DIAMOND: A Framework for Dividing Interfaces Across Multiple Opportunistically aNnexed Devices

Heather M. Hutchings, Jeffrey S. Pierce
College of Computing
801 Atlantic Drive
Georgia Institute of Technology
Atlanta, GA, 30332-0280,USA
Tel: 1-404-385-4239
{Heather.Hutchings, jpierce}@cc.gatech.edu

ABSTRACT
Despite the increasing prevalence of physically proximate computing devices, current interfaces remain largely limited to single computing devices because of the prevailing assumption that interfaces can only draw on input and output (I/O) resources attached to the same device. That assumption has led previous research to emphasize transferring interaction to the computer with the best available I/O resources, but that approach introduces security and privacy risks. We propose to instead allow users to divide interfaces across multiple devices so that they can allocate functionality and information appropriate across trusted and untrusted devices. In this paper we identify requirements for a framework to effectively support the creation of divisible interfaces; we describe DIAMOND, a framework meeting those requirements; and we present example applications that we built with it.

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

General terms: Design, Human Factors, Security

Keywords: Divisible interfaces, opportunistic annexing

INTRODUCTION
User interfaces require input and output (I/O) resources. We can view the design of a user interface, therefore, as a process of determining how to make the best use of available I/O resources to allow users to access and modify information. While that process may entail making more effective use of existing resources, it can also entail incorporating additional resources.

Indeed, we can view the evolutionary improvement of user interfaces as relying in part on the gradual increase in the set of available resources. The evolution in input resources, driven by the desire to increase speed and expressiveness, started with improvements to a single device (switches to cards to keyboards) and continued with the additional of multiple devices (e.g. keyboard plus mouse, stylus, and camera). The evolution of output resources has followed a similar path, starting with improvements to standalone devices (e.g. larger monitors) and leading eventually to multiple devices.

This evolution of interfaces has been bounded, however, by the prevailing assumption that a computing device can only draw on I/O resources connected to a single device. That assumption unnecessarily limits the I/O resources that interfaces can draw on. It has also lead prior work (e.g. [6,22]) to emphasize transferring interaction, and all of the information necessary to support it, to the computing device with the best available I/O resources. That approach presents both security and privacy risks.

Consider the case of a user, carrying only a cell phone, who wants to read and respond to email. Although the user could conceivably read and respond to email on his cell phone, a more likely solution is for the user to find a nearby desktop computer (e.g. in an Internet café) and access his email using the World Wide Web (WWW). In order to access his email, the user will need to type his password into that computer, thus introducing a security risk: the computer could capture that password for later use.

After authenticating himself, the user gains access to his email inbox. This access presents a relatively minor privacy risk: the information in the inbox interface could conceivably contain sensitive information that the computer could capture. A more serious risk is that the computer is now acting as the user's intermediary his data, and the computer's actions may not match the users intentions. While the user might choose to open only those messages that do not contain sensitive information, the computer could surreptitiously open and store *all* of the messages without revealing its actions to the user. Depending on the interface, treating the computer as a trusted intermediary could also present a security risk. The computer could, for example, delete all of the user's email or send email purporting to be that user.

While researchers have explored authentication mechanisms to eliminate the need to type in passwords (e.g. [13,14]), any approach that relies on transferring interaction is vulnerable to a malicious intermediary. A better approach is to take advantage of the increasing proliferation of computing devices by breaking the assumption that I/O resources must be physically connected to the same computing device and allowing users to *divide* interfaces across multiple devices. Users can then allocate interface functionality and information across their own devices and devices in the environment based on their level of trust and the perceived risk.

Developing divisible interfaces with current software frameworks is possible but not easy. In order to encourage the development of applications with divisible interfaces and facilitate exploration of the design space, we need a framework that simplifies the development and deployment of divisible interfaces. In this paper we present DIAMOND, a framework for Dividing Interfaces Across Multiple Opportunistically aNnexed Devices We begin in the next section by describing previous work on multi-machine interfaces (MMIs). We follow that section with a discussion of the requirements for a framework that supports creating divisible interfaces. We then describe the implementation of DIAMOND, present several example applications built with it, and discuss how well it meets the requirements we identified. We close with a discussion of future work.

PREVIOUS WORK

The idea of interfaces spanning multiple devices is not new. Previous research on single-user multi-machine interfaces (MMIs) [10] typically falls into one of two categories: development of interfaces that replicate all or part of an interface across devices, and development of interfaces that allocate different interface components to different devices.

While applications that replicate interfaces across multiple devices are more common for computer-supported cooperative work (e.g. [7]), a single user can also employ them. VNC [24] and Microsoft's Remote Desktop both replicate the whole desktop interface across devices to facilitate remote access, while WinCuts [25] allows users to replicate parts of an interface across devices. While replication is relatively easy to develop and deploy, it does not provide the same level of privacy protection as dividing interfaces.

