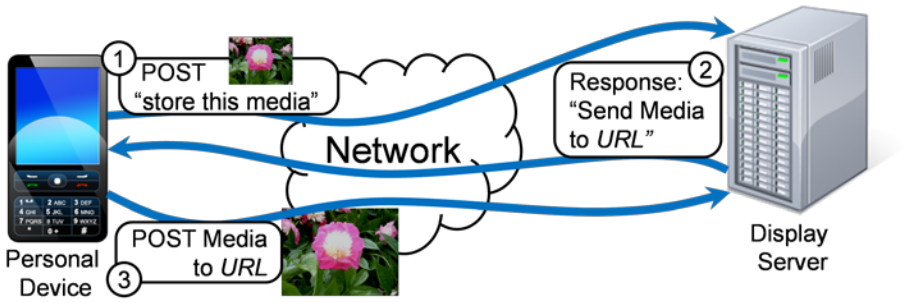


Figure 4:7—Preloading media. 1) The reference is sent in a request to store media on the display server. 2) The display server requests the media. 3) The personal device transmits the media.

When the client no longer needs that media, it may send a "dispose this media" message so that the display server can reclaim the media's consumed space.

A display server may have limited storage space. If the server runs out of space then it may start disposing of media. The display server starts by disposing of unreferenced media, then it disposes of preloaded media. When the display server has enough space again it may re-request the preloaded

media. When the personal device disconnects from the display server all media associated with that connection are disposed.

4.3.3.1 Web Server on the Personal Device

A URL could have been used to reference the media on the personal device, but this approach has some limitations: hosting, authorization, NATs, and single-application devices. First, the personal device may not be able to host a web server due to limitations in the device's platform (e.g. Silverlight does not allow TCP listeners). Second, a web server could allow devices other than the display server to retrieve media from the personal device, which is insecure and reduces the personal device's performance. Third, if the personal device is behind a NAT relative to the display server then the display server cannot initiate a connection to the personal device to retrieve the data via URL. Finally, some portable devices (e.g. the iPhone [3] or Windows Phone 7 Series [34]) can execute only one application at a time (although Apple claims that the iPhone can multitask, it executes only the active application and suspends all non-active applications, but can resume a suspended application quickly which emulates multi-tasking [4, 16]). If the botanist uses such a device to give his presentation and he initiates another application then his presentation application is either suspended or terminated. Either way, the application is usually allowed to resume when he swaps back to it. Ideally, this resumption should be seamless to the user and the display server. Consequently, the display server must continue to operate when a personal device is temporarily disconnected. In addition, the display server cannot expect to initiate connections to the personal device.

NATs and single-application personal devices influenced the design of SPICE to use only client-initiated connections. The display server often has messages it must communicate specifically to the personal device, like the "send media to *URL*" message used to request media from the personal device.

Therefore, the display server queues messages which are delivered the next time the personal device connects to the display server.

By queuing messages and delivering them on the next request from the client, SPICE gets around the limitations of hosting, authorization, NATs, and single-application devices. Because the portable device initiates a connection it does not have to host a web server. Because it does not host a web server, the personal device does not need a specialized authorization mechanism. The NAT is not an issue because the display server does not need to initiate a connection to the personal device. And because messages are queued, the next request from an application on a single-application device can come from the application the next time it is activated.

4.3.4 Streamed Images

Key to the adoption of a protocol such as SPICE is the ability to share legacy applications on the display server. All shareable applications ultimately render to pixels, so SPICE is designed to capture and transmit an application's pixels, similarly to VNC.

SPICE provides a special object called "streamed image" to transmit a sequence of screen scraped pixels from an application on the personal device to the display over an HTTP channel. This sequence of images is intended to be screen captures of a single window, but may be used to transmit any images (with varying degrees of performance).

The streamed image object acts similarly to VNC, with two key differences. First, VNC servers have only one pixel-buffer. This buffer has only enough pixels for the pixels rendered to the screen. In SPICE, each server-side streamed image object has its own pixel buffer and the server software composes those buffers according to the containing scene-graphs and window layouts. Consequently, the VNC "*draw a rectangle of pixel data at this location*" command is only relative to the buffer for that streamed image not the display server's screen. Second, SPICE is an output-only protocol. No data,

including input, is transmitted from the display server via the streamed image object to the personal device.

The client-side software can create a streamed-image object and may immediately start handing images to it. However, the display server must be informed of the streamed image object using the process outlined in Figure 4:6 (with a reference to a streamed image object instead of an image or other media). Then the client-side streamed image object can post images and their changes to the display-server-specified URL for rendering.

Transmitting RAW pixels is a network-intensive process, so SPICE has some heuristics for compressing the transmitted pixels and detecting changes between window captures. SPICE is a research tool and is not intended to compete performance-wise with other VNC clients such as TightVNC [50] or RealVNC [41]. Currently SPICE uses Portable Network Graphics (PNG) [12] to encode all pixels but other encodings are possible and readily accepted. However, the way the streamed image object detects image changes may be useful.

As images (screen captures) are handed to the streamed image object, it detects changes between the current image and the previous image and transmits only the changed regions. This process is efficient enough to reflect a significant change in a 1024x768 window to the display server in less than a half-second on an 802.11g network. With modern multi-core processors, the required CPU overhead rarely impacts the current user interaction.

To detect changes, the streamed image object follows the process outlined in Figure 4:8. The new image in part (b) is used to build histograms in X and Y of changes from the original in part (a). If there are changes, then the histograms are used to find all the possible “change rectangles” within the window, as illustrated by overlapping regions caused by the projections in part (c). Each possible change rectangle is checked to see if it actually has any changed pixels. If there are any changed pixels within a

possible change rectangle, then that change rectangle is transmitted to the display server in PNG format (the upper-right and lower-left rectangles in part (d)).

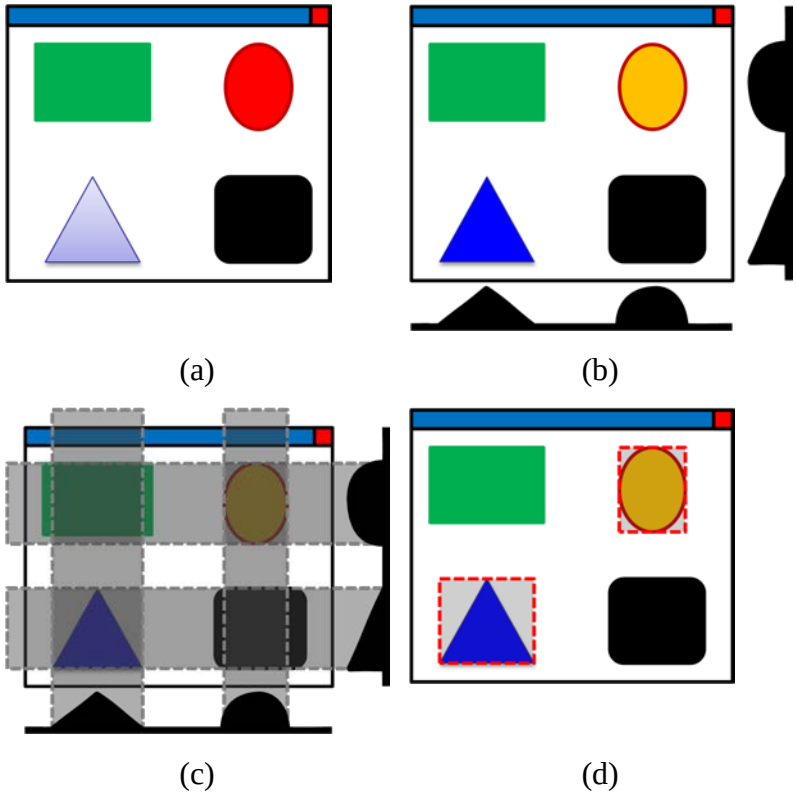


Figure 4:8—Streamed image detecting changes. (a) The original window. (b) The changed image is used to build histograms of change in X and Y. (c) The histograms are used to find all possible change rectangles. (d) Transmit only the rectangles that changed.

Although the streamed image object is intended to accept window captures, it may be used to transmit the entire desktop or a region of a window/desktop. The streamed image object accepts images regardless of the source.

4.4 Applications

This paper will now discuss how the two applications mentioned in section 4.1—and illustrated in depicted in Figure 4:1 and Figure 4:2—are implemented. These applications were created to demonstrate SPICE's simplicity and power. The first application is a multi-screen presenter application

designed to run on a handheld. The second is a collaborative screen sharing application. Neither of the application user interfaces was evaluated as to their appropriateness for users. The purpose of these applications is to demonstrate the SPICE protocol, to show what kinds of powerful interactions users can have, and to demonstrate how SPICE addresses the stated user needs for applications.

Because with SPICE the user carries his computer with him, he automatically has easy access to his applications, which provides familiarity. Portability comes from the kind of device the user chooses as his portable computer. The user's collaborative needs are met because multiple people can display information on the display space simultaneously. The applications demonstrated in sections 4.4.1 and 4.4.2.3 show how the user is provided powerful computing via a much larger screen space. The details of the screen-sharing algorithms in section 4.4.2 show how he is protected when sharing individual windows from his desktop.

4.4.1 Multi-screen Presenter

Several requirements must be met to implement the botanist example. The presentation must support images, audio, and video. The slide transitions must be instantaneous. Different media must be presented on different screens simultaneously. And the presentation must be delivered from a phone like the HTC Tilt whose wireless speeds limit the presentation. Relying on VNC will not produce multiple screens, nor fast slide transitions, nor render video at 30 frames per second. However, with the SPICE protocol, such presentations may be delivered with ease.

Existing multi-screen presentation software has significant requirements on personal devices. For instance, MultiPresenter [26] requires the personal device to be outfitted with additional graphics hardware. MultiPresenter requires a separate VGA port for each screen the user wishes to use. This limits the number of screens the user may annex to the number of physical ports on the personal device. The wireless nature of SPICE imposes no such restrictions.

In the authors' experience presentations consist primarily of images, audio, and video. Converting a presentation to be composed entirely of these primitives is straightforward. Slides showing text and/or pictures must ultimately be rendered to pixels, which can be captured as images. Animations such as slide transitions, if necessary, can be similarly rendered as video.

4.4.1.1 Multi-Screen Presentations

Key to building a multi-screen presentation is assigning media to specific screens in a specific order. Typically a slide transition will change media on only one screen, but some transitions will simultaneously change media on multiple screens. Both these effects must be incorporated into the presentation.

The authors built a rudimentary tool which may be used to arrange a multi-screen presentation, complete with transitions that change multiple screens simultaneously. This tool is called the "authoring tool."

To build a presentation, users must first import media into the authoring tool. As shown in Figure 4:9, the upper-right section of the sequencing mode of the authoring tool houses all the imported media. The user can then lay out the sequence of slides by dragging the media into the sequence bar located at the bottom of the window. If multiple screens change at the same time, then the user must tie that media together on this window. The last two slides in the sequence in Figure 4:9 have been tied together as signified by the double arrow between them. All other media shows up one at a time, so they are not tied together.

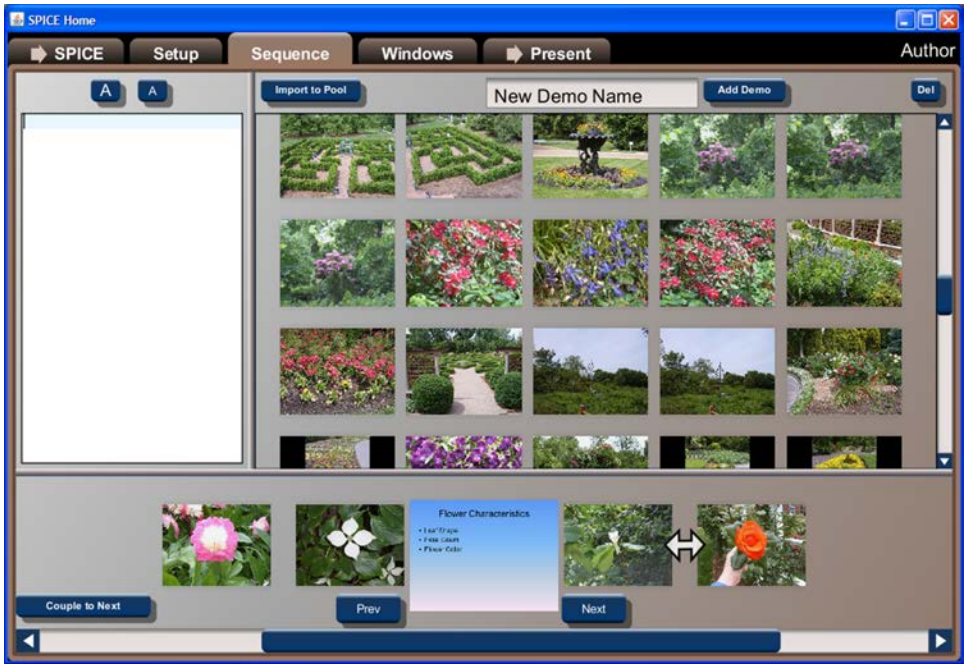


Figure 4:9 –Arranging slide sequence in a multi-screen presentation. The user imports media, including slides, into the upper-right section of this window. Then the user can drag each image into the sequence region at the bottom of the window. The user may also tie two slides together (which is indicated by the arrow between the last two slides). Tied slides appear simultaneously during a presentation.

After sequencing slides, the user must assign each piece of media to a particular screen. When the authoring tool is in the next mode, as shown in Figure 4:10, the user assigns each media to a specific window. Across the middle of the screen is a miniaturized view of the screen arrangement within the conference hall. The user simply clicks a screen and then the media selected in the bottom-half of the window is tied to that screen.

