Towards Multi-Domain Collaboration Toolkits

Jacob Bartel
Computer Science Department
University of North Carolina
Chapel Hill, NC 27514 USA
+1 919 962 1890
bartel@cs.unc.edu

Prasun Dewan
Computer Science Department
University of North Carolina
Chapel Hill, NC 27514 USA
+1 919 962 1823
dewan@unc.edu

ABSTRACT
A multi-domain collaboration toolkit hides heterogeneity of user-interface toolkits and associated domains from both programmers and end users of collaborative, widget-synchronizing, applications. We have developed such a system for the stand-alone, Eclipse, and web domains; and the AWT, Swing, SWT, and GWT single-user toolkits associated with these domains. Several new concepts are supported to meet these requirements including a widget server allowing a distributed widget client to manipulate widgets on an interactive device, flexible widget synchronization, flexible placement of widget listeners, "piping" centralized non-interactive replicas communicating with interactive user replicas, factory-based retargeting of the user-interface toolkit, and a new process architecture.

Author Keywords
Heterogeneity; distributed user-interfaces; user-interface toolkits; web; multi-device interfaces

ACM Classification Keywords
D.2.2 [Software Engineering] Tools and Techniques –user interfaces

General Terms
Design; Performance

INTRODUCTION
All forms of computer systems, such as hardware systems, programming/command languages, and operating/ database systems, exhibit some degree of heterogeneity, which can be defined as the existence of different concrete mechanisms for implementing the same abstract concept. Heterogeneity has led to efforts to hide aspects of it from users of these systems. These efforts have had two main goals. First, allowing developers to create a single unifying implementation of some functionality for heterogeneous systems. Second, allowing heterogeneous systems to work together or interoperate with each other.

CSCW'12, February 11–15, 2012, Seattle, Washington, USA.

This paper focuses on unifying and interoperating collaboration capabilities that hide heterogeneity of user-interface toolkits and associated domains such as standalone and web applications. The unifying capabilities would allow collaboration toolkits built on top of different user-interface toolkits/domains to share some or all of their code. The interoperation mechanism would allow heterogeneous widgets, and ideally, also widget compositions, to be synchronized with each other. The unifying and interoperation goals can be met independently. However, we consider both goals in this paper, because, as we see below, a common set of concepts can be used to address both of them.

A more intriguing goal is to create a *cross fertilizing* heterogeneous toolkit. The requirement was first articulated in the context of the work of Heering and Klint[1] to unify command, programming, and debugging languages into a single "monolingual environment". As Heering and Klint argue, even if it is not possible to develop a practical unified system, the attempt to integrate systems in different domains can lead to a cross fertilization in which crucial features found in one domain are incorporated as useful features in another domain.

In the rest of the paper, we expand on what it means to achieve these three main goals, and describe a first-cut system for realizing them

SCOPE
Before we can discuss our solution, we need to better describe the problem and solution requirements. We begin by explaining the terms *user-interface toolkit*, *domain*, and *collaboration*.

Single-User Layers and Toolkits
In general, the I/O of a single user is processed by several user-interface layers. A *framebuffer* treats the screen of the user as a two dimensional array of pixels, allowing higher layers to (a) access and manipulate these pixels, and (b) intercept keyboard and mouse events. A *window system* divides the screen into smaller regions, called windows, allowing drawing of text and images in a window and interception of window-specific events, such as typing and mouse clicks in a window or resizing and movement of a window. A *user-interface toolkit* is a layer above the

window system that divides windows into finer-grained abstractions such as text-boxes, sliders, and menus, and allows interception of widget-specific events such as typing and commitment of text in a text-box. A *view* is a layer between the toolkit and the semantic component of an application, called the *model*. The view composes widgets into a user-interface, and keeps the model and user-interface state consistent.