Researchers have also created point designs for applications that allocate different interface components to different devices. As part of the PEBBLES project, Myers et al explored interfaces that span PDAs and desktop computers in order to, among other things, allow users to control a desktop PowerPoint presentation with their PDA [10] or use the PDA as an input device for their non-dominant hand when interacting with the desktop [9]. As another example, Rekimoto et al developed an MMI where a PDA augments interaction with a digital whiteboard [23], While these (and other) point designs explore the potential of different points in the design space for divisible interfaces, the designs themselves rely on a fixed allocation of interface functionality across devices. Users cannot easily move interface components between devices if they prefer a different allocation of functionality.

In addition to developing applications with interfaces that span multiple devices, researchers have also started to develop infrastructures that allow users to opportunistically annex [17] computing devices in order to draw on their I/O resources. iCrafter, together with iROS, allows users to employ their mobile devices to access services within an interactive workspace [18]. iCrafter supports annexing by allowing users to push information to display services that will then display that information about the room; multi-browsing is an example of this use of iCrafter [4]. XWeb is another infrastructure that allows users to access services from their mobile devices [11]. XWeb allows users to annex devices they encounter and push interfaces to them by providing a Capture primitive that connects devices to the user's interaction session [12]. SpeakEasy also facilitates annexing by facilitating the interconnection of arbitrary I/O and computing devices [20]. While each of these infrastructures makes it easier for users to annex available I/O resources, they provide little or no support for dividing an application's interface once the user has annexed devices. What we lack is a framework to overlay those infrastructures that makes it easier for developers to build applications that can dynamically divide their interfaces across computing devices based on user preferences and the available I/O resources. We identify the requirements for such a framework in the next section.

REQUIREMENTS

Existing frameworks could, in theory, support the creation of divisible interfaces. To be successful, however, a framework must meet the requirements of users who will work with applications built with the framework, the requirements of developers who will build those applications, and requirements imposed by the need to widely deploy the framework to provide a sufficient density of devices that support divisible interfaces.

User Requirements

The involvement of end users in the design of user interfaces is typically limited to the initial design and development phases. That approach is sufficient for interfaces that will reside on a single device and remain consistent across contexts, but it is problematic for interfaces that divide across devices because the appropriate division may depend heavily on the situation at run-time.

The appropriate division for an interface depends on a variety of factors. The number and types of devices will dictate the set of possible divisions. The physical setting of the devices and users' particular preferences will shape the division of functionality across devices. A physically distant monitor might only be useful for displaying overviews of information, rather than details, while a user who does not like working with a stylus might be reluctant to employ

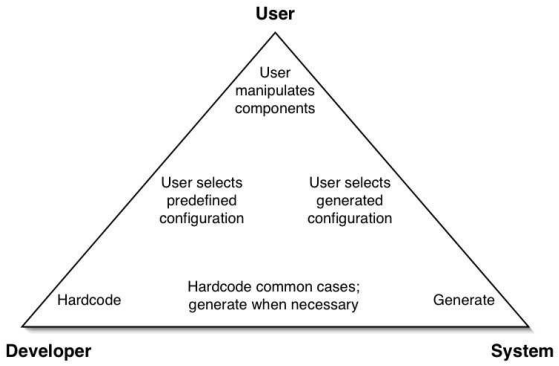

Figure 1: The responsibilities for designing interfaces are shared between the user, system and developer.

a tablet PC for input. The context of interaction can also affect the appropriateness of a particular division; users might want to divide an interface onto a large display in different ways depending on whether the display is in a public or private space. The level of trust that users place in different devices will also affect how they want to divide an interface. The types of information that they will want to transfer to a device will vary when they own the device, when their company owns the device, and when the device is owned by another (possibly rival) company.

The dependence of the appropriate division on the run-time situation means that neither the developer nor the system can completely determine how to best divide an interface. Only the user knows, for example, how much they trust the involved devices. As a result, we need to adjust the process of determining the design of an interface to incorporate the user as well as the developer and system (see Figure 1). A successful framework for supporting divisible interfaces should *allow users to affect the allocation of interface components across devices* based on how they want to divide subtasks across devices. It should also *allow users to affect the functionality of interface components on different devices*, particularly to allow users to restrict or modify the functionality of interface components on untrusted devices. Finally, the framework should *allow users to affect the content of interface components on different devices* in order to allow users to restrict the information available to untrusted devices and to modify the information displayed in public or semi-public contexts.

While supporting some user input is necessary for appropriately dividing interfaces, requiring too much user input is problematic. Users want to spend their time working with a divided interface, not specifying how to divide it. We believe that an appropriate balance of control for the common case would be for the developer to identify likely divisions, behaviors, and content; for users to choose among the possibilities; and for the system to adapt the interface composition based on device capabilities. Each entity can then make the most effective use of their particular knowledge: developers know how to build effective interfaces, users know how they want to work with devices, and systems know about the capabilities of devices.

When we allow users some control over the division of interfaces, we must accept that users may make mistakes. While poor division choices that unnecessarily reduce the effectiveness of an interface are unfortunate, poor choices that negatively impact privacy and security are more serious. A successful framework should therefore *make the flow of commands and information between devices transparent* in order to allow users to detect an inappropriate behavior. While users would ideally be able to stop inappropriate behaviors before they have an appreciable impact, increasing the user's awareness of device activities should at least help them choose a more appropriate level of trust in the future.