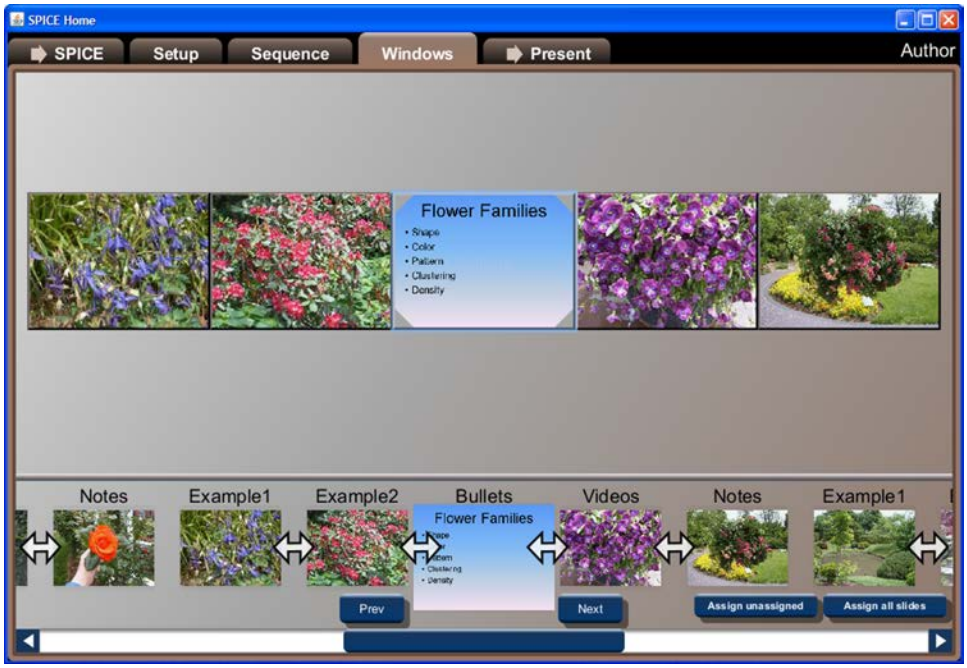


Figure 4:10 –Assigning slides to screens. The user must assign each slide to show on a specific screen. The slides are listed along the bottom. The screens are shown across the main section and illustrate how the presentation will appear.

When giving a presentation, the user operates the "presentation software" shown in Figure 4:11. The user can advance slides normally by clicking on the "next" button at the bottom or clicking the subsequent slide. If multiple slides are tied together, they advance as a group and all the screens update accordingly.

Figure 4:11 –Giving a multi-screen presentation. A world-in-miniature of the presentation is at the top. The slide sequence is along the bottom. To advance media, the user just clicks on the "next" button or the next media in the sequence which is right of the center.

4.4.1.2 Presenting From a Handheld

The presentation software in Figure 4:11 is intended to be run from a laptop. However, lugging around a laptop to give a presentation may be cumbersome. In such cases the user may find it more convenient to carry around a handheld device such as a smartphone.

The authors created presentation software for a Windows Mobile smartphone. This software was created using C# and the .NET Compact Framework [56]. Because of the screen and input constraints, the UI on the smartphone is different from the laptop version of the presentation application.

To deliver a presentation, the user interacts with the UI shown in Figure 4:12. Showing a world-in-miniature on the small screen does not provide a useful interface on the smartphone. Instead, the UI shows the current slide as the largest image on the screen with the previous slide below and to the left

and the next slide below and to the right. To advance the slide the user can tap on the current slide, tap on the next slide, or press right on the phone's hardware directional pad.

Figure 4:12 – Multi-screen presentation from a smartphone. The current slide is shown at the top, the previous in the bottom left, and the next in the bottom right. The user navigates by tapping on the previous or next slides, or by using the hardware directional pad.

However, delivering media from the phone is not as fast as delivering from the laptop because the wireless radio does not transmit information as quickly. To make sure the slide transitions are instant, the smartphone's presentation software uses SPICE to preload media to the display server using the process outlined in Figure 4:7. The presentation software registers the media in the order in which they occur in the presentation. The display server is designed so that it requests the registered media in the order in which the "store this media" messages are received. With slide resolutions of 1024x768, each slide consumes approximately 1 MB of space and is transmitted in roughly two seconds at 5 Mbps. A 30-slide presentation is completely preloaded within a minute, and if the user transitions approximately once every 15 seconds (which is unusually quick) the presentation is preloaded and can

transition instantly by the time the user gets to the second slide. Even if the user is giving a presentation on five screens and all five screens change images with a transition, the presentation transitions instantly for the first slide transition and has already preloaded three more slides.

Whether the presentation is delivered from a cellphone or a laptop, the opportunity to give a presentation that covers multiple screens gives a much richer presentation experience for the presenter and the audience. SPICE facilitates rich interaction by preloading the media so that the presentation runs more smoothly than a transmit-on-demand solution would and runs with fewer resources than a hardware-based solution would.

4.4.2 Collaborative Screen Sharing

Multiple users need to be able to share content from existing applications on the same display space. These users may have heterogeneous devices, such as laptops, netbooks, tablets, that are supplied by a variety of manufacturers (PC or Mac). They need to quickly and cleanly share only the windows they choose. They need to be able to rearrange windows, even if they do not have access to the display server's input hardware. VNC cannot handle multiple clients, and as WeSpace [55], IMPROMPTU [10], Lacome [28], WinCuts [49], and Wallace et al [53] have shown, a custom display space management tool is necessary.

The *screen-sharing application* is designed to facilitate discussions like the one in Figure 4:2. It is built using Java, and although it has Java Native Interface (JNI) [27] hooks for some features, those hooks have been designed for both PC and Mac so devices from both manufacturers may be used at the same display space. The screen-sharing application is similar to applications developed by Wallace and by WinCuts. However, the screen-sharing application has more rigorous privacy controls.

In the screen-sharing application, a user can choose to share his entire screen, a region of his screen, or an individual window. The screen-sharing application then captures the pixels from the

specified area and uses the streamed image object to transmit those pixels to the display server. Using multiple streamed image objects on a client allows a client to share multiple streams of pixels.

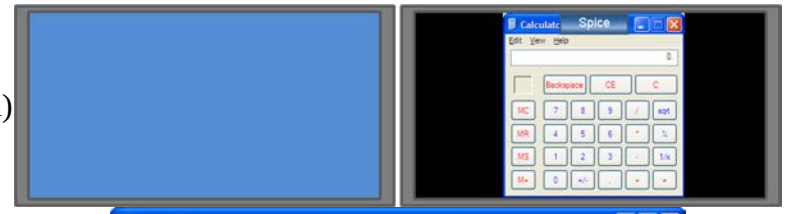
4.4.2.1 Screen Sharing

The process for users to share windows is straightforward. Figure 4:13 shows the "Share" window which the screen-sharing application provides for all top-level windows. As the user moves his mouse within a top-level window, the Share window appears. Moving the cursor near the Share window makes it interactive. It supplies menu options that let the user share the associated top-level window or access the screen sharing application ("Click for controls").

Figure 4:13—Directly sharing windows. Part (a) shows the transparent Share window that appears for all top-level windows. As the mouse cursor approaches the window, the window becomes opaque like in part (b) and the first option under that window lets the user share that window.

After the user has shared a window, he may move or resize that window on the display server via the "world in miniature" controls on his personal device. Figure 4:14 (a) illustrates a dual-screen display server on which a user has shared the "calculator" application. The user has chosen to enlarge the window on the display server so other people in the room can easily see his calculations. The screen-sharing application maintains the calculator's aspect ratio, and fills in the space around the calculator with a black background. Using the controls in Figure 4:14 (b) which are provided on the client machine, people may resize or reposition the space allocated on the display server.

(a)



(b)

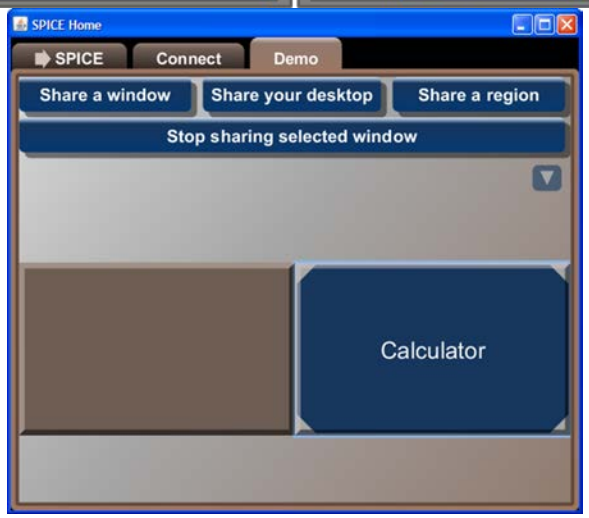


Figure 4:14—Rearranging windows on the display server. The user shares his Calculator application. He can reposition and resize that remote window via the screen-sharing application in part (b).

To help protect the personal device from potential infection, SPICE is an output-only protocol. Display servers are not expected to provide input, and as such, the SPICE protocol does not support transmitting input to the personal device. If a user shares a window then he is the only person who may interact with that window. If the user unknowingly connects to a display server harboring malware his personal device is protected because the personal device does not accept input from the display server and the protocol does not support transmitting code.

4.4.2.2 Window-Specific Sharing

Users need to share only the windows they choose. Typical screen sharing techniques, such as those used by Wallace or VNC, scrape pixels from the portable device and show them on the annexed screen. Unfortunately, if a notification window like the one in Figure 4:15 appears other people in the room can see it. To avoid this potentially embarrassing situation, the screen-sharing application includes

a heuristic which avoids transmitting pixels generated by overlapping windows. Although this sub-section details how the overlap-avoiding heuristic operates, it is important to note that this heuristic is not a feature of SPICE but a feature of the screen-sharing application which is built on top of SPICE.

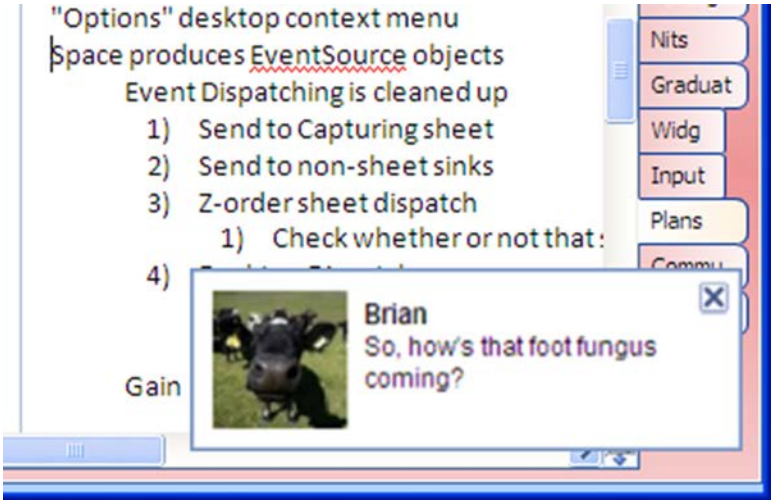


Figure 4:15 – An embarrassing Instant Message that may be received during a collaborative session.

The screen-sharing application tracks window positions using JNI hooks. The JNI hooks register with the operating system's accessibility features and track the position and z-order of every window on the screen. (There are separate implementations for Windows and for Mac.) As a window is repositioned or resized, the screen-sharing application tracks the window's updated bounds and captures pixels from the updated bounds.

While a shared window is the active window (i.e. the window with keyboard focus) the screen-sharing application uses screen scraping to capture that window's pixels. When that window is no longer the active window, the screen-scraping application no longer captures that window's pixels.

To protect the user's privacy, pixels from unshared windows are not given to the streamed image object and are thus not transmitted to the display server. For example, to protect the user, the pixels from the IM in Figure 4:15 should not be transmitted. Figure 4:16 shows the pixels transmitted to the display server in lieu of the IM's pixels. Using the JNI hooks the window sharing application detects the IM

window overlapping the shared window. Then the screen sharing application transmits grayed-out pixels from the prior unobstructed capture.

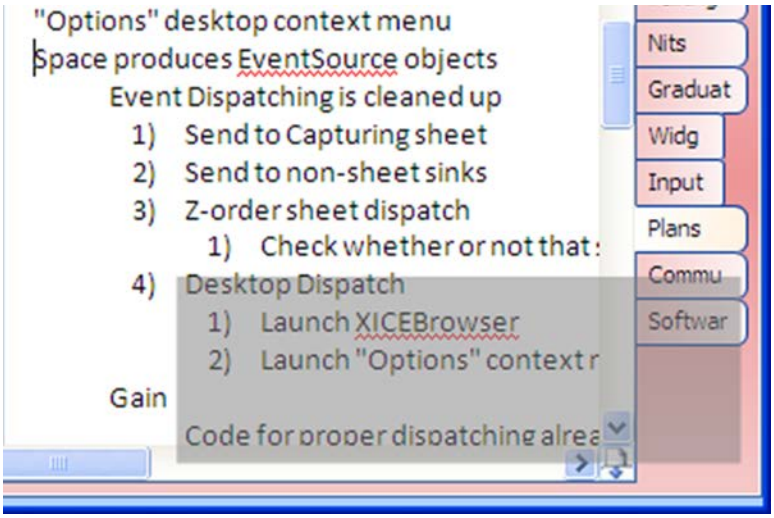


Figure 4:16—Protecting sensitive pixels. When a popup window overlaps a focused shared window on the personal device the screen-sharing application transmits the previous capture with the overlapped pixels grayed out.

Regrettably, notifications for window movements and repaints are not synchronous. Time passes between a window's hiding/destruction and when its vacated pixels are repainted. Consequently, if the IM in Figure 4:15 closes then the screen sharing application would be unaware that those pixels are from a dead window and would transmit them to the display server. In an attempt to prevent most cases of this incidental capture and transmission of pixels from an unshared window, each unshared window's bounds are tracked from the prior screen capture cycle and used to filter out potentially damaged pixels.

Another potentially problematic situation occurs with the repaint cycle happening too quickly. The screen-sharing application detects window bounds before it captures the pixels and then filters out overlapping windows. If a window is created and painted between the time the screen-sharing application detects the windows' bounds and captures pixels, then an unshared window's pixels would be transmitted. To guard against this situation, the bounds of overlapping windows are collected before and after a screen capture. Any overlaps from before or after are not transmitted.

With these window-tracking controls in place, the user's sensitive data is almost-always well-protected. This may fail in rare cases where the shared application freezes and does not re-render itself in a timely fashion. In lab experiments with a few different machines, tracking window bounds to within 200 milliseconds is sufficient to catch these potential private-pixel breaches.

However, with tighter integration with a compositing window manager, such as Windows Vista or Mac OS X, the window-tracking and repaint problems can be avoided. With a compositing window manager pixels would be captured from the shared window's hidden frame buffer instead of the screen's frame buffer, which avoids the problem of potentially sensitive pixels altogether.