User-interface toolkits can create stand-alone or plug-in widgets in some applications. We shall refer to each environment in which widgets are created as a *domain*. Our work is currently restricted to Java applications, and addresses four popular Java toolkits: AWT, Swing, IBM's SWT, and Google's GWT. Together, these define three domains: the (desktop) stand-alone, (browser) web, and Eclipse domains. AWT and Swing support both stand-alone widgets and web-browser plug-in widgets in an applet. SWT supports both stand-alone widgets and plug-in Eclipse widgets. GWT supports plug-in (web) browser widgets by converting Java to AJAX-based JavaScript We have not yet fully targeted toolkits developed for mobile computers. We have developed single-user support for the Android toolkit, but not collaboration support, because certain Java features on which our implementation currently depends are missing in Android. However, the concepts described here should work for mobile user-interface toolkits.

In all of these toolkits, the client and user-interface toolkit layer run in the same process and hence host. Each toolkit (a) supports calls to instantiate or change a widget and associate a widget with observer or *listener* client objects, and (b) each announces widget events to interested listeners. All of them assume a single user views and manipulates each widget. In the rest of the paper, we shall assume this model of a single-user user-interface toolkit.

Collaboration Tools

A collaboration tool allows sharing of abstractions in one or more user-interface layers. There are unique advantages and disadvantages of sharing each layer [2]. In particular, sharing the user-interface toolkit layer allows non WYSIWIS (What You See is What I See) collaboration, and does not require special abstractions designed for collaboration. Sharing the framebuffer or window layer forces near-WYSIWIS collaboration. Sharing higher layers (a) does not allow sharing of toolkit events not intercepted by the higher layers, such as scrolling and incremental changes to a text field widget; and (b) constrains the abstractions that can be used to create the view and/or model. Thus, for each user-interface toolkit, it is important to offer a collaboration tool that allows sharing of its widgets.

A *layer-independent collaboration tool* offers general synchronization mechanisms such as remote procedures and shared objects [3, 4], which can be used by application programmers to manually intercept and share events/calls of one or more layers among different users. A *layer-dependent collaboration tool*, on the other hand, understands the events and calls of the layer on which it depends, and provides automatic sharing of events/calls of these layers, possibly using abstractions of a layer-independent tool. By a *collaboration toolkit*, we mean a layer-dependent collaboration tool that automatically shares events/calls of one or more user-interface toolkits.

The applications supported by a collaboration toolkit are *collaboration-aware* or *collaboration-transparent* based on whether they are aware that they are being used by multiple users. It is possible to support collaboration transparency at all layers. NetMeeting and Suite [5] are examples of desktop/window and model sharing systems, respectively, that support collaboration-transparent applications. It is also possible to transparently make individual single-user applications collaborative without changing them – for example, as shown in [6], Microsoft Word and PowerPoint. Any tool that provides tailoring of collaboration functionality supports collaboration-aware applications. As we see below, the cross fertilization goal requires a small amount of collaboration awareness in the application. Therefore, rather than using the dichotomy of collaboration awareness and transparency to evaluate the automation of our tool, we use the "proportional effort" requirement given below.

Requirements of Multi-Domain Collaboration Toolkit

As mentioned in the introduction, ideally a multi-domain collaboration toolkit should support unification, interoperability, and cross fertilization. Below, we refine these abstract goals by outlining specific requirements such a toolkit should meet.

Unification: It should offer a single set of mechanisms for sharing widgets of each of the target user-interface toolkits in each of the domains in which these toolkits can be used. This requirement was personally motivated by a collaboration system we have worked on for several decades. We have had to continuously port it to the single-user toolkit "most in fashion" at that time. In particular, we have created versions of it for Motif, UIL, HTML, AWT, and Swing. Each of these resulted in a different code base. As it is was difficult to keep multiple code bases consistent, only one of these toolkits was supported at one time. This is not a problem if newer user-interface toolkits supersede previous ones; however, this is not the case today. For instance, there is no consensus today on whether AWT, Swing or SWT should be used to create standalone user-interfaces for a Java application. More important, as mentioned above, different plug-in domains offer different toolkits.