Developer Requirements

A framework that successfully supports the creation of divisible interfaces must be *expressive* enough that developers can build the interfaces they envision. Developers will not adopt and use a framework that restricts them to a small set of interface components or that prevents them from building complex interfaces.

While building a divisible interface will obviously entail more work that building a traditional single-device interface, a framework should *minimize additional work by developers*. In particular, when building interfaces with W possible allocations of components, X possible behaviors for those components, Y possible variants of content for those components, and Z possible types of devices, developers must not need to create W*X*Y*Z separate interfaces.

As part of minimizing additional work, a framework should be *amenable to tool support*. Developers should be able to create and use tools that simplify or expedite the process of building an application on top of the framework. For example, it should be possible to create a graphical user interface (GUI) builder that helps developers arrange interface components and specify possible divisions through direct manipulation.

Deployment Requirements

A successful framework must allow developers to build divisible interfaces that users actually want to use, but it must also be widely deployed so that users can leverage the ability to divide interfaces in as many situations as possible. Because the capabilities of computing devices in our environments vary widely, a framework must be *lightweight*: it must make minimal assumptions about the capabilities of devices and possess a relatively small footprint in order to allow users to divide interfaces onto a wider variety of devices.

User interfaces, interaction techniques, computing devices, and I/O resources are moving targets. A framework that supports the current state-of-the-art will not succeed over time if it is not *extensible*. That extensibility should also not be at the expense of backward compatibility; develop-

ers should not need to rewrite applications or restructure existing interface divisions when someone extends the framework to incorporate a new type of device or support for gracefully degrading interfaces [2] across classes of devices.

While the framework must provide sufficient value to the end users actually working with divided interfaces, it must also *provide privacy and security protection for device owner*s. Device owners will not make their devices available for annexing if they believe that users could steal information from those devices or otherwise compromise them.

DIAMOND

DIAMOND is an implementation of a framework for Dividing Interfaces Across Multiple Opportunistically annexed Devices. We designed DIAMOND both as an initial effort at meeting the requirements we identified for successfully supporting divisible interfaces and as a tool to allow us to rapidly explore the design space for divisible interfaces. In this section we describe a sample user experience for an application we built using DIAMOND, and we discuss DIAMOND's implementation.

User Experience

Returning to the scenario we raised in the Introduction, the user could have had a much different experience if she had an email application built with DIAMOND on her cell phone. Instead of relying completely on a nearby PC to access her email, she could choose to annex the PC and interact with her email using both devices. DIAMOND allows the user to access her email account through her cell phone, regardless of which device she uses to display the inbox and messages. By accessing email from her trusted device, she avoids the possibility of compromising her security by entering her email password on the PC.

After viewing her inbox on her cell phone, the user could decide how she wants to take advantage of the available I/O resources. If the inbox does not appear to contain sensitive information, the user might choose to push all interactions with her email over to the PC, where she could take advantage of the superior I/O resources (Figure 3). If the user was concerned about the trustworthiness of the PC, she could choose to use her cell phone to view the information flowing between the devices. Monitoring that information could alert the user if the PC tries to access data that the user did not intend to share. Going a step further, the user could choose to use her phone to approve or disapprove all information requests from the desktop.

In situations where the user does not trust the PC or does not want to publicly display the messages in her inbox, she could choose to divide the interface between the devices. Figure 2 shows the process of choosing the options for division. To avoid transferring the information in her inbox to an untrusted device, the user might choose to keep the inbox on her cell phone and only take advantage of the desktop to reply to selected messages. The user could also tailor the interface behaviors to match her level of trust.

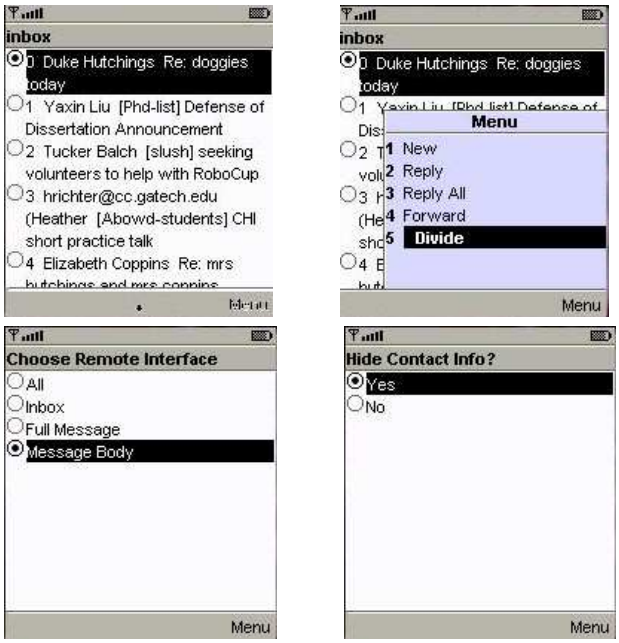

Figure 2: After viewing the inbox on her cellular phone, the user may chose to divide the email interface by sending the Message Body to a desktop with the contact information hidden in the message.