4.4.2.3 De-multiplexing pixels

A new interactive technique becomes possible for a single user employing the screen-sharing application with multiple screens. This technique is called *de-multiplexing pixels*, and gives users a much larger interactive surface for an existing laptop, netbook, or tablet PC.

Windowing systems give users more screens space by overlapping windows. For example, the window layout illustrated in Figure 4:17 shows 4 overlapping windows. Most of the pixels in the middle of the screen are really owned by 4 windows. The windowing toolkit resolves this situation by showing only the top-most window's pixels. This is really a form of multiplexing those pixels so that they may be used by multiple windows.

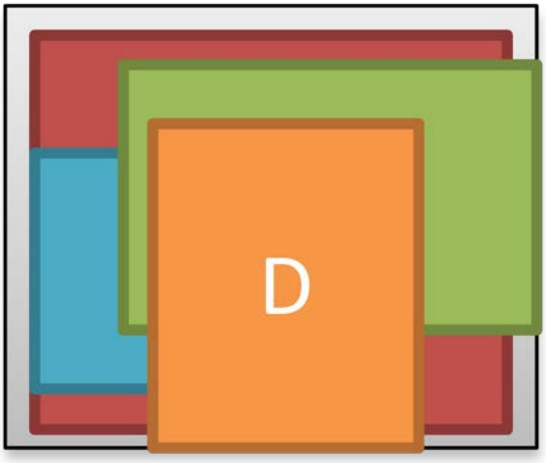


Figure 4:17—Window arrangement on the personal device.

A preferable situation would be to spread out the windows in Figure 4:17 across a wall of displays as illustrated in Figure 4:18. If windows can be spread out then a user can arrange his windows into consistent locations on an annexed display and look at them simultaneously instead of switching among those applications. Simultaneously viewing multiple windows helps the user add context. Adding context is an important feature for users that have additional screen space [7, 8, 11, 21].

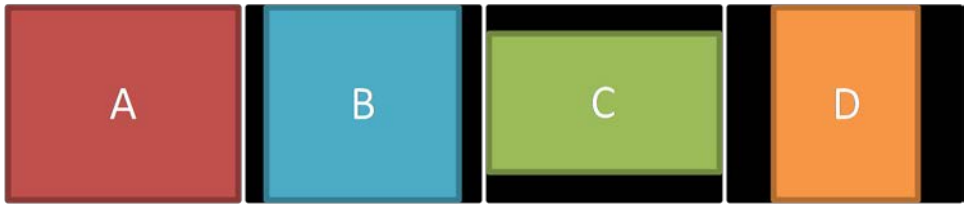
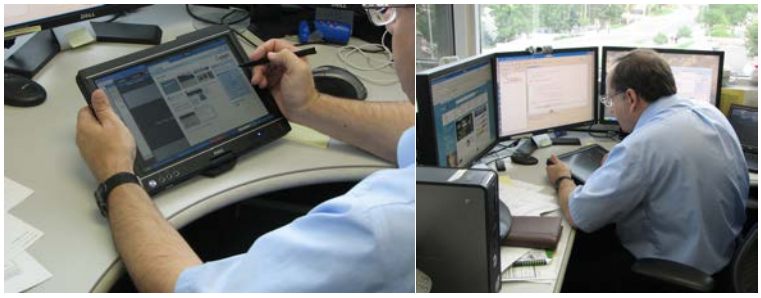


Figure 4:18—Window arrangement on the annexed screens. Windows are spread out so the user can see each one simultaneously.

The screen-sharing application de-multiplexes pixels without any special configuration. It allows the user to take overlapping windows on his personal device and spread them out on the annexed screens. A user with a multi-screen display server on his desktop simply shares windows to that display server and arrange them as he likes. By default, the screen-sharing application attempts to share one window per screen so arranging directly may not be necessary.

The user in Figure 4:19 is de-multiplexing pixels at his desktop. All interaction with his applications is through his portable computer (part (a)), but he can see all of his important windows at the same time on his desktop monitors (part (b)). These monitors provide context while he works.



(a) (b)

Figure 4:19—An individual at his desk. (a) His tablet PC executes all of his applications, but (b) he replicates the output of some windows to his desktop monitors to provide context while he works.

Consider a computer science instructor teaching an introductory class using her laptop. She would like to show API documentation on one screen while working with an Interactive Development Environment on another screen. If a classroom is outfitted with SPICE, the user simply shares the web browser with the documentation to one screen and the IDE to the other screen. She can interact with her laptop as she normally would, but the students have more context as she teaches. This setup is illustrated in Figure 4:20.

(a)

(b)

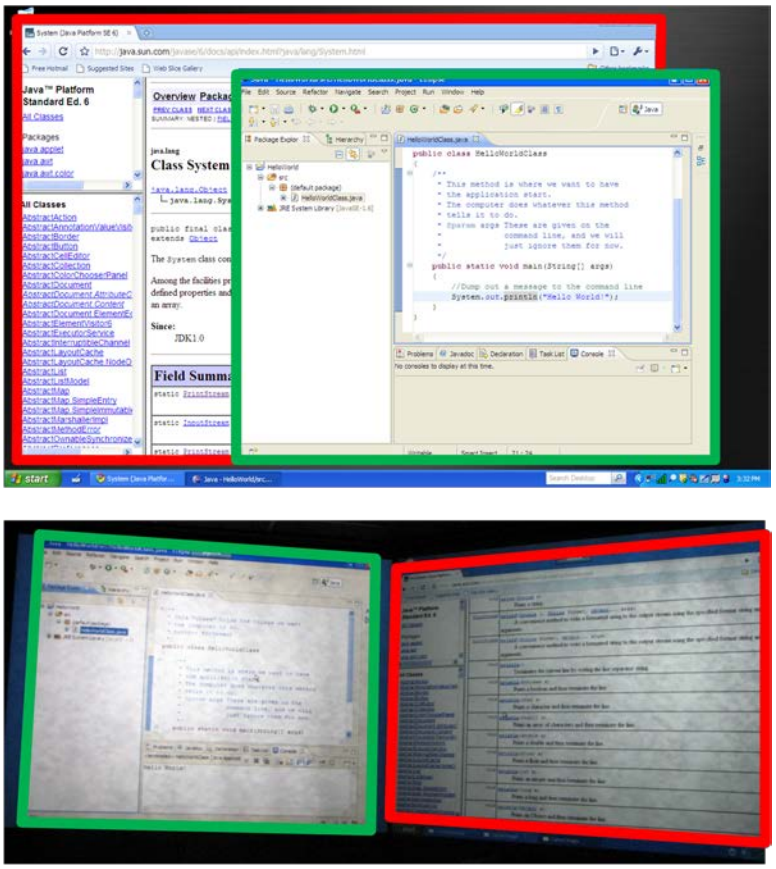


Figure 4:20—A professor uses two projectors in her classroom. Part (a) shows what she sees on her laptop's screen. Part (b) shows what her students see on the projectors.

4.5 Summary

This paper introduces the SPICE framework for annexing screens and sharing application windows. SPICE's power is demonstrated with a presentation application that runs on a smartphone and a screen-sharing application that runs on laptops, netbooks, and tablets that run the full Microsoft Windows OS or Mac OS X.

SPICE provides a consistent communication protocol based on standard HTTP techniques. The protocol has been implemented in Java and C# on multiple platforms and may therefore be used by a variety of personal devices and display servers. SPICE also supports the standard MIME types for images, audio, and video, along with streamed images and a small set of plain text command messages.

The display server software supports concurrent connections from multiple personal devices. With concurrent connections, multiple users can show content on the same screen. These users can collaborate by interweaving shared windows from different applications and different users.

The screen-sharing application protects user privacy by only showing pixels from windows that the user explicitly shares. Any pixels generated from overlapping windows are not serialized to the annexed screen. Instead, pixels from a prior, un-overlapped screenshot are serialized.

4.6 REFERENCES

1. Adobe Systems, Adobe Flash, http://get.adobe.com/flashplayer/, 1996, accessed June 2010.
2. Apple Computer, iPad, http://www.apple.com/ipad/, accessed September 2010.
3. Apple Computer, iPhone, http://www.apple.com/iphone/, accessed July 2010.
4. Apple Computer, What's New in iOS 4, http://developer.apple.com/technologies/ios/whats-new.html, 2011, Accessed April 2011.
5. R. Arthur and D.R. Olsen, 2011. Privacy-aware Shared UI Toolkit for Nomadic Environments. Software: Practice and Experience. May 2011, 28 pages, DOI=10.1002/spe.1085 http://dx.doi.org/10.1002/spe.1085.
6. R. Arthur and D.R. Olsen, 2011. XICE windowing toolkit: Seamless display annexation. *ACM Transactions on Computer-Human Interaction.* 18, 3, Article 14 (August 2011), 46 pages. DOI=10.1145/1993060.1993064 http://doi.acm.org/10.1145/1993060.1993064.
7. P. Baudisch, N. Good, V. Bellotti, and P. Schraedley, Keeping things in context: a comparative evaluation of focus plus context screens, overviews, and zooming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Changing Our World, Changing Ourselves* (Minneapolis, Minnesota, USA, April 20 - 25, 2002). CHI '02. ACM, New York, NY, 259-266.
8. P. Baudisch, N. Good, and P. Stewart, Focus plus context screens: combining display technology with visualization techniques. In *Proceedings of the 14th Annual ACM Symposium on User interface*

Software and Technology (Orlando, Florida, November 11 - 14, 2001). UIST '01. ACM, New York, NY, 31-40.

9. Bederson, B. B., Grosjean, J., Meyer, J., "Toolkit Design for Interactive Structured Graphics." *Software Engineering*, IEEE 2004, 535-546.
10. J.T. Biehl, W.T. Baker, B.P. Bailey, D.S. Tan, K.M. Inkpen, and M. Czerwinski, Impromptu: a new interaction framework for supporting collaboration in multiple display environments and its field evaluation for co-located software development. In *Proceeding of the Twenty-Sixth Annual SIGCHI Conference on Human Factors in Computing Systems*(Florence, Italy, April 05 - 10, 2008). CHI '08. ACM, New York, NY, 939-948.
11. X. Bi and R. Balakrishnan, Comparing usage of a large high-resolution display to single or dual desktop displays for daily work. In *Proceedings of the 27th international Conference on Human Factors in Computing Systems* (Boston, MA, USA, April 04 - 09, 2009). CHI '09. ACM, New York, NY, 1005-1014.
12. T. Boutell, T. Lane et. al., PNG (Portable Network Graphics) Specification, *W3C Recommendation*, October 1996, RFC 2083, Boutell.Com Inc., January 1997.
13. Cisco Systems, Inc. WebEx, http://www.webex.com/, 1997, accessed June 2010.
14. Citrix Systems, Inc. Citrix Online, http://www.citrixonline.com/, 1997, accessed June 2010.
15. D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON). http://tools.ietf.org/html/rfc4627, July 2006.
16. J. Diaz, How Multitasking Works in the New iPhone OS 4.0, Gawker Media, http://gizmodo.com/#!5512656/, April 2010, Accessed April 2011.
17. Dropbox, http://www.dropbox.com/, accessed July 2010.
18. K. Everitt, C. Shen, K. Ryall, and C. Forlines, MultiSpace: Enabling Electronic Document Micro-mobility in Table-Centric, Multi-Device Environments. In *Proceedings of the First IEEE international Workshop on Horizontal interactive Human-Computer Systems* (January 05 - 07, 2006). TABLETOP. IEEE Computer Society, Washington, DC, 27-34.
19. D. Flanagan, *JavaScript: the Definitive Guide.* O'Reilly Media, Inc., 2006.

20. J. Gosling, B. Joy, G. Steele, and G. Bracha, 2000 *Java Language Specification, Second Edition: the Java Series*. 2nd. Addison-Wesley Longman Publishing Co., Inc.

21. J. Grudin, Partitioning digital worlds: focal and peripheral awareness in multiple monitor use. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Seattle, Washington, United States). CHI '01. ACM, New York, NY, 458-465.

22. HTC Corporation, HTC TyTn II http://www.htc.com/www/product/tytnii/overview.html, accessed July 2010.

23. S. Izadi, H. Brignull, T. Rodden, Y. Rogers, and M. Underwood, Dynamo: a public interactive surface supporting the cooperative sharing and exchange of media. *User Interface Software and Technology* (UIST '03), ACM 2006, 159-168.

24. B. Johanson, A. Fox, and T. Winograd, The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. *IEEE Pervasive Computing* 1, 2 (Apr. 2002), 67-74.

25. B. Johanson, S. Ponnekanti, C. Sengupta, and A. Fox, Multibrowsing: Moving Web Content across Multiple Displays. In *Proceedings of the 3rd international Conference on Ubiquitous Computing* (Atlanta, Georgia, USA, September 30 - October 02, 2001). G. D. Abowd, B. Brumitt, and S. A. Shafer, Eds. Lecture Notes in Computer Science, vol. 2201. Springer-Verlag, London, 346-353.

26. J. Lanir, K.S. Booth, and A. Tang, MultiPresenter: a presentation system for (very) large display surfaces. In *Proceeding of the 16th ACM international Conference on Multimedia* (Vancouver, British Columbia, Canada, October 26 - 31, 2008). MM '08. ACM, New York, NY, 519-528.

27. S. Liang, 1999 Java™ Native Interface: Programmer's Guide and Specification, Prentice Hall.

28. X. Liu, Lacome: a Cross-Platform Multi-User Collaboration System for a Shared Large Display, Computer Science, University of British Columbia, 2007. http://hdl.handle.net/2429/378

29. Microsoft Corporation, COM: Component Object Model Technologies http://www.microsoft.com/com/default.mspx, accessed July 2010.

30. Microsoft Corporation, Roaming User Profiles, http://msdn.microsoft.com/en-us/library/bb776897(VS.85).aspx, accessed August 2010.

31. Microsoft Corporation, Silverlight, http://www.microsoft.com/silverlight/, accessed June 2010.

32. Microsoft Corporation, Windows Live Sync, https://sync.live.com/, accessed July 2010.

33. Microsoft Corporation, Windows Mobile 6.5, http://msdn.microsoft.com/en-us/library/bb158486.aspx, accessed July 2010.