The problem of creating separate implementations of a collaborative system for each popular user-interface toolkit is not peculiar to our project. Consider an IM tool. Today we see independent implementations of it in web user interfaces (e.g. Gmail), in the Eclipse environment (e.g. Jazz [7]), and, of course, on the desktop (e.g. Windows

Live Messenger) as a stand-alone tool. An even more serious problem is that no collaboration toolkit has been built so far for the relatively new SWT and GWT toolkits. A single collaboration toolkit layered on top of Swing, AWT, GWT and SWT, supporting all of the domains in which these user-interface toolkits can operate, would solve the above two problems.

Compatibility: Such a collaboration toolkit must bridge the gap between both the target user-interface toolkits and domains. One way to bridge the domain gaps is to use the cross fertilization idea to make all domains equal from the point of a collaboration toolkit. This approach has been used in [8] to make web browsers directly communicate events with each other, much as stand-alone replicas in current collaboration toolkits do. However, it is not practical to change web browsers, and more important, violate web constraints. Therefore, we include the following broader compatibility requirement: The collaboration toolkit must follow domain constraints.

Cross fertilization: Cross fertilization to the web domain is consistent with the general view that the web is a liability for collaboration, though recent work has shown that this is not the case in many situations[9, 10]. We go a step further and suggest that it is, in fact, an asset, in that many aspects of it should be included in other domains. In particular, in all domains, it should be possible, as in web applications, to:

(a) *centralize communication* through a central server, as such communication provides more ordering guarantees, which can be exploited in replica consistency algorithms[11]; and does not require the user computer to accept connection requests, disallowed by certain firewalls. As centralized communication adds network latency to remote response times, it should be done only for those widgets such as text for which such guarantees are important.

(b) *centralize computation* in a central server, for several reasons. Certain resources, such as some files and databases, may not be available to stand-alone applications on a user computer. In addition, the computation may be expensive, and thus carried out faster on a powerful server than a slow user computer. Furthermore, certain computations such as file writes and email sends are not idempotent, that is, yield the same result regardless of how many replicas carry them out. As centralized computation adds network latency to response times, it should be done only when it is essential or beneficial.

(c) *use a generic program to join* a session so that all users in the session are not responsible for installing and keeping up-to-date application-specific code, which can be a heavyweight and error-prone task.

Begole et al[12] have shown that these are essential features when application code runs in a web browser, and we argue here that they are important convenience features in other domains.

Interoperation: It should be possible to support real-time collaboration among collaborators using different single-user toolkits and domains. Increasingly, the same application is being implemented in different domains (e.g. Microsoft's Word and Google Docs's web version of it, and the Google Map/Translate implementations on the web and various mobile devices). In various scenarios (some of which are given in [13, 14]), a group of collaborators may wish, or be forced, to use different domains and associated user-interface toolkits, and thus, have a need for such interoperation.

Multiple widget-compositions: Often different domains offer different widget-compositions for the same application (e.g. Google translate) that make use of the unique capabilities and constraints of the domain. However, synchronizing different widget compositions seems fundamentally at odds with collaboration toolkits. To illustrate, consider the classic case of an integer value being represented by a slider in one user-interface and a textbox in another. Synchronizing two different widget types directly requires a way to translate from one to another, which in turn, implies that it is implemented in a layer above the user-interface toolkit.

However, it is possible to use the following observation to relax the requirement in current collaboration toolkits that only identical widget-composition can be synchronized. The widget compositions may differ, not because the same abstraction is displayed by different types of widgets, but because certain user-interfaces (a) contain optional widgets not displayed in other interfaces, and/or (b) provide different layouts and composition of common widgets. Therefore, allowing collaboration among multiple (but not arbitrary) widget compositions is another requirement.

Controlled retargeting: As collaborators may interact from different domains, associated with different user-interface toolkits, and may also have different preferences for the user-interface toolkit, the programs run by them should be able to control which user-interface toolkit is used to create the user-interface. This requirement distinguishes our work from a multi-platform user-interface toolkit, such as AWT, which provides a logical layer that is targeted automatically at multiple physical user-interface toolkits offered by different platforms.

Proportional programming effort: The requirements above imply several customization capabilities, which increase programming overhead. This effort should be proportional to the amount of customization desired by the application. In particular, applications that wish the traditional semantics supported by existing collaboration toolkits should require no collaboration awareness.