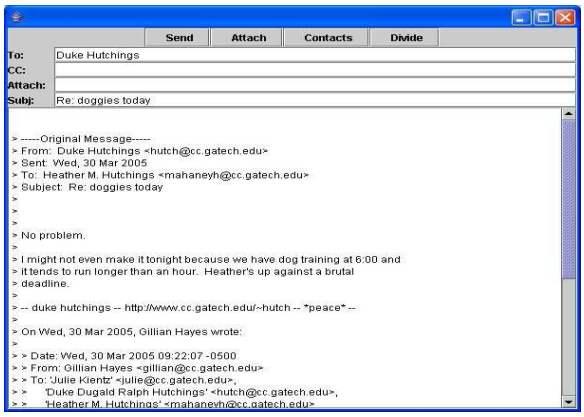

Figure 3: The user annexes desktop to reply to an email received on her cell phone.

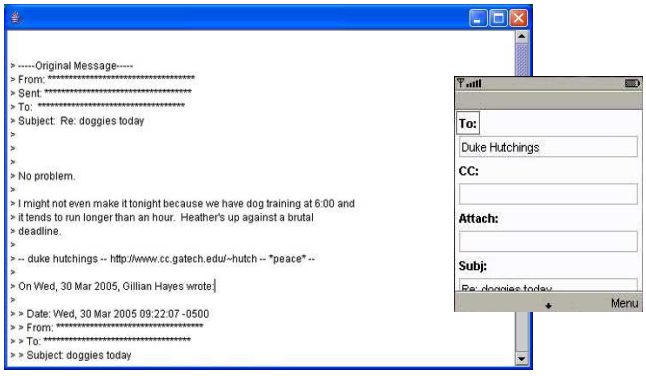

Figure 4: The user divides the reply window between the desktop and cell phone to take advantage of I/O resources while protecting contact information.

When annexing an untrusted device, the user may disable the ability to automatically query the address book when she enters text in the address fields. To avoid transferring contact information, the user could divide the reply window, to keep the in the message headers on the cell phone (Figure 4). Additionally, the user could to alter the content displayed in the interface to obfuscate contact information in the message body. By allowing the user to divide her inbox and messages across devices, DIAMOND enables the user to tailor her interface to meet her current needs.

Implementation

DIAMOND's implementation consists of XDL, an XML Interface Description Language; a distributed tuple space for routing information between modules and devices; and the XDL Composer, Model, and Interface Interpreter modules.

XDL: XML Interface Description Language. Like previous systems (e.g. [1,15,19]), DIAMOND developers use an XML-based Interface Description Language (XDL) to describe interfaces. Interface Description Languages allow developers to create high-level descriptions of interface elements, leaving the exact rendering of the interfaces to the system at run-time. XDL supports the description of interface components, behaviors, and division points, and provides abstractions to support user choices for interface configuration at run-time.

Using XDL, a developer creates a single interface description that can be rendered on and divided among multiple types of devices. Individual devices interpret the interface description and render an interface using a UI library, such as Swing or MIDP, available for their platform. Specifying interfaces at this level greatly reduces the additional work that developers must perform to create interfaces for multiple devices.

```
<button number="1" text="Reply" positionX="3"
   postionY="0" weightX="0" weightY="0">
   <click behavior="makeNewFrame"
      template="replyFrame" args="0,3">
   </click>
</button>
```

The lowest level of description in XDL is the *component* or widget. XDL allows developers to specify a component's attributes and layout within an interface. The example above shows the description for the "Reply" button on an email interface. The button element specifies the text and layout attributes for the button within a panel. XDL provides support for most common interface components, including text fields, tables, menus, images and button groups. We designed XDL to be extensible, so that developers can extend the language to support specialized or custom components.

In addition to describing an interface's components, developers also describe the *behaviors* for those components in XDL. The click element above specifies the "Reply" button's behavior. When a user clicks the "Reply" button, DIAMOND should make a new frame (or window) for the reply. XDL supports a range of common component actions such as hover and key pressed. Behaviors allow developers to specify that an interface reacts to user actions by setting attributes of interface components, displaying new interfaces, or returning information about the interface state to the application.

Developers organize components in XDL into *panels* and then into *frames*. The panels and frames represent the division points in the interface. By organizing components into panels, developers can insure that certain elements always remain together when a user divides the interface. For example, a textbox and its label should always appear together so that the interface remains comprehensible to the user.

While the developer can specify many attributes of interfaces at design-time, XDL must also allow developers to indicate where information from the application should be inserted into the interface at run-time. XDL uses *variable* elements to set the attributes of interface components that depend on application state. Variable elements contain references to the name of the class and method where DIAMOND can find the appropriate data for the interface at run-time. As shown in Figure 5, when creating the message body of an email reply window, a variable element can be used to indicate that the content of message body should be inserted into the text attribute of a text area

In addition to obtaining information from the application at run-time, XDL must also allow users to affect the division of interfaces at run-time. Developers use *choice* elements to indicate options that the user can choose from when dividing an interface. Choices can provide the user with options for specifying which device a component should appear on, how a component should behave, and what content a component should contain. In Figure 5, the value of choice element 0 indicates whether the message body corresponds to a new message, or a reply. Developers create an initial interface configuration by giving each choice a default value. XDL uses *if-statement* elements to adapt the interface to the user's choices at run-time. Each time DIAMOND creates an interface, it evaluates the if-statements to generate an interface corresponding to the user's selections. For example, if the user chooses to compose a new message, no text is inserted into the text area. If she decides to reply to a message, the content of the previous message must be retrieved from the application.