34. Microsoft Corporation, Windows Phone 7 Series, http://www.windowsphone7.com/, accessed July 2010.

35. Microsoft Corporation, Microsoft XNA, http://www.xna.com/, accessed July 2010.

36. Netflix, Inc., http://www.netflix.com/, accessed July 2010.

37. D.R. Olsen, Interacting in Chaos, in *Interactions*, ACM (1999), pp 42-54.

38. A. Oprea, D. Balfanz, G. Durfee, and D.K. Smetters, Securing a remote terminal application with a mobile trusted device, *Computer Security Applications Conference, 2004. 20th Annual* , vol., no., pp. 438-447, 6-10 Dec. 2004.

39. C. Petzold, Applications = Code + Markup: A Guide to the Microsoft Windows Presentation Foundation, Microsoft Press 2006.

40. T. Prante, N. Streitz, and P. Tandler, Roomware: Computers Disappear and Interaction Evolves. *Computer* 37, IEEE, 12 (Dec. 2004), 47-54.

41. RealVNC Limited, RealVNC, http://www.realvnc.com/, accessed August 2010.

42. J. Rekimoto and M. Saitoh, Augmented surfaces: a spatially continuous work space for hybrid computing environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*: the CHI Is the Limit (Pittsburgh, Pennsylvania, United States, May 15 - 20, 1999). CHI '99. ACM, New York, NY, 378-385.

43. T. Richardson, Q. Stafford-Fraser, K.R. Wood, and A. Hopper, Virtual Network Computing, *IEEE Internet Computing*, Vol. 2, No. 1, 1998.

44. R.W. Scheifler and J. Gettys, The X Window System, *ACM Transactions on Graphics*, vol 5(2), (April 1986), pp 79-109.

45. R. Sharp, A. Madhavapeddy, R. Want, and T. Pering, Enhancing web browsing security on public terminals using mobile composition. In *Proceeding of the 6th international Conference on Mobile*

Systems, Applications, and Services (Breckenridge, CO, USA, June 17 - 20, 2008). MobiSys '08. ACM, New York, NY, 94-105.

46. R. Sharp, J. Scott, and A.R. Beresford, Secure Mobile Computing via Public Terminals. In *Proceedings of the International Conference on Pervasive Computing* (PerCom 2006). IEEE Computer Society, 238-253.

47. C. Shen, K. Everitt, and K. Ryall, UbiTable: Impromptu Face-to-Face Collaboration on Horizontal Surfaces. UbiComp 2003, Springer Berlin/Heidelberg, 2003, pp 281-288.

48. N.A. Streitz, J. Geißler, T. Holmer, S. Konomi, C. Müller-Tomfelde, W. Reischl, P. Rexroth, P. Seitz, and R. Steinmetz, i-LAND: an interactive landscape for creativity and innovation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*: the CHI Is the Limit (Pittsburgh, Pennsylvania, United States, May 15 - 20, 1999). CHI '99. ACM, New York, NY, 120-127.

49. D.S. Tan, B. Meyers, and M. Czerwinski, WinCuts: manipulating arbitrary window regions for more effective use of screen space. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems* (Vienna, Austria, April 24 - 29, 2004). CHI '04. ACM, New York, NY, 1525-1528.

50. Tight Group, TightVNC, http://www.tightvnc.com/, accessed August 2010.

51. B. Tritsch, Microsoft Windows Server 2003 Terminal Services, Microsoft Press.

52. U3 LLC., U3 Launchpad, http://www.u3.com/ (dead link), see http://en.wikipedia.org/wiki/U3, accessed July 2010.

53. G. Wallace and K. Li, Virtually Shared Displays and Input Devices. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference* (Santa Clara, CA, June 17 - 22, 2007). J. Chase and S. Seshan, Eds. USENIX Association, Berkeley, CA, 375-379.

54. R. Want, T. Perins, G. Danneels, M. Kumar, M. Sundar, and J. Light, The Personal Server: Changing the way we think about Ubiquitous Computing. *Ubiquitous Computing* (UbiComp '02), Springer Verlag, (2002).

55. D. Wigdor, H. Jiang, C. Forlines, M. Borkin, and C. Shen, WeSpace: the design development and deployment of a walk-up and share multi-surface visual collaboration system. In *Proceedings of the*

27th international Conference on Human Factors in Computing Systems (Boston, MA, USA, April 04 - 09, 2009). CHI '09. ACM, New York, NY, 1237-1246.

56. A. Wigley, D. Moth, and P. Foot, *Mobile Development Handbook*, Microsoft Press, 2007.

Chapter 5 Window Brokers: Collaborative Display Space Control

Richard B. Arthur, Dan R. Olsen, Jr.

Computer Science Department

Brigham Young University

startether@startether.com, olsen@cs.byu.edu

ABSTRACT

As users travel from place to place, they can encounter *display servers*; machines which supply a collaborative content-sharing environment. Users need a way to exhibit control over how content is arranged on these display spaces. The software for controlling these display spaces should be consistent from display server to display server. However, display servers could be controlled by institutions which may not allow for the control software to be installed. This paper introduces the window broker protocol which allows users to carry familiar control techniques on portable personal devices and use the control technique on any display server without installing the control software on the display server. This paper also discusses how the window broker protocol mitigates some security risks that arise from potentially malicious display servers.

5.1 INTRODUCTION

People are nomadic and need to collaborate. Modern collaboration regularly involves discussing data. Portable computers are often a source of that data because a user can guarantee that her data, software, and settings are always available wherever she is located.

An increasingly common form of collaboration is via a large annexable shared display space. When a user annexes a screen, she may then share individual windows to that screen so that others in the room may view and discuss the contents of those windows. To annex the screens and share content in the most flexible way requires a network UI distribution framework. Such a framework necessitates a

network-connected dedicated computer called a *display server*. Ideally, these display servers (regardless of manufacturer) would use a consistent annexation protocol so that any *client machine* may annex them.

Figure 5:1 illustrates a situation where three users wirelessly connect to a display server that controls a single display space. Each user is sharing at least one window on that space.

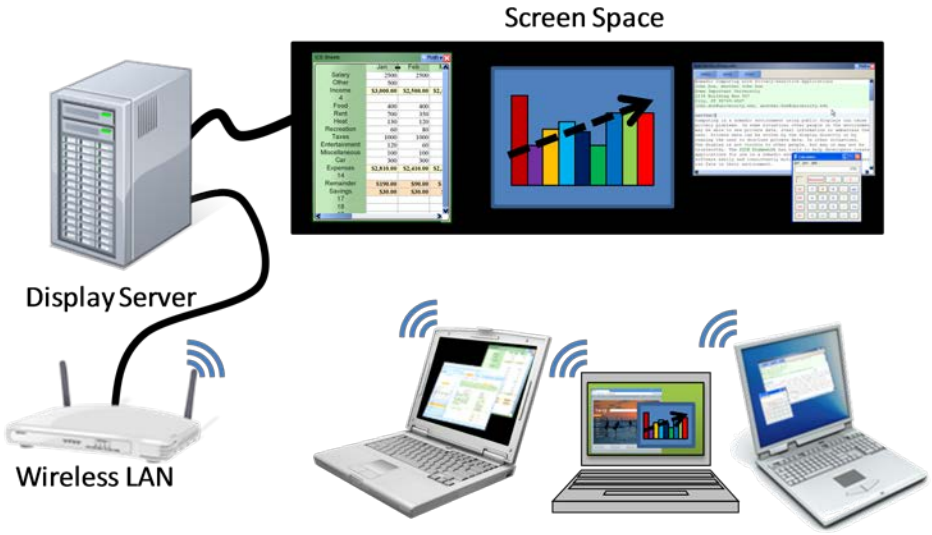


Figure 5:1—Three users bring their laptops and annex a display space to share and discuss data. Via software controls on their laptops, users may rearrange the remote windows.

Problems arise in how the display server coordinates shared windows. For instance, should the person in the middle be allowed to enlarge his window and overlap one or both of the other users' windows? The answer to this question depends on the situation. The answer may be 'yes' if the users are the only ones in the room, but 'no' when there are other people in the room that would have difficulty tracking the discussion if windows overlapped. Or the answer may be 'no' if the users are dividing the display space, but 'yes' if the users are comparing windows.

Users could rely on pure social coordination (e.g. saying "please don't do that") to take care of conflicts, which may work well for a close-knit group of people. Unfortunately, the social coordination

approach is slower and is prone to abuse. Some users need an automated approach to controlling a display space.

Users also need consistent control. The display space may be located in a room controlled by a trusted institution (e.g. the employer), a room controlled by a less-trusted institution (e.g. a conference hall), or a room controlled by an un-trusted institution (e.g. a competitor's conference room). In addition, coordinating logins and control before interacting with a display space is counter-productive to quickly establishing a collaborative group. Consequently, this control may need to be exhibited anonymously. Users need to be able to quickly and smoothly exhibit control on display spaces they may encounter only once and which they may not trust.

This paper is about how to manage and flexibly enforce all the different control paradigms a user might need in a variety of collaborative environments.

5.1.1 Motivating Examples

There are an infinite number of collaborative situations users could encounter. For the discussions in this paper, the following four example situations are helpful in understanding these issues: presenter, discussion, panel, and visitor. This paper uses these situations to illustrate six different *control techniques* for managing the display space.

In the *presenter situation*, someone gives a presentation to his teammates. This person is in control of the presentation software and uses a control technique which shows only his windows. When he opens the discussion to his teammates, one team member has some relevant data she would like to share. The presenter opts to accept and show her shared window on the display space. In this situation the presenter is in control and no one else may show content without the presenter's permission. This situation may also occur in a classroom or conference presentation.

The *discussion situation* consists of a group of people working on a common project. Depending on the group members and the kind of discussion there are multiple ways to digitally assist the discussion. The authors have chosen to discuss the following four control techniques (a subset of the possible options): free-form, moderated control, personal space, and tiled. The *free-form technique* is when window placement is un-constrained. Anyone may place any window anywhere. The *moderated control technique* is when one user is delegated as the "moderator" and she is the only person who may rearrange windows on the display space. The *personal space technique* is one where the display space is partitioned so that each participant has their own section of the display. Windows may reside only within the partition assigned to the window's owner. The *tiled technique* is one where windows may be placed anywhere on the display space, but if the window overlaps any other windows, the overlapped windows are moved out of the way or shrink in size so that no two windows overlap. The tiled technique manages the display space similarly to the Flatland [12] whiteboard management tool—if one window's movement would overlap another window, the other window is pushed away or, if there is insufficient room to move, shrunk or hidden.

In the *panel situation*, a panel discussion is held at a conference. Several guests are invited to be on the panel and one person is the moderator. The discussion moderator annexes the screen via her phone. Her phone has software installed which she can use to choose which of the guests has the floor. The *floor control technique* will only show windows owned by the guest who has the floor. Windows from other guests are hidden. This result is illustrated in Figure 5:2. Assume Alice is currently granted floor control. When the moderator, Kathy, changes the floor control to Geoff, then all of his windows are shown and all of Alice's windows are hidden. Geoff may then show any content he chooses using whatever software he owns.

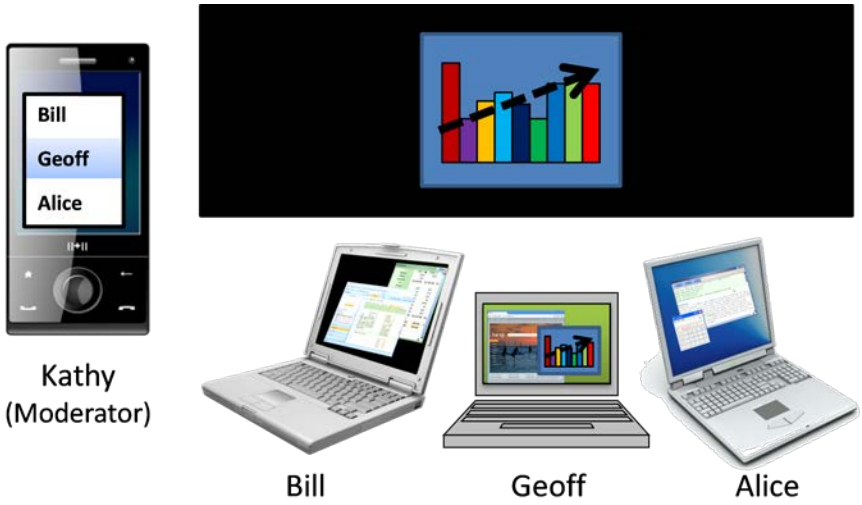


Figure 5:2—Kathy, the moderator, chooses Geoff to have floor control so the windows from Bill and Alice are hidden.

Kathy is hosting other panels at other institutions as part of an ongoing project. Rather than learning each institution's moderating software, in this *visitor situation* she uses her personal moderating software at all the institutions.

These four situations illustrate six different control techniques: presenter, free-form, moderated control, personal space, tiled, and floor. A single display space may be called upon to support all of these different techniques and any others which users deem necessary. The display space should support techniques that are not known to the display space but are familiar to its users.

5.1.2 Solution Requirements

There are two paradigms for how control techniques are provided to users: via the display server or via a personal device. For instance, in the panel situation Kathy used the floor control technique installed on her phone. But the display server could house the control technique instead. Kathy would use a computer embedded in the room's podium to select which of the presenters has the floor. While embedding the software in her phone allows Kathy to carry a familiar application with her wherever she goes, embedding the software in the display server ensures that Kathy does not need to carry anything.

If the display server supplies the control techniques a user can only employ server-installed control techniques. If the user is previously unaware of an appropriate control technique, then the user must find and learn a new technique on the spot. If the user is already familiar with a technique that she wishes to apply, then she can encounter frustrations at foreign rooms. Consider Kathy transitioning from the panel situation to the visitor situation. She is accustomed to a specific piece of software at her home institution and must now moderate a discussion at a foreign institution.

In a foreign room Kathy must first find the floor control technique. If the technique is available she must recognize and choose it from the list of (potentially many) items. But if the floor control technique is unavailable, then Kathy has three options: use an alternate technique, install the technique, or forgo the technique. Kathy may choose a technique that she hopes is similar enough and possibly learn that technique on the fly, which is likely to be frustrating. She may install the technique she wants, but because most institutions keep careful control over what may be installed, installing new software will lead to interacting with the support (Information Technology or IT) staff. Interacting with the IT staff requires her to either install the software on the fly or ahead of time, which consumes time and energy. Finally, Kathy may forgo using the floor control technique because that choice is easier than the prior paths, but requires her to spend more time giving verbal commands to the guests.