DIAMOND uses *selection interfaces* to present choices to the user at run-time. Selection interfaces display the options that the developer provides for each choice. For example, the selection interfaces in Figure 2 allow the user to choose the location and content of the different interfaces in an email program.

Because XDL interface descriptions can contain conditional elements, we distinguish between *templates* and *concrete interface descriptions*. Templates incorporate conditional elements (variables, choices, and if-statements), as

```
<panel number="2" positionX="0" positionY="2" weightX="0" weightY="1" layout="gridbag">
 <if choice="0" value="0"> //message type =new
  <textArea number="0" positionX="0" positionY="0" weightX="1" weightY="1"  width="600"
   height="400" text=""/>
 </if>
 <elsif choice="0" value="1"> // message type =reply
  <if choice="7" value="1"> // hide contact =no
   <textArea number="0" positionX="0" positionY="0" weightX="1" weightY="1"text="">
    <variable name="text" class="model" method="getReply" args="">
     <variable name="args" class="inbox" method="getSelectedMessage" args="row"/>
    </variable>
   </textArea>
  </if>
  <elsif choice="7" value="0"> // hide contact =yes
   <textArea number="0" positionX="0" positionY="0" weightX="1" weightY="1"text="">
    <variable name="text" class="model" method="obfuscatedReply" args="">
     <variable name="args" class="inbox" method="getSelectedMessage" args="row"/>
    </variable>
   </textArea>
  </elsif>
 </elsif>
</panel>
```

Figure 5: An XDL template for the message body of an email interface.

```
<panel number="2" positionX="0" positionY="2" weightX="0" weightY="1" layout="gridbag">
 <textArea number="0" positionX="0" positionY="0" weightX="1" weightY="1"text="No problem. I
   might not even make it tonight because we have dog training at 6:00 and it tends to run
   longer than an hour … "></textarea>
</panel>
```

Figure 6: A concrete XDL description for the message body of an email interface generated by inserting information from the application and the user into the template in Figure 5.

well as frame, panel, component, and behavior elements. At run-time, DIAMOND takes an interface template and draws on information from the application and user to replace the conditional elements with concrete details. A device then uses the resulting concrete interface description to instantiate and render the desired interface. Figure 5 and Figure 6 show an example of a template at design-time with the concrete description generated by DIAMOND at run-time.

Tuple Space. Dividing interfaces across devices requires a network infrastructure that routes interface descriptions and information requests to the correct devices. We designed DIAMOND with a point-to-point connection model because we did not want to depend on the existence of shared infrastructure, such as servers or memory space, in the environment. In DIAMOND, we use a modified version of LiME to handle network connections and information transfer between devices. LiME is Java-based middleware that coordinates applications in ad-hoc, mobile environments [8]. We altered LiMEs discovery and group coordination mechansims to support direct connections between devices rather than multicast beaconing. We made these changes because we want users to explicitly choose which devices annex rather than allowing connections with unauthorized devices. Additionally, eliminating multicasting was necessary when porting LiME to J2ME.

LiME relies on a light-weight tuple space named LighTS [16] to create the illusion of a global shared memory space between applications on multiple devices. Applications built on top of LiME use agents to read and write tuples. Each agent has its own local tuple space, which may be either private or shared. Multiple agents, residing on either a single device or multiple devices, may share a tuple space to allow them to communicate. LiME manages the connections and data transfer between the tuple spaces to create the appearance of a global shared tuple space. This model simplifies application development by allowing developers to design applications without knowing whether the application's components will reside on a single device or on multiple devices at run-time.

DIAMOND regulates the transfer of sensitive information across devices by tagging tuples with the intended device's identity. These tags indicate to LiME whether a tuple should remain in the local tuple space or if it should be transferred to a particular device. By using a wild card tag, DIAMOND can also allow non-sensitive tuples to be sent to all connected devices.

We chose to implement information transfer in DIAMOND using the shared memory model of a tuple space because it allowed us to satisfy several framework requirements. The ability of agents can to tuples from the tuple space without removing them or altering the behavior of other agents facilitates extensibility. This property also facilitates transparency, because both applications and external tools can easily monitor and display to the user the flow of information between devices. The tuple space model also reduces the developer's burden by making it easier to write applications that work on both single and multiple devices.

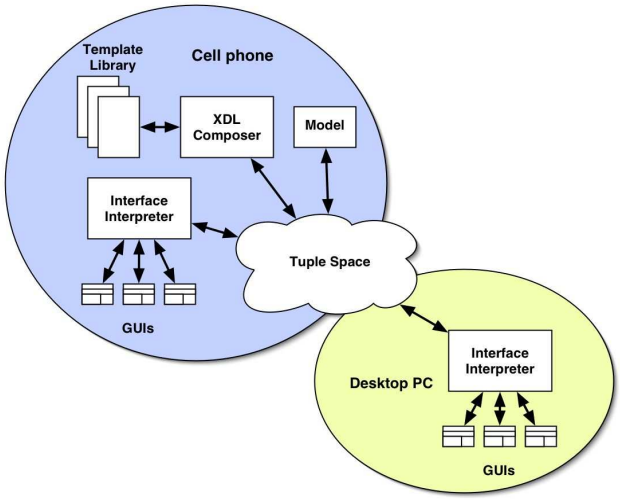

Figure 7: The general architecture of the DIAMOND framework.

Modules. An application running on top of DIAMOND consists of three decoupled modules that interact through a shared tuple space (Figure 7). DIAMOND provides the XDL Composer module, which generates concrete interface descriptions using templates in the Template Library, and the Interface Interpreter module, which interprets the concrete descriptions and renders the resulting interface. The application's developer is responsible for providing the Model module, which contains the application's functionality, and the XDL templates that describe any custom interfaces for the application.

In order to create a new interface, the XDL Composer module reads the interface's frame and panel templates from the Template Library. The XDL Composer receives the name of the frame template when an application inserts an interface request into the tuple space. To create a concrete description, the XDL Composer must obtain the values for the variables from the Model by placing requests in the tuple space. The XDL Composer initially uses the default values for the interface's choice elements to create the concrete interface description, which it then inserts into the tuple space. The default values provided by the designer are intended to insure that the interface initial interface is usable on the current platform. When the user's run-time actions require changes to a displayed interface, the XDL Composer creates a new concrete description that causes either updates to existing components or the display of additional components (such as popup menus or dialog boxes).

When the user chooses to divide an interface, the XDL Composer is also responsible for creating concrete descriptions of the selection interfaces. DIAMOND includes several selection templates for different platforms, and developers may add additional templates. Once the user chooses the options for the interface, the XDL Composer uses the choices to create new concrete descriptions for it.

The Interface Interpreter module interprets and renders the concrete interface descriptions generated by the XDL Composer. While the XDL Composer need only reside on one device, every device that wishes to display interfaces must run an Interface Interpreter. Interface Interpreters are device-platform and UI-toolkit specific. Our current implementation of DIAMOND provides a desktop Interface Interpreter that translates descriptions into interfaces using Java's Swing Toolkit and a cell phone Interface Interpreter that employs MIDP 2.0. The Interface Interpreter is also responsible for interpreting the user's interactions with an interface. The interpreter listens for events from the user interface and triggers the specified behaviors by placing information in the tuple space. Behaviors may require obtaining application data from the Model, obtaining new interface descriptions from the XDL Composer, or updating the other modules with the current state of the interface.

DIAMOND supports reusability by using the XDL Composer and Interface Interpreter modules, as well as the Template Library, across applications. Because DIAMOND applications use the Model, View, Controller (MVC) design pattern, developers must build an application-specific Model containing the functionality, data, and state for the application. The developer also designs new XDL templates for the application-specific Views and inserts them in the Template Library. Finally, the developer creates a Controller to handle method requests from the XDL Composer and Interface Interpreter at run-time.

The Controller is actually part of a Control Agent that mediates between the Model and the tuple space. Each module in DIAMOND has a corresponding agent that acts as an intermediary between the module and the tuple space (Figure 8). The agents are responsible for putting tuples in the tuple space that initiate actions, such as information requests and interface creation, by other modules. The agents also listen for tuples for their module. For example, the Control Agent listens for tuples corresponding to each

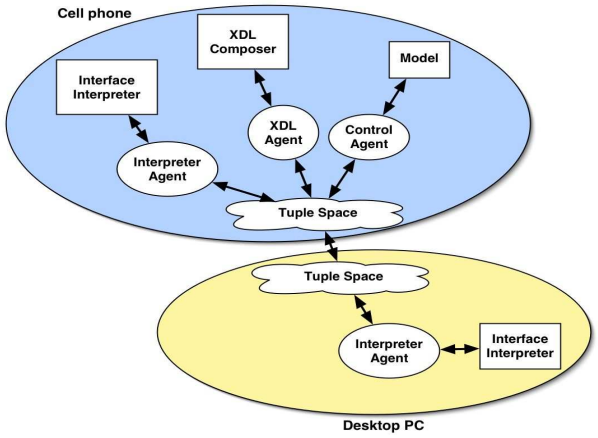

Figure 8: The detailed architecture of the DIAMOND modules, agents and tuple spaces.

public method in the application Model, and the XDL agent listens for tuples indicating requests for new interfaces. When the agent receives a matching tuple, it calls a method in its module to transfer the data obtained from the tuple.

Example Applications

In addition to the email application that we used to illustrate the user experience of working with a DIAMOND application, we have also used DIAMOND to create a variety of other applications, including a calendar, an information flow monitor, and a multi-device drawing program.

A divisible calendar application residing on a PDA or cell phone could allow users to more easily coordinate their schedules with others without unnecessarily revealing private information. By dividing the interface to show an overview of the schedule on a larger display and details on the personal device (Figure 9), the users could easily negotiate a meeting time without revealing the private details of their schedules.