Because there are a potentially infinite number of control techniques that could be developed, it is unreasonable to expect the display server to have every possible version pre-installed. In addition, installing the software on the display server also increases the efforts of the maintenance staff to keep the display server up-to-date and introduces users to potential versioning conflicts. For instance, a user may inadvertently choose an older version of a familiar control technique, and then become frustrated with the bugs present or lack of features. Or, conversely, a newer version of the technique may have

unfamiliar features or options that get in the way when the user is trying to accomplish some task. Consequently, the display space control techniques should not be built in to the display server.

Users should be able to bring their own control techniques and apply them to any room whether or not that technique was previously applied to that room. If Kathy brings her own familiar floor control technique then she can apply it to any foreign room. This requires carrying a portable computer, but if that computer is her phone, then this barrier is low. Now she does not need to learn a new technique, interact with the IT staff, nor deal with versioning conflicts.

In summary, a solution to the illustrated collaborative situations must meet at least these points:

- *Automated display control*—support automated control techniques (display space management techniques)
- *Plugin-free display control*—support new control techniques without installing software on the display server
- *Familiar display control*—users do not need to learn new software to utilize a known control technique

5.1.3 Proposed Solution

This paper introduces the *window broker* protocol which provides automated, plugin-free, familiar, display control. Display servers do not have any control techniques installed but instead allow portable devices to supply a technique via network procedure calls to the user's personal device. Now a limitless variety of display space control techniques may be applied to any given display server.

To implement the window broker protocol a display server designates one machine as a *broker*. For instance, in Figure 5:2 Kathy's machine is set as the broker. When Bill attempts to move the shown window, that attempt is forwarded to Kathy's software which denies that movement because Bill does not have floor control. However, if Geoff moves the window, the attempt is also forwarded to Kathy's software which approves it because Geoff has floor control. The display server implements the results of each forwarded call.

With the window broker protocol, users may carry their own display management software anywhere they travel. Displays may now transparently implement new control techniques without installing new software because the decisions underlying the techniques are made by the broker on a client machine.

5.2 Prior Work

There are several collaborative screen management technologies that have been developed for various installations, each with varying control techniques.

Some systems implement a free-form control technique and use the display space's computer mouse as the arbiter. The most basic systems that implement this are X-Windows (X11) [57], Remote Desktop Protocol (RDP) [68], or Virtual Network Computing (VNC) [56]. These machines place control over the placement of windows in the display server, and users must have physical control of the display server's keyboard and mouse to rearrange the windows. Other systems that take a similar approach include IMPROMPTU [2], WinCuts[65], and Lacome [28] which are designed for groups of individuals who may periodically want to share and discuss information via a shared display space. In such collaborative situations, a user is unlikely to go up to the shared display and grab the input hardware just to exert control; he would rather exert control from his own device using software he is familiar with.

Other systems include control techniques that facilitate specific cases. WeSpace [6] provides an API that can be used to implement control techniques which are installed on the display server. One such control technique is implemented by LivOlay [6]. With WeSpace, someone could implement the tiled technique where no two windows are allowed to overlap. Unfortunately, the control technique must be installed on the display server, so people can only use a technique if it is installed on the server. This rigidity prevents users from portably using software such as LivOlay or the tiled technique wherever they need it.

Collaborative single-display software will often have case-specific control software. For instance, the Dynamo [5] interactive display environment implements a sophisticated version of the personal space technique. The display space has several computer mice installed and a user may use one mouse to carve out a personal workspace from an available portion of the display space. However, interaction at the display space is limited by the number of computer mice installed and is the only control mechanism available. Limiting the number of users limits the collaboration possible. Although Dynamo provides a nice technique for managing multiple users, the implementation is rigid and cannot support the other control techniques discussed.

iRoom [30] with PointRight [8] provides an interactive multi-user environment, but does not provide a display control system. PointRight is built on top of iRoom and controls what hardware device may provide input to an individual display. Input from any device in an iRoom setup may be directed to any display, but only a single set of input (keyboard and mouse) may be directed to a display at any one time. Fundamentally, this approach controls a single display at a time rather than the entire display space. For example, if one user is interacting with a window on one display, then all other windows on that display are inaccessible to other users. Or consider a user who is giving a presentation across multiple displays. That user may control one display at a time, but cannot control all of the displays.

A new flexible display space control mechanism is needed which allows for a wide variety of interactive collaboration techniques. This new control mechanism must support automated, plugin-free, familiar, display space control so that institutions have lower maintenance costs and users have familiar controls in any interactive room.

5.3 Window Broker

The window broker protocol is a display space management protocol which provides automated, plugin-free, familiar display space control. The Window Broker protocol has a simple authorization

mechanism which gives a single user's machine the power to control if and where windows are created and positioned on the annexed display space.

The window broker protocol is implemented as part of a screen sharing protocol called SPICE (SPaces for Interactive Computing in Education), but the techniques could be incorporated into other UI distribution protocols such as X11, RDP, or VNC. The SPICE and window broker protocols are built primarily in Java [22] (version 1.6), and to ensure broader compatibility a C# [16] version of the core protocols also exists.

The window broker protocol design stems from an attempt to control changes which affect the overall display space. For instance, an instructor may need to blank the screen to gain access to a whiteboard or show the contents of the screen again. Or, he may need to change the volume of a rendered video or mute the screen altogether. Other changes that need to be controlled involve the arrangement of windows on the screen: create, move, show, hide, destroy, or change z-order. The protocol supports the following controllable changes:

- Blank/Show screen (whiteboard access)
- Mute/Allow screen sound
- Change shared-audio volume
- Create a window
- Move a window
- *Shelve*/*Unshelve* (hide/show) a window
- Destroy a window
- Bring window to front
- Send window to back

The problem is how to authorize changes to these nine types of display space changes and how to dynamically change the authorization mechanism. Automated authorization allows a variety of display server control techniques. Dynamically changing those control techniques allows users to have greater familiarity.

The window broker protocol designates one of the connected client machines as the *broker*, which is also called the *broker machine*. Clients ask to be the broker and are granted on a first-come first-serve basis. If there is no broker and no client asks to be broker, then the display server implements the free-form control technique.

When there is a broker, the display server forwards requests affecting any of the nine controllable changes (called *forwarded requests)* to the broker machine. The software on the broker may allow, alter, or deny any request. The results of the *broker software*'s decisions are sent back to the display server for implementation. This process flow is illustrated in Figure 5:3.

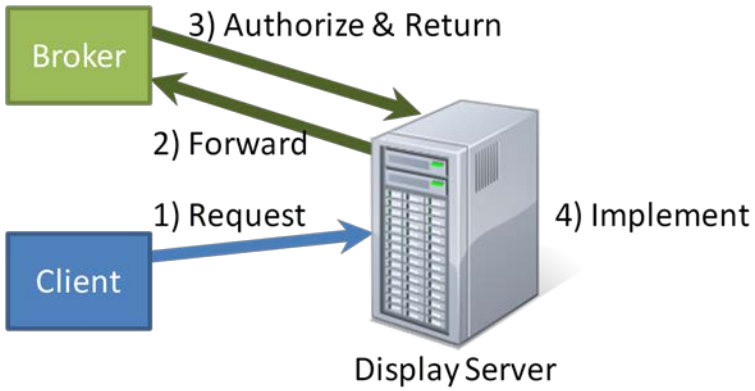


Figure 5:3—The display server forwards requests to the broker for authorization and then implements the results.

Requests that are not forwarded to the broker machine are those that change the content within a window: images, audio, video, and other graphics primitives. Those requests must come from the window's creator and are always honored by the display server.

The broker machine may have client software which attempts to make changes to the display space. For simplicity of software development and the toolkit the client software is separate from the broker software. The client software requests are routed by the window broker API directly to the broker software (bypassing the display server). The results of calling the broker software directly are sent to the display server and implemented.

The window broker protocol provides automated, plugin-free, and familiar display space control. The broker software provides automation because it can automatically enforce a control technique by modifying requests. For example, a broker algorithm could enforce boundaries on a particular user by changing any requests that extend outside that user's boundaries into requests that stay inside the boundaries. Because the broker software runs on a personal device, a control technique can be enforced without requiring a new plug-in to be installed on the display server. Including the notion of forwarded requests also provides familiarity by allowing users to carry and use their own broker software.

5.3.1 The User Experience

The SPICE framework is designed for environments where display servers are unlikely to provide hardware input. The display server may not provide input because it is physically distant from the users (e.g. a large conference hall) or otherwise inconvenient for supplying input. Therefore, the user's personal device provides the input necessary to rearrange windows.

Instead of using PointRight's [8] approach of treating the display space as an extension of the user's desktop, SPICE treats the display space as a separate screen space. On the user's personal device is a UI which shows a representation of the display space and its windows—typically a world-in-miniature display. The widget used to represent the display space and the windows is called a *window arranger*. Figure 5:4 shows a UI that users may encounter when annexing a display server. In the middle of the figure in part (a) is the window arranger which shows all the different windows that are shared on the display server. Other widgets in this UI are for sharing individual windows to the display server or applying an individual control technique. Part (b) shows a screen shot of the actual display space represented in part (a). In part (a) of this figure, the user has selected a window in the middle which he may move, resize, or close. In addition, there is one hidden window, which is listed at the bottom of the window. Note that the world-in-miniature view shown in Figure 5:4 (a) is only a potential UI and is not

required by the window broker protocol or windowing toolkit. Client software could implement any of a variety of user interfaces that accomplish this task. A given user will always see the UI that they are most familiar with.

(a)

(b)

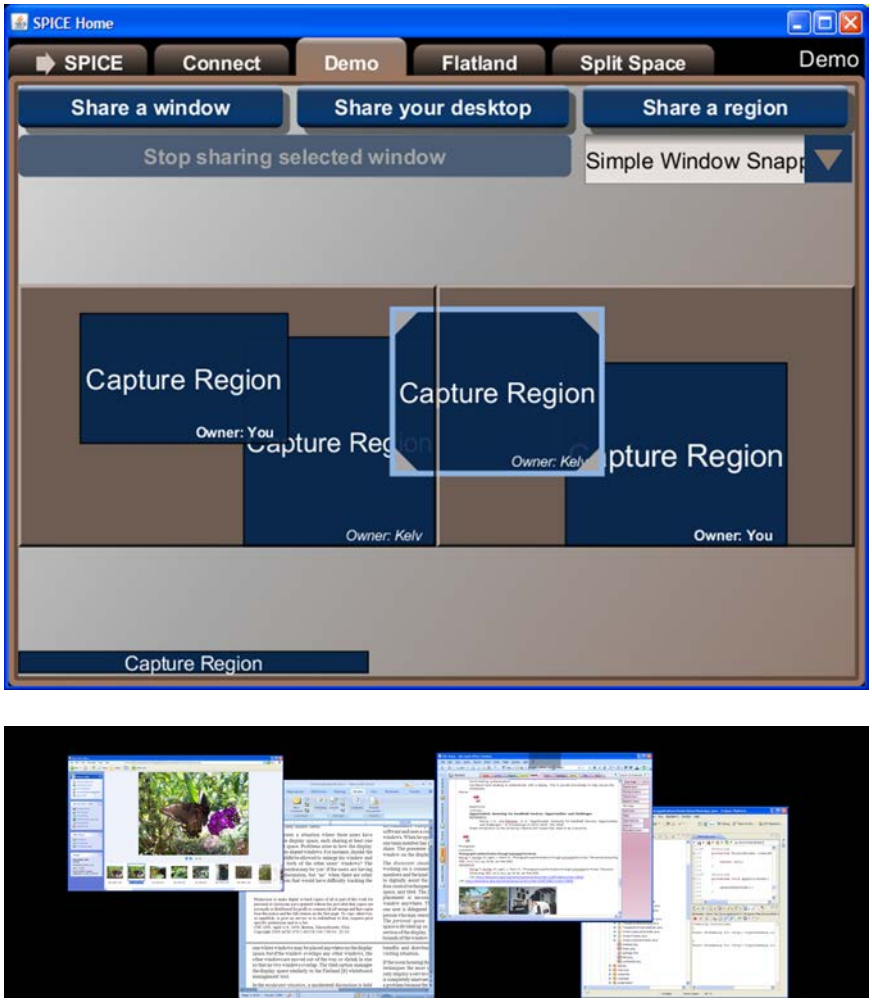


Figure 5:4—Window Arranger. Part (a) shows the icon view for managing the screen space. It shows a representation of the windows shared on a dual-screen display server. Part (b) shows a screen shot from the display server.

Moving or resizing a window via the window arranger is not performed live. Instead, the window arranger shows a preview of where the user is attempting to move a window. When he releases his mouse button, the window arranger attempts to move the window to that destination location. To move the window the software creates a move request and sends it to the display server, which forwards it to the broker machine, which authorizes the request and returns a response to the display

server, which then implements the response. If the move request is denied, then the display server sends the client a “rejected” notification. When the client software receives the rejection the software causes the moved icon of the window to snap back to its previous location which informs the user that the request was denied. If the broker software moves the window to a different location, then the new location is sent to the client and similarly reflected in the window arranger.

If the broker user is the user trying to move a window, the process is a little simplified. The broker user still sees the same feedback that a typical user sees. The broker user’s requests are treated like any other user’s requests. The requests bypass the display server and are forwarded directly to the broker software. The broker software’s results are reflected to the broker user, and any changes to the display space are sent directly to the display server and performed. Subsequently, all other client devices are notified by the display server.

5.3.2 Solving the Four Situations

The window broker architecture can be used to implement the four example situations (from section 5.1.1) via a separate broker algorithm for each situation.

Broker software has one or more sets of rules which govern how the software handles forwarded requests. A set of rules is called a *policy* and broker software may have more than one policy, but the software will enforce only one at a time. Policies are analogous to control techniques.

The following sub-sections discuss each of the various brokers and policies developed to solve the four situations delineated earlier.