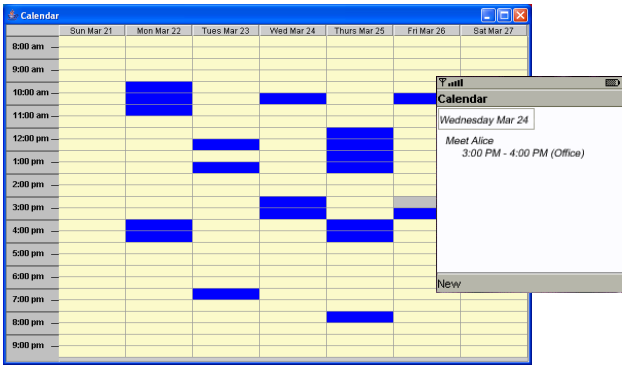

Figure 9: The divisible calendar application allows users to share overview of their schedules on annexed devices while viewing details on their cell phones.

While we have concentrated our discussion on using DIAMOND to divide interfaces across personal devices and devices in the environment, users can also divide interfaces solely across their personal devices. As a simple example, a user editing an image on her tablet PC while sitting near her desktop computer might want to zoom in on the image using her tablet PC for more control over pixel-level changes while showing an overview of the image on her desktop display to assess the impact of those changes. We implemented a simple drawing program on top of DIAMOND that allows users to provide such focus and context views across multiple devices (Figure 10).

Discussion

In Table 1, we examine how our implementation of DIAMOND attempts to support the user, developer and deployment requirements for interface division. Because we focused our development efforts on enabling the user experience, we feel that our frame work leaves the most room for improvement in meeting the deployment requirements, especially providing privacy and security protection for device owners. We are also currently working to

Figure 10: The drawing application allows users to take advantage of the Tablet's pen input while viewing changes to the image on the desktop.

further simplify the design process by developing a tool which extends the Eclipse Visual Editor [26] to generate XDL templates.

FUTURE WORK

We have identified risks of transferring interaction to a single, potentially untrustworthy device. DIAMOND provides a mechanism of overcoming that risk through interface division. However, because we rely on the user to divide interfaces, we do not guarantee that the interface division will overcome these risks. In order for users to make appropriate choices when dividing interfaces, they need to understand what risks are involved. We plan to conduct studies to explore whether and how users think about the risks of interacting across devices.

Once we understand how users think about threats to privacy and security, we can explore how to best inform users of the risks and benefits that may result from the division. We plan to develop and test more classes of selection interfaces to explore methods of presenting tradeoffs to users. We will focus on creating easily usable methods of choosing division which provide users with knowledge of how their choices will affect the layout, content and behavior of the interface and the risks involved in transferring information to another device.

DIAMOND relies on the existing infrastructure for discovery and addressing of devices to annex. While we believe that is sufficient, we might eventually extend the framework to incorporate better methods such as beaconing [5] and bumping [3] that researchers have proposed to allow users to explicitly indicate the device they wish to annex.

DIAMOND is currently built assuming that users push interface elements to devices rather than pulling them [17], avoiding the need for users to authenticate to their device. DIAMOND also assumes that users will not need to authenticate to devices in the environment. We should even-

Class	Requirement	Supported By
User	Allow users to affect the allocation of interface components across devices.	**Selection Interfaces** created with **XDL** allow users to specify the locations of the frames and panels.
	Allow users to affect the functionality of interface components on different devices.	**Selection Interfaces** created with **XDL** allow users to specify the behaviors of interface components.
	Allow users to affect the content of interface components on different devices	**Selection Interfaces** created with **XDL** allow users to specify content of interface components.
	Make the flow of commands and information between devices transparent.	The **Monitor** tool allows users to view the most recent commands in the **Tuple Space**.
Developer	Expressive enough that developers can build the interfaces they envision.	**XDL** supports common interface components and behaviors.
	Minimize additional work by developers.	**XDL** allows designers to create an interface once for multiple devices. The **Tuple Space** model abstracts networking and data transfer.
	Amenable to tool support	The **Tuple Space** model supports addition of new tools without altering application performance.
Deployment	Light-weight	**Point-to-Point** architecture does not require environmental support. **Tuple Space** model works seamlessly without or without networking.
	Extensible	**Decoupled Architecture** allows developers to alter and extend DIAMOND components without requiring changes to applications.
	Provide privacy and security protection for device owners.	DIAMOND makes sandboxing straight-forward, although we have not yet implemented it.

Table 1: We developed DIAMOND to support interface division focusing on the requirements for users, developers and deployment.

tually extend DIAMOND to support both push authentication [13] and authentication to annexed devices.

CONCLUSIONS

In this paper, we present the DIAMOND framework for supporting applications with interfaces that may divide across multiple devices. We lay out requirements for infrastructure support for interface division which focus on the user, developer and deployment. We explore a representative user experience enabled by DIAMOND and explain the implementation of the framework and describe sample applications built on top the framework. We discuss how DIAMOND meets the requirements we laid out for interface division and how our future efforts will focus on better addressing these requirements. By allowing interface division across trusted and untrusted devices, we believe DIAMOND will facilitate applications that make effective use of I/O resources on multiple devices.