5.3.2.1 Presenter Situation

In the presentation situation, the user employs software that has a *presenter broker* algorithm which handles all forwarded requests. The presenter broker uses one of two policies: exclusive and audience. Because the presenter should not be interrupted while giving a presentation, the exclusive

policy is the default policy and denies all forwarded requests. With the exclusive policy in use audience members cannot directly affect the presentation (e.g. show unsolicited material on the display space). Once the presenter finishes his presentation he may switch to the audience policy. The audience policy denies all requests except create requests. However, the audience policy *shelves* the created window (i.e. hides the window so it is not rendered on the display server). An example of a shelved window is the single window shown along the bottom of the UI in Figure 5:4 (a). The bottom of the UI is also called the *shelf*. The presenter software is informed of the created window so that the presenter may choose to drag that window from the shelf onto the display space, showing the window. In this way, audience members may still participate in the discussion, and cannot show unsolicited material.

5.3.2.2 Discussion Situation

In the discussion situation, a group of people work on a common project. This paper illustrated four control techniques for managing this space: free-for-all, moderated control, personal space, and tiled. The first two techniques are described in the next subsection. The other two are described separately in the following two subsections.

5.3.2.2.1 Free-For-All and Moderated Techniques

The *discussion broker* software implements the free-for-all and moderated control techniques. The discussion broker software can employ one of three separate user-chosen policies: moderated-control, only-owned, and free-for-all.

All three policies allow users to create windows. However, like the audience policy, the *moderated-control* policy shelves all created windows and only the group leader may choose to show any of the hidden windows. The moderated-control policy denies all other forwarded requests. The only-owned policy honors requests that affect windows that the requester owns and rejects requests that would affect other windows. The free-for-all policy allows anyone to affect anything shown on the

display space. Now the various group members can share and discuss individual windows on the display space using either the free-form, only-owned, or moderated-control technique.

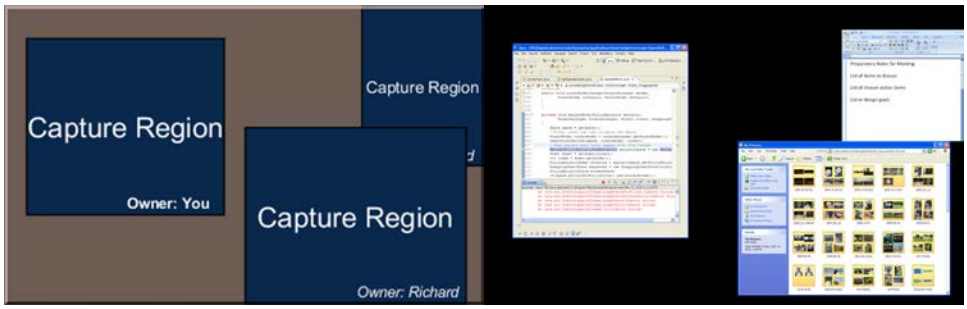
5.3.2.2.2 Personal Space Technique

The personal space technique is implemented as separate broker software called the *split space broker*. This broker has a single policy with three broad rules:

- divide the space equally among the participating users
- allow users to only move windows they own,
- keep a user's windows within that user's partition.

This split space policy acts similar to the Dynamo display space management paradigm except that the space is automatically allocated to each user, rather than "carved out" by each user.

Figure 5:5 illustrates what the display space partitioning looks like. Part (a) shows a clipping from the window arranger with window placements, while part (b) is the screen-shot of the display space represented in part (a). In this example, two people are connected to the display space, so the policy divides the space horizontally. Both users get equal-sized partitions of the space that are as close to square-shaped as possible (minimum perimeter length); one partition is on the left, the other on the right.



(a) (b)

Figure 5:5—Split space enforcement. 2 Users share the same display space. Part (a) is the screen management tool a user could have. Part (b) is the display space output. Each user's windows may only appear in that user's half of the display space.

The split space broker software must be able to handle adding and removing participants, particularly when display space is already divided among some participants. Suppose some user connects to the display space, not to share content, but to copy content and take notes. Connecting to the display space should not cause a rearrangement of all the windows on the display space, which can be frustrating to the users who already have space allocated and are utilizing that space. Consequently, the broker software allows the user—or *broker user*—to select which of the connected clients are considered participants.

To allow the broker user to choose which client machines are participants, the split space software provides the UI in Figure 5:6. When a client becomes broker the display server supplies that client with a list of all the connected clients and notifies the broker machine when a client connects or disconnects.

The split space broker software uses these connection notifications to update the participant selection UI shown in Figure 5:6. This UI lists all the non-participating clients in the left column and the broker user may move any of those clients to the "participants" category on the right.

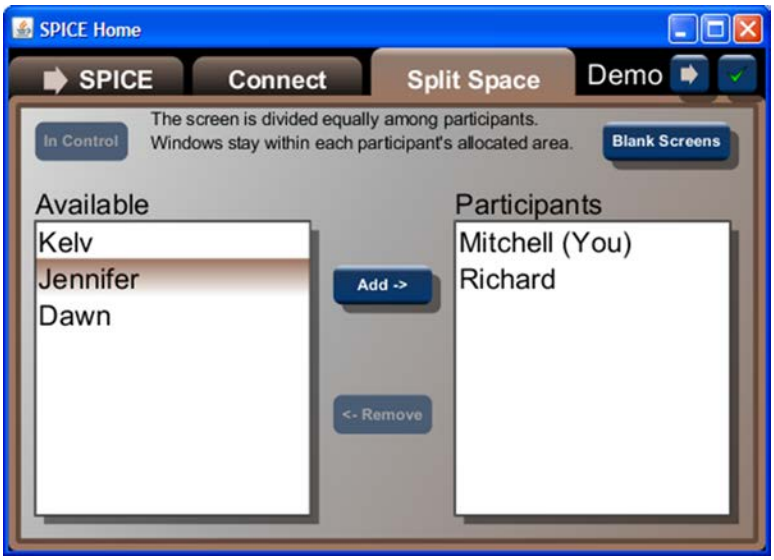


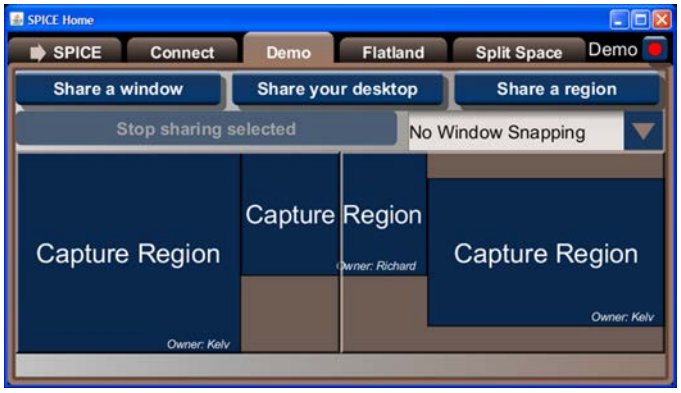
Figure 5:6—Participant selection UI. The broker user chooses participants from the list of connected machines.

With each selection or removal of a participant the split space broker software re-divides the display space. The software attempts to preserve the window arrangements within divisions and attempts to keep the divisions as close to their original location as possible.

5.3.2.2.3 Tiled Technique

The tiled technique is implemented as the only policy in the *flatland broker*. This broker allows anyone to share any information on the display space. The only constraint is that no two windows may overlap. Rather than enforcing this constraint by preventing window movement, any window may be moved to any location on the display space and all other windows move out of the way. Figure 5:7 shows a screen-shot from the client machine in part (a) and from the display server in part (b).

(a)



(b)

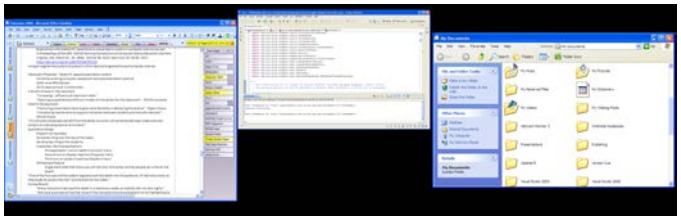


Figure 5:7—Flatland window management where no two windows may overlap. Part (a) shows the icon view for managing the screen space on a dual-screen display server. Part (b) shows a screen shot from the display server.

5.3.2.3 Moderated Situation

The moderated situation has combinations of the presentation situation and the personal space technique. No audience members can interrupt the various presentations, only the presenters may show

windows on the display space, and windows can only be shown and arranged by the client with floor control. The moderator launches her *moderator broker* software, becomes the broker for the display space, chooses who the participants are via a UI similar to the one in Figure 5:6, and then moderates the discussion. She chooses one person at a time to have floor control and thus absolute control of their own windows on the display space.

The moderator uses the UI in Figure 5:8 to select the participant to assign as the floor. The chosen participants are shown down the right side and the selected one is the floor. For example, Mitchell has the floor in Figure 5:8. The moderator can also time Mitchell's presentation using the clock on the left, and give him subtle feedback about his progress.

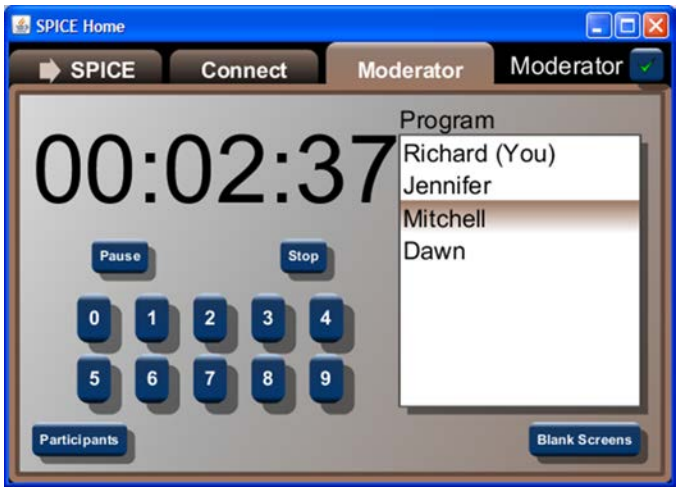


Figure 5:8—Possible moderator software UI. The person selected on the right is the floor. The countdown on the left is used to time the current speaker.

5.3.2.4 Visitor Situation

When the moderator is asked to travel to a new location, she can become broker for that display space. Because of the window broker API she does not need to install any software supplied by that display space's owner on to her personal device, nor must she install new software on the display space to support her moderator software. The only software that must be consistent between her client machine

and the display server is the window broker protocol. With such consistency, she can confidently manage the display space for a separate discussion.

5.3.2.5 Situation Solutions Summary

In each of these situations the broker software is always installed on a user's personal device. The display server is always just a display server and performs the tasks required to display information. Because the display server can forward requests to the broker device via remote procedure calls, the display sever does not need to change to support a new interactive technique.

On the other side of the issue is the fact that users always bring their own broker software with them. This broker software is always the software that they know and are familiar with. The software presents a UI on the user's personal device via a known screen and input devices, which keeps the software familiar regardless of the environment.

5.4 New Challenges

With the window broker protocol, some new challenges present themselves. These challenges include situations without brokers, wresting control from a broker, distrusted display servers, asynchronous requests, and client-side broker preview.

5.4.1 Situations without Brokers

Some collaborative situations may not need a window broker. In such cases the display server performs all of the requests from any connected device without forwarding the requests. This approach effectively results in the free-for-all control technique. Conflicts that occur between users are expected to be resolved socially.

5.4.2 Wresting Control from a Broker

Envision an English professor who has just finished instructing a class. The English professor neglects to relinquish broker control as he leaves. A math professor teaching the subsequent class enters the room a few minutes later, attempts to become broker, and discovers that someone else has control. The math professor must be able to quickly and easily gain control of the display space.

Or, imagine the case where the math professor attempts to take control of a display server and finds that someone else in the room is already broker. This student may be attempting to surreptitiously interfere with the math professor's presentation by allowing the professor to give his presentation but during the presentation the student rearranges windows or shows additional content.

There are two options for resolving these situations: ask for or take control. To ask for control means physically and personally asking the English professor to relinquish control. Unfortunately, the math professor may not know that the English professor is the prior professor in the room, let alone who the English professor is so that she may contact him. Worse yet, the English professor may have granted control to a student who also left; in such case tracking that student down may be difficult. Instead, the window broker protocol must allow the math professor to take (or wrest) control from the current broker.

The math professor is in the same room as the display space so wresting control should be straightforward. The default SPICE display server software provides a UI which can discard the current broker and possibly assign a specific machine as broker. Other installations may have a different setup but a similar goal. This UI may be a physical button on the display server which, when pressed, disconnects the current window broker. Or the UI may be a more sophisticated interface listing the currently connected users, from which the math professor may select himself. The key is that physical control of the display server can override any currently assigned broker.

5.4.3 Distrusted Display Servers

Within a controlled environment (e.g. a corporate or classroom environment) users can probably expect that a display server is completely trustworthy. The maintenance staff will keep the display servers free of malware that could damage client devices. However, not all display server environments would necessarily be as controlled or safe. For instance, a display server embedded in a restaurant table, in a mall kiosk, or a hotel room may not be as well maintained. These display servers may have malware that was transmitted to the display by a previous user intentionally or unintentionally. The window broker protocol should not be available as a virus vector so that future users may be protected from infection.

The window broker protocol does not act as a virus vector because it never transmits software. Instead of transmitting code the window broker protocol supports a limited set of commands which client and server machines interpret.

Flaws in the window broker protocol implementation (e.g. buffer overruns) may still exist on the display server or client machines, but the design of the protocol does not invite infection.

5.4.4 Asynchronous Requests

Some personal devices may generate situations with high network latency or temporary disconnections from the display server. The display server should not lock up during these situations and neither should the personal devices. Within the window broker protocol there are two main sources for high network latency or temporary disconnections: small devices and manual authorization.

Small devices, such as smartphones, tablets, and netbooks could become broker machines. These machines might not share windows on the display server, but they could still manage the screen space. For instance, in the moderator situation the user employs her smart phone to moderate the discussion

without showing any windows. These small devices often have slower processors than laptops, and may be limited to slower network speeds.

In some cases (e.g. iPhone [2] or Windows Phone 7 Series [10]) these devices can only execute one application at a time. Although iOS 4 claims support for multi-tasking, it still can only execute one application at a time; suspending any application that is not the foremost application. This means that if the first application is the window broker then it could be swapped out at any time; the user may temporarily be swapping to another application and could be back soon. As a result, the window broker software must be prepared to resume as broker at any time.