ACKNOWLEDGMENTS

The authors would like to acknowledge Microsoft Research for their gift supporting this research.

REFERENCES

1. Bojanic, P. The Joy of XUL. The Mozilla Organization. December, 2003. http://www.mozilla.org/projects/xul/joy-of-xul.html
2. Florins, M., Vanderdonckt, J. Graceful degradation of user interfaces as a design method for multiplatform systems. In *Proceedings of IUI 2004*, pp140-147.
3. Hinckley, K. Synchronous Gestures for Multiple Users and Computers. In *Proceedings of UIST 2003*, pp. 149-158.
4. Johanson, B., Pennekanti, S., Sengupta, C., and Fox, A. Multibrowsing: Moving Web Content across Multiple Displays. *In Proceedings of Ubicomp 2001*, pp. 346-353.
5. Kindberg, T., Barton,J et al. People, Places, Things: Web Presence for the Real World. In *Proceedings of WMCSA 2000*, pp 365-376.
6. Kozuch, M., Satyanarayanan, M., Bressoud, T., Helfrich, C., and Sinnamohideen, S. Seamless Mobile Computing on Fixed Infrastructure. In *IEEE Computer*, July 2004, pp. 65-72.
7. Li, D. and Li, R. Transparent sharing and interoperation of heterogeneous single-user applications. In *Proceedings of CSCW 2002*, pp. 246-255.
8. Murphy, A. L., Picco, G. P., Roman, G-C. Lime: A Middleware for Physical and Logical Mobility. In *Peoceedings of ICDCS* pp 524-533.
9. Myers, B., Lie, K.P., and Yang, B.C. Two-Handed Input Using a PDA and a Mouse. In *Proceedings of CHI 2000*, pp. 41-48.
10. Myers, B.A. Using Handhelds and PCs Together. In *Communications of the ACM*, 44(11), 2001, pp. 34-41.

11. Olsen, D.R., Jefferies, S., Nielsen, T., Moyes, W., and Fredrickson, Cross-Modal Interaction using XWeb. In *Proceedings of UIST 2000*, pp. 191-200.

12. Olsen, D.R., Nielsen, S.T., and Parslow, D. Join and Capture: a Model for Nomadic Interaction. In *Proceedings of UIST 2001*. pp, 131-140.

13. Patel, S.N., Pierce, J.S., and Abowd, G.D. A Gesture-based Authentication Scheme for Untrusted Public Terminals. In *Proceedings of UIST 2004*, 157-160.

14. Pering T., Sundar, M., Light, J., and Want, R. Photographic Authentication through Untrusted Terminals. In *IEEE Pervasive Computing: Mobile and Ubiquitous Systems*, October 2002, pp. 30-36.

15. Phanouriou, C., Abrams, M. Uiml: a device-independent user interface markup language. PhD thesis, 2000.

16. Picco, G. P., Balzarotti, B., Costa, P. LighTS: A Lightweight, Customizable Tuple Space Supporting Context-Aware Applications. To appear inn *Proceedings of 20th Annual Symposium on Applied Computing (SAC 2005).*

17. Pierce, J.S. and Mahaney, H.E. Opportunistic Annexing for Handheld Devices: Opportunities and Challenges. *Human Computer Interaction Consortium 2004.*

18. Ponnekanti, S.R., Lee, B., Fox, A., Hanarahan, P., Winograd, T. ICrafter: A service framework for ubiquitous computing environments. In *Proceedings of Ubicomp 2001*, pp. 56-75.

19. Puerta, A., Eisenstein, J. XIML: a common representation for interaction data. In *Proceedings of IUI 2002*, pp. 214-215.

20. Newman, M.W., Izadi, S., Edwards, K., Sedivy, J.Z., and Smith, T.F. User Interfaces When and Where They are Needed: An infrastructure for Recombinant Computing. In *Proceedings of UIST 2002*, pp. 171-180.

21. Nichols, J., Myers B. A., Higgins, M., Hughes, J., Harris, T. K., Rosenfeld, R., and Pignol, M. Generating Remote Control Interfaces for Complex Appliances. In *Proceedings of UIST 2002*, pp.161-170.

22. Want, R., Pering, T., Danneels, G., Kuman, M., Sundar, M., and Light, J. The Personal Server: Changing the way we think about ubiquitous computing. In *Proceedings of Ubicomp 2002*, pp. 194-209.

23. Rekimoto, J. A Multiple Device Approach for Supporting Whiteboard-Based Interactions. In *Proceedings of CHI 1998*, pp. 344-351.

24. Richardson, T., Quentin, S., Wood, K.R., and Hopper, A. Virtual Network Computing. In *IEEE Internet Computing*, 2(1), 1998, pp. 33-38.

25. Tan, D.S., Meyers, B., and Czerwinski, M. WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. In *CHI 2004 Extended Abstracts*, pp. 1525-1528.

26. Visual Editor Project. Eclipse Project. March, 2005. http://www.eclipse.org/vep.