If the display server drops the smartphone as the broker because the window broker software is swapped out, then the smartphone may not be able to resume as broker. Consequently someone else may take control in the interim. If the display server stops accepting requests—or locks up—while the broker software is swapped out, then the display server will not be able to update content supplied by other attached users. For example, if the moderator swaps to another application momentarily to take a note, then the person with floor control may not be able to change to the next slide. The display server must allow the smart phone to continue being broker and must not lock up while waiting for that broker application to resume.

The window broker protocol employs a simplified network remote procedure call protocol. Remote procedure calls are typically treated as a standard (albeit slow) method call. To ensure that the call waits until a response is available, most RPC systems rely on the call stack and TCP connection to maintain state. Because the blocking thread is frequently the UI rendering thread, an application can appear to freeze or lock up.

If a slower device is the broker and if all requests to the device are synchronous (meaning the display server blocks on a thread and socket while waiting for a response), then the slower device

becomes the bottleneck in the experience for all other users of a display space. For instance, the display space could lock up while waiting for a response to a “move window” request. In addition, the client requesting to move that window would lock up while waiting for the display server which is waiting for the broker software to respond.

Locking up while waiting for a response from the broker machine is not acceptable. The protocol must support broken connections and occasionally long periods of time between requests and responses.

Related to the problems created by slower devices are manually authorized requests. In most cases broker algorithms are likely to be completely automated. In some cases, however, the broker algorithm may make decisions based on user input. For instance, the algorithm may show a prompt when a client requests to blank or show the screen. In such cases, the display server and other client software should not block while waiting for the broker machine to process the request.

To support the high latency and temporary disconnection required for slower devices and manually authorized requests the window broker protocol requests are always asynchronous (meaning the display server and client do not block on a thread and socket while waiting for a response). The source of each request assigns that request a unique ID and the broker tags the corresponding response with the same ID. Then the display server can match a response with the original request and implement the response accordingly. When the broker machine is temporarily disconnected, the display server queues any incoming forwarded requests and transmits the requests to the broker the next time it connects. In the meantime, any client software waiting for requests to be authorized are still interactive, but the display server will not reflect the requested changes until the broker responds.

To detect when a temporary disconnection is actually a permanent disconnection—regardless of whether that client is the window broker—the broker protocol uses a technique similar to HTTP session management. If a client does not reconnect within a small timeout—2 minutes by default—then the

client is assumed to have disconnected (e.g. the owner turned off his personal device and left the room without explicitly disconnecting). The display server then dumps all pending messages for that client. If the client is the window broker, then the display server processes each pending request denying each. Each request is denied so that the display space maintains its current state instead of trying to merge all the pending state-change requests; having the display server suddenly update to match all the pending requests can be a visually jarring experience for the other clients. After clearing all pending queues another client may become broker.

5.4.5 Client-side Broker Preview

The broker can greatly affect the actions that other clients may perform. It may accept, deny, or alter any of the forwarded requests. To help a non-broker user make good decisions the window arranger should provide feedback about possible actions. For instance, if the flatland broker is currently enforced, a non-broker user might like to see a preview of how the other windows would be positioned when he moves a window.

Consider Figure 5:9 which shows the client-side view of the display server's state. In this case the current broker is the split space broker and the user is enlarging the window by dragging the bottom-right corner. As can be seen in part (a), this move is illegal per the split space broker because the window encroaches on another user's partition. It would be better if the client were presented with the legal move shown in part (b).

(a) (b)

Figure 5:9—Reflecting split-space constraints on a client machine. Part (a) shows an illegal move that an uninformed UI could present. Part (b) shows the proper, legal action where the window is constrained to the left half of the display space.

The window arranger provides direct feedback about potential actions. One option would be to ask the window broker directly about the limits or consequences of a user's actions. Unfortunately, because the broker could be disconnected or the network could be a little slow, asking the broker directly for feedback about potential actions may not give timely information.

Another option would be for the broker to transmit its software via the display server to each of the other clients. The code transmitted would be some common framework such as JavaScript [20], Java, ActionScript [11], or MSIL [16]. However, this is a highly insecure option. The window broker could transmit malicious broker software to the display server, which it transmits to each client. Or, the display server could have malicious software installed which it transmits to clients in lieu of the actual broker software. The only way to mitigate this approach is to painstakingly sandbox the environment for hosting the transmitted software.

Instead of these prior two options, the window broker protocol makes a compromise; transmit static information about the window broker, and possibly use the same broker software if it is already installed on the client machine. To implement this, the window broker protocol has two additional features: policy hints and policy emulators. *Policy hints* supply a static, simplified version of the currently enforced policy and give client software a broad idea of potential actions. *Policy emulators*

give information about the actual policy being enforced so that clients that have that policy installed can instantiate the policy and use it to emulate what the window broker software would actually accomplish.

5.4.5.1 Policy Hints

Many brokers implement similar rules, although sometimes with slightly different implementations. For instance, the discussion broker's only-owned policy is similar to the split space and moderator broker policies. Likewise, the discussion broker's only-broker policy is identical to the presenter broker's audience policy (which creates windows but shelves them) and similar to the presenter broker's exclusive policy (which does not allow creation at all).

Most of the decisions these window brokers make are straightforward, simple decisions. These decisions are usually to allow or deny and are usually based on ownership or participation. In some cases extra processing is needed for the broker software to make a decision (like the split space or flatland brokers which keep windows within a partition or move other windows in response). The window broker protocol takes advantage of the similar simple decisions of many brokers while allowing more complex decisions by other brokers.

To inform client machines about possible actions without transmitting code the window broker protocol supports transmitting *policy hints*. These hints are key-value pairs of a hint type and its value.

There are 15 hints, 14 of which represent a forwarded request. The 15th is described in a moment. For each request that affects a window (move, destroy, shelve, to front, and to back) there are two hints: one for window owners and another for non-owners. The value for each hint is one of three values: allowed, denied, or brokered. "Allowed" means that the request will always be authorized by the broker software. "Denied" means that the request will always be denied by the broker. "Brokered" means that the result is too complicated to express as either allowed or denied; the client can request that action, but the broker could allow, deny, or alter that request. The "create window" policy hint supports

a fourth option called “Shelved”. “Shelved,” means that window creation is allowed but the window is hidden.

The 15th policy hint is a Boolean which expresses whether the broker software is participant based. If the software is participant based—such as the moderator and split space brokers—then participants use the 14 other hints and non-participants’ software assumes every action is denied.

Consider the policy hints for the tiled control technique which are shown in Figure 5:10. This control technique constrains the movements of the affected window (by keeping it within the display space’s bounds) and arranges any neighboring windows. Hence the “Brokered” tag on window creation, movement, and shelving.

Participants Only		*False*
Create Window		*Brokered*
Change Volume		*Denied*
Change Mute		*Denied*
Change Blank		*Denied*
	Owned	**Not Owned**
Move	*Brokered*	*Brokered*
Shelve/Unshelve	*Brokered*	*Brokered*
To Front	*Allowed*	*Allowed*
To Back	*Allowed*	*Allowed*
Destroy	*Allowed*	*Allowed*

Figure 5:10—Policy hints for the tiled control technique in the flatland broker. This is a participant based broker, and users are allowed to alter any windows but the broker may adjust a window’s final positions.

Also observe the policy hints for the only-owned policy (which is part of the discussion broker) shown in Figure 5:11. This policy is participant based and only allows users to affect their own windows. Creating windows is brokered because the participant with the floor can show those windows immediately, while all other created windows are shelved. Similarly, only the participant with floor control can change the shelved state of his windows.

Participants Only		*True*
Create Window		*Brokered*
Change Volume		*Brokered*
Change Mute		*Brokered*
Change Blank		*Brokered*
	Owned	**Not Owned**
Move	*Allowed*	*Denied*
Shelve/Unshelve	*Brokered*	*Denied*
To Front	*Allowed*	*Denied*
To Back	*Allowed*	*Denied*
Destroy	*Allowed*	*Denied*

Figure 5:11—Policy hints for the moderator technique in the moderator broker. This is a participant based broker, and users are allowed to alter their own windows but no others.

For a window arranger on a client machine to give proper feedback about what options are allowed, the window arranger must know whether its executing client machine is a participant. Client machines do not have a direct connection to the window broker because the display server is an intermediary. Consequently, the display server must inform each client machine about whether it is a participant. The display server can only know which clients are participants if the window broker exposes that particular information to the display server. So, a participant based window broker must inform the display server when a participant is added or removed. Then the display server can inform each of the clients of changes in the participants list.

5.4.5.1.1 Exploitability

The policy hints portion of the window broker protocol is difficult for a malicious display server or window broker to exploit. Client software interprets the policy hints without executing server- or broker-supplied software because each policy hint is a static value and contains no code. Although the display server may alter the broker-supplied values, altering those values provides no visible benefit to the display server toward exposing sensitive data or infecting a client machine.

5.4.5.2 Policy Emulators

The policy hints effectively express the simple decisions that brokers may make. However, brokers such as the tiled and split space brokers have sophisticated algorithms that are too complicated to describe using policy hints. These algorithms cannot be transmitted using the policy hints portion of the window broker protocol.

Rather than transmit the window broker's code, client machines could use software they already have installed. Envision a group of three people using the split space control technique. The first worker is the broker user and has the split space broker software installed, the second user works at the broker user's institution, and the third user is visiting. The second user has the discussion broker installed while the visitor does not. When the first user's personal device annexes the display space it requests to be broker and informs the display server of the policy hints which includes the name of the split space control policy.

The second user annexes the display server and her personal device is informed that the split space control technique is in use. Her personal device finds and executes the locally installed split space control code so that the window arranger can use it to emulate a preview that keeps her windows within her partition, similar to what is illustrated in Figure 5:9 (b).

When the visitor's device annexes the display server, it is also informed of the policy hints, including the name of the split space control policy. But the visitor's machine does not have the split space software installed. Akin to what is shown in Figure 5:9 (a), he can drag windows around but the preview does not constrain the windows to his partition. However, he only sees the constraint after the response finally arrives from the broker. Thankfully, because of the policy hints, his window arranger reflects that he cannot drag or resize any of the other people's windows.

To inform clients of the current broker software the window broker protocol transmits two extra strings with the policy hints. The first string uniquely identifies the class that may be used to instantiate the broker software, called a *policy source*. This policy source is usually a fully-qualified class name, such as those used in Java or .NET [16]. The runtime uses this class name to find and instantiate that policy source via reflection. The second string is the *policy version*, and is passed to a single method on an instance of the policy source. The method (called getPolicyEmulator) takes in the policy version and returns an instance of a *policy emulator*. The policy emulator can be used to emulate the execution of the current broker's policy.

The policy emulator should exactly emulate the broker software. However, the broker software frequently has more information in its model than just the state of the display server. For instance, the split space broker assigns a rectangular partition to each participant. Although the display server can inform the policy emulators of the chosen participants, the display server is generalized enough that it cannot inform the emulators of the partitions. The emulators could deduce the partitions from the current window arrangement, but if more than one client has no windows shown then the emulators cannot precisely deduce the partitions for all clients. The broker software must be able to expose additional state data via the display server. The additional state data allows policy emulators to accurately emulate the broker software.

To transmit extra state information to the clients the display server has a generic annotation mechanism. The broker software may attach key value pairs of strings as metadata to window handles or the broker handle. The broker handle is an object exposed through the window broker API which represents the actual broker machine and software. This handle provides clients access to the broker's policy hints and the name of the person who is the broker user.

When the broker software annotates a window or the broker handle that key value pair is transmitted to the display server and subsequently to each of the connected clients. To extract the annotation each client machine's policy emulator may then inspect the window handles or the broker handle.

For the split space broker software to inform each client of how the display space is divided, it serializes the array of partitions to a string and then annotates the broker handle with that string. The display server transmits this annotation to each client, where the software on the second user's personal device can decode the partitions. Now the split space emulator on the second user's personal device has the complete model that the broker user's software has, and can present an accurate preview of window movements.

5.4.5.2.1 Exploitability

Because no code is transmitted from the broker machine to each of the clients, the clients are protected from potential infection. In addition, the toolkits on the client machines do not need to be designed to sandbox the broker software.

The policy emulator approach is less secure than the policy hints but more secure than transmitting code. If there is a known exploit in a particular window broker a malicious display server could easily transmit the exploitable broker's policy source and policy version strings instead of the current window broker. Then the server could attempt to exploit the weak policy emulator by transmitting malicious serialized model data.

Developers of such network-enabled software should always create code that protects itself from such exploits. Client machines can protect themselves from exploitable window brokers. A well-maintained (i.e. regularly patched) client machine would have a list of known exploitable broker sources and would prevent those sources from ever being instantiated.

5.5 Summary

This paper introduced the window broker protocol. This protocol allows a separate machine to participate in the window arrangement on a display server. In particular, users may bring their own machine to a display server and enforce new window arrangement rules. This flexibility provides automated, plugin-free, portable display space control.

This paper also illustrates several different automated broker software implementations: presentation, discussion, split space, flatland, and moderator brokers. Each broker is implemented in Java and demonstrates the portable, plugin-free nature of the display space control afforded by the window broker protocol.

In addition, the window broker protocol is designed to support small devices that may be intermittently connected to the display server. This support is implemented by messages in the protocol which operate asynchronously.

To provide feedback for client machines which may or may not have the broker software installed, the window broker protocol also provides policy hints. These hints let a client machine know what broker software is being used (in case the client has that software available for emulation), and some of the basic decisions which the broker makes (in case the client does not have that software available).

5.6 REFERENCES

1. Apple Inc. iPhone. http://www.apple.com/iphone/, accessed June 2010.
2. Biehl, J. T., Baker, W. T., Bailey, B. P., Tan, D. S., Inkpen, K. M., and Czerwinski, M. 2008. Impromptu: a new interaction framework for supporting collaboration in multiple display environments and its field evaluation for co-located software development. In *Proceeding of the Twenty-Sixth Annual SIGCHI Conference on Human Factors in Computing Systems*(Florence, Italy, April 05 - 10, 2008). CHI '08. ACM, New York, NY, 939-948.

3. Flanagan, D. 2006 *JavaScript: the Definitive Guide.* O'Reilly Media, Inc.

4. Gosling, J., Joy, B., Steele, G., and Bracha, G. 2000 *Java Language Specification, Second Edition: the Java Series*. 2nd. Addison-Wesley Longman Publishing Co., Inc.

5. Izadi, S., Brignull, H., Rodden, T., Rogers, Y., and Underwood, M. "Dynamo: a public interactive surface supporting the cooperative sharing and exchange of media." *User Interface Software and Technology* (UIST '03), ACM 2006, 159-168.

6. Jiang, H., Wigdor, D., Forlines, C., Borkin, M., Kauffmann, J., and Shen, C. 2008. LivOlay: interactive ad-hoc registration and overlapping of applications for collaborative visual exploration. In *Proceeding of the Twenty-Sixth Annual SIGCHI Conference on Human Factors in Computing Systems* (Florence, Italy, April 05 - 10, 2008). CHI '08. ACM, New York, NY, 1357-1360.

7. Johanson, B., Fox, A., and Winograd, T. 2002. "The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms." *IEEE Pervasive Computing* 1, 2 (Apr. 2002), 67-74.

8. Johanson, B., Hutchins, G., Winograd, T., and Stone, M. 2002. PointRight: experience with flexible input redirection in interactive workspaces. In *Proceedings of the 15th Annual ACM Symposium on User interface Software and Technology* (Paris, France, October 27 - 30, 2002). UIST '02. ACM, New York, NY, 227-234.

9. Liu, Z. Lacome: a Cross-Platform Multi-User Collaboration System for a Shared Large Display, Computer Science, University of British Columbia, 2007. http://hdl.handle.net/2429/378

10. Microsoft Corporation, Windows Phone 7 Series, http://www.windowsphone7.com/, accessed June 2010.

11. Moock, C. 2007 *Essential Actionscript 3.0*. First. O'Reilly.

12. Mynatt, E. D., Igarashi, T., Edwards, W. K., and LaMarca, A. 1999. Flatland: new dimensions in office whiteboards. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: the CHI Is the Limit* (Pittsburgh, Pennsylvania, United States, May 15 - 20, 1999). CHI '99. ACM, New York, NY, 346-353.

13. Richardson, T., Stafford-Fraser, Q., Wood, K. R., Hopper, A., "Virtual Network Computing", *IEEE Internet Computing*, Vol. 2, No. 1, 1998.

14. Scheifler, R. W. and Gettys, J. “The X Window System,” *ACM Transactions on Graphics*, vol 5(2), (April 1986), pp 79-109.

15. Tan, D. S., Meyers, B., and Czerwinski, M. 2004. “WinCuts: manipulating arbitrary window regions for more effective use of screen space.” In *CHI '04 Extended Abstracts on Human Factors in Computing Systems* (Vienna, Austria, April 24 - 29, 2004). CHI '04. ACM, New York, NY, 1525-1528.

16. Thai, T. L. and Lam, H. 2002 .*NET Framework Essentials (2nd Edition)*. O' Reilly & Associates, Inc.

17. Tritsch, B. 2003 Microsoft Windows Server 2003 Terminal Services, Microsoft Press.

Chapter 6 Summary and Conclusions

The goal of this research is to allow a cellphone or smaller-sized device to become the user's dominant computing device. Part of this goal is for the device to remain physically small, remain safe, and to supply a consistent, larger interface for the user to interact with at any display server he encounters. This chapter summarizes this work's contributions toward these goals and identifies future work.

6.1 Contributions

Large UI from a Small Device: As was shown in Chapter 2, a cellphone-sized device can provide a much larger interface by seamlessly distributing an application's UI to other machines. Several common user applications were demonstrated, such as a word processor, note taker, spreadsheet, and presentation applications. In each case the application stays on the personal device processing the user's data, while the UI from that application can be transmitted to an annexed display server for rendering. Because of this arrangement of software and hardware, the cellphone-sized device can now become a user's primary computing device.

Consistent UI from Display Server to Display Server: By distributing the UI as a well-defined tree-structure of primitives, the UI can be consistent from display server to display server. In partial demonstration of this assertion, a display server was implemented in both Java and C#—which have different rendering engines—and consistent output was achieved from both. Because the number of graphical primitives is small (under two dozen) and the primitives for positioning are precise, such consistent output should be achievable by any modern rendering toolkit.

Rich Input from Annexed Display Servers: The XICE toolkit includes a simple input transmission protocol which processes a wide range of input types. The protocol handles the standard keyboard and mouse and is also designed to handle input from a stylus, fingers, and hands.

Safe Interaction at Untrusted Display Servers: When a user annexes a display server he has never employed before, the windowing toolkit prompts the user for information about the user's trust level for the display server; this prompt defaults to distrusting the display server. The user-assigned trust level is supplied to an application for each window the application transmits to the display server. The application then has the opportunity to augment itself to protect any data deemed sensitive. How sensitive data is identified and how such protection is performed is not specified by this framework.

In addition, because the windowing toolkit only sends rendering primitives to the display server—instead of the user's data or applications—the display server has less room to exploit the user's software and data. By assigning the aforementioned trust level, applications can be even more vigilant against input that affects sensitive data (e.g. by causing that data to show) by double-checking such input on the personal device.

Legacy Support for Existing Applications: The SPICE framework was developed as a screen sharing protocol for legacy applications which is more advanced than VNC, but not as tightly coupled to the rendering toolkit as RDP or X11. SPICE allows windows from existing applications from a user's machine to be replicated pixel-for-pixel on an annexed display server and to be positioned on the display server independently of the window's original arrangement. By allowing windows to be shared on annexed displays in this way, a smaller device (e.g. a laptop) can appear to be a much larger device (e.g. a multi-display driving computer). In addition, SPICE demonstrated a cellphone being used to give a media-rich, multi-screen presentation; the cellphone supplied all the media in the presentation as well as the controls for navigating through the presentation.

Portable, Login-free Display Space Control: Because a display space can be utilized by multiple people simultaneously and anonymously, users may need to employ some control over that display space. The Window Broker protocol was added to SPICE to allow a user to carry his preferred

control system to whatever display space he may annex. Rather than trying to interact with a potentially unfamiliar application which would be installed at the display server, the user always has access to a familiar application because he brought it with him.

6.2 Future Work

The interaction space made available by XICE is successful in its purpose to "allow a cellphone or smaller-sized device to become the user's dominant computing device, to remain physically small, and, in a relatively safe manner, to supply a consistent, larger interface for him to interact with at any display server he encounters." As was shown in the previous section, the idea of allowing a user to carry his data, settings, and applications allows him to have familiar, powerful, interactive experiences wherever he may go.

However, the research within this dissertation is only a beginning for much more potential research. This section will list out several areas of research that could be pursued because of the interaction model championed by XICE and SPICE.

6.2.1 Do Users Like "the Cloud?"

One key piece of information which needs to be elucidated is whether or not an end-user actually likes the "live in the cloud" interaction model where he keeps all of his data on web servers and just logs in via any computer with a web browser. This is a convenient approach to user interaction, but as shown in Chapter 2 section 2.3.3.3, it has some serious flaws that should be carefully weighed.

In addition to those flaws, a user may feel like living in the cloud is like having big brother constantly over his shoulder as websites and applications constantly monitor his every move. A user may also worry that a trusted site may go bankrupt at some point, or unexpectedly lose his data. A user may feel more comfortable carrying (and thus taking blame for losing) his own data than trusting a third party to preserve that data. If a user can have the same convenience of living in the cloud—namely

being able to access his data, applications, and settings anywhere—without the associated drawbacks, he may prefer the interaction model afforded by XICE for data storage and access. But, as stated in Chapter 2 section 2.3.3.3.1, XICE is not meant to replace the web, just to enhance it. The user would still have access to websites, just in a more portable format.

However this user preference data is collected, care must be taken to properly collect user "live in the cloud" opinions without unduly biasing responses. Many users may not be aware of the risks, so some on-the-spot education may be necessary in process of the survey. The survey may even collect data both before and after informing participants so as to help make the research community aware of whatever bias may have been introduced by the survey.

6.2.2 Unifying Input Systems

The user interface presented to users must be flexible to a wide variety of input mechanisms. Developers targeting multiple input types are often required to handle each potential input system independently. Thus, when designing a UI, the developer must handle a mouse, stylus, finger, hand, laser pointer, and pen individually.

In addition, the Microsoft Kinect [2] is becoming an attractive interaction tool which has completely different affordances from the mouse and keyboard. What works for a gaming controller may not work for a mouse. And what works for a mouse may not work for the indirect input from Kinect; free-space full-body detection.

For developers, the cross product of all these different input systems with the potential form factors for the UI results in a difficult problem which exponentially multiplies the complexity a developer must cope with. Developers need a simpler user input interaction model which has a few choices that cover a majority, if not all, of the potential interaction combinations.

6.2.3 Cross-Modal Widget and UI Design

Related to unifying input systems is merging the way that input/output combinations are developed. For example:

- Buttons should be larger on displays that accept finger input.
- Screens that accept only mouse input may want to show scroll bars which the user can interact with.
- It is easier to experiment with a right-click when you have a mouse, than when you only have fingers.
- Getting tool tips is currently impossible when there is no mouse.

If the UI for the user changes too much, the user can be left frustrated or confused, resulting in a poor experience. But there may be interactions which are more convenient with fingers than with a mouse or stylus as well as vice versa. If the UI does not change to afford such interaction styles then the user may be equally frustrated. Even worse, the user may be in a situation that accepts both the mouse and finger input, and the UI must flex to efficiently accept both.

In the end, developers need tools that help them plan for all the different ways information can be displayed and supplied, while keeping the complexity low.

A related topic is globalization. Different cultures have different ways of presenting information (e.g. right-to-left, or top-to-bottom). Currently, developers attempt to make existing UIs change to fit such presentation styles, but the approaches they have for changing UIs may be insufficient. It might be better to design applications to follow Model-View-Controller design and then build a separate view for each culture. Or, to build a small set of views which cover the wide range of cultures the developers choose to target. Such research, although it doesn't directly stem from the nomadic interaction model presented in this dissertation, may help fuel the research into cross-modal widget and UI design because it helps highlight the parts of the widget/UI design process which need flexibility and simplification as well as providing options for managing that flexibility and simplification.

6.2.4 Effective Privacy-Aware Applications

Although XICE is shown to be a privacy-aware windowing toolkit, most prior privacy-aware application research has been done with self-contained applications; each application must independently collect the user's trust level for the display. XICE, on the other hand, collects the user-specified trust level and provides the trust level to any application that needs it. Supplying the trust level at the toolkit level frees researchers from building their own application-specific privacy-collection software. Consequently, the researchers can focus on developing novel privacy-aware applications.

In Chapter 3 many different privacy-aware application examples were explored. However, none of the example applications were measured as to their effectiveness at helping users control sensitive data within those applications. More research is necessary to help software developers understand what is successful in terms of privacy-aware application UI design.

6.2.5 Non-Display-Server Device Annexation

Display servers are not the only devices users may want to annex. Users may want to annex devices such as printers, stereos, or thermostats. In these cases, the user would not want to install a device driver (this is a virus-prone approach) but still be able to employ the device.

Consider a user who may want to interact with his stereo somehow. Maybe he would like to play music over it, or control some of its settings, or configure it to operate with his television setup, or maybe he wants to configure the stereo's advanced settings via his television. That stereo could provide a UI to the handheld that could be shared to the TV, which gives the stereo much more room to display its features. Companies that produce stereos are constantly trying to improve their stereos through differentiating features. Consequently, a protocol which limits what features are available is a disincentive to product designers. A protocol is necessary which helps protect the user's handheld device while allowing manufacturers to innovate and make their features available.

6.2.6 Secure Anonymous Pairing

A key piece of this framework is anonymously establishing connections with a display server. A standard TCP connection may not be secure enough for users; other devices may be able to sniff the traffic and somehow exploit the user's interaction with the display server. But modern web-based security protocols require a trusted third party to issue digital certificates. Unfortunately, an institution may not be willing to pay the annual expenditures necessary to supply certificates to each display server it may own.

A crucial improvement to this interaction model would be to allow users to establish secure connections to display servers without requiring a well-known/trusted third party authentication system like a Certificate Authority, akin to the research by McCune, Perrig, and Reiter [1]. If a user does not need a CA, but can trust the connection is secure, then he can more confidently annex such display servers even when he does not have access to the Internet. In addition, with such a decentralized security system, display servers are more likely to be widely adopted.

6.2.7 Collaboration

When multiple users collaborate, they may want to share information between their personal devices. The users could just employ the connection they have with the display server to share their data, but the display server may be untrusted; any data shared to another user through that display server could be captured or altered.

Users need to be able to establish secure connections with each other independently of the display server so that they can share data while still annexing untrusted machines.

6.2.8 Multi-screen Presentation Authoring and Delivery

Although the SPICE research in Chapter 4 demonstrated creating and authoring multi-screen presentations, such presentations were not the focus of that work. More research should be performed to

discover how to effectively develop multi-screen presentations. Such information might include how to help the designer see how his presentation will appear and how transitions will happen (because some transitions change all screens, while others change one or two screens), as well as how to effectively arrange information for viewers (should users constantly cycle content on each of the three screens, or is there a more consistent way of presenting information that can help audiences absorb that information).

Other research might include how to effectively get a multi-screen or single-screen presentation to flexibly and (potentially partially) automatically upgrade or downgrade its presentation. The user may have more or fewer screens than the presentation was designed for, and the presentation should be able to take advantage of or gracefully degrade to the new arrangement.

6.2.9 Hardware Experimentation

The research into input styles in Chapter 2 uncovered that it is much easier to control an annexed display server if the user's personal device has separate input hardware. This research should be pursued further and devices such as the MousePuter explored more. For instance, a smartphone may include a touchpad on its back side; any time the user positions his phone face-down he can interact with a shared display whereas when the cellphone is face up he can interact with it normally.

6.3 REFERENCES

1. McCune, J.M.; Perrig, A.; Reiter, M.K.; "Seeing-is-believing: using camera phones for human-verifiable authentication," *Security and Privacy, 2005 IEEE Symposium on*, pp. 110-124, 8-11 May 2005 doi: 10.1109/SP.2005.19
2. Microsoft Corporation, Kinect, http://www.xbox.com/en-US/kinect, accessed September 2011.