No performance penalty: A collaboration toolkit meeting these requirements and targeted at a particular user-interface toolkit and domain should be able to offer the same performance as one designed for that user-interface toolkit and domain.

No superfluous constraint: The system must impose only those constrains required by the domains. In particular, it should not force two processes to (a) centralize a computation, if the computation can be safely replicated, and (b) centralize a communication, if it is possible for it to be safely done directly.

TECHNICAL CONCEPTS

To explain how we address these requirements, we start with an overview section describing the process architecture, and then individually address some of the components of the architecture.

To concretely understand the architecture and other concepts, let us consider an application, inspired by Google Translate, that allows English-speaking users to study Chinese by viewing together the translations of a collaboratively composed sequence of English words. It comes with two user-interfaces (Figure 1), a small and a large user-interface. The small user-interface presents, in a single column, two text fields for displaying an English phrase and its translation, and a button for performing the translation. The large user-interface uses text areas instead of text fields as the two text components, provides an extra button to clear these widgets, and lays widgets in a matrix rather than a column. The small user-interface has been developed for the mobile and Eclipse domains, where screen space is an issue, and the large one for other domains. The two interfaces are implemented by the classes, ASmallGUI and ALargeGUI, respectively.

Process Architecture

Like traditional collaboration toolkits, our system assumes that a user in a collaborative session runs some local process that joins the session and creates and manipulates local widgets for that user. Corresponding widgets created by different local processes in the same session are kept consistent with each other. Therefore, like other collaboration toolkits, we will refer to these processes as (local) *replicas*, even though they are not required to run the same code. Each replica has a module called a *widget server* module, which is provided by our collaboration toolkit. A widget server is like a window server in a network window system, distributing the user-interface toolkit rather than the window layer. This, it accepts remote calls from a widget client to create and update widgets, and sends widget events to the latter. The widget server module can execute as part of (a) a *generic session joiner*, provided by our toolkit, in which case it receives remote calls to both create the initial user-interface and make updates to it in response to remote actions, or (b) a program executing application specific-code such as ASmallUI and ALargeUI, in which case it receives calls only to update the user interface in response to remote actions.

For each kind of user-interface, a separate *piping replica* process is created, so named, because instead of making calls in a single-user toolkit, it directs them to the session server, which in turn forwards them to regular replicas to remotely create and update their user-interfaces. Thus, in this example, two piping replicas are created, for the small and large interfaces, respectively. As we support the web domain, we assume that each collaborative application is installed on some web server. In addition, we assume that existence of a *session server*, which (a) allows users of the application to form one or more sessions, (b) creates piping replicas for each session, (c) centralizes communication of information among replicas, and (d) and stores session state downloaded into latecomers.

All centralized processes must be located on well-known hosts. In our implementation, they all run on the machine hosting the web server. It is attractive to combine them all in one central process to reduce the startup costs. A session server and piping replica are separate because they execute toolkit-defined and application-defined code, respectively. A web server and other processes are separate because a web server can execute external code only when a web browser referencing that code connects to it; and we support sessions in which no web browser is involved.

Let us continue with the example to illustrate and further refine the architecture above. Assume that programmer Alice has just finished creating the latest version of the code. She installs it on a well known directory, *translator*, at a web server, www.univ.edu, associated with this application. In addition, she installs it as a plug-in in the Eclipse environment.

To join a new session with the application, she starts a local replica that makes the following call:

VirtualToolkit.join(joinDescription, replicaId, false, false)

All four arguments are passed by the local call to the remote session server. The two Boolean arguments indicate that by default, the communication between widget replicas is direct and the listeners of the widgets are replicated. "replicaId" is an optional argument, and is used by the local replica to register with the session server an address that can be used by other replicas to communicate with it directly. The join description consists of three parts: a web server address, and an application, session, and UI description, as shown below:

www.cs.univ.edu/~translator/?session=test1&kind=small

Here ~translator, session=test1 and kind=small are the application, session, and UI descriptions, respectively. The UI description is passed to the session server so that it can connect it to the appropriate piping replica. In response to (a) the first join request, the session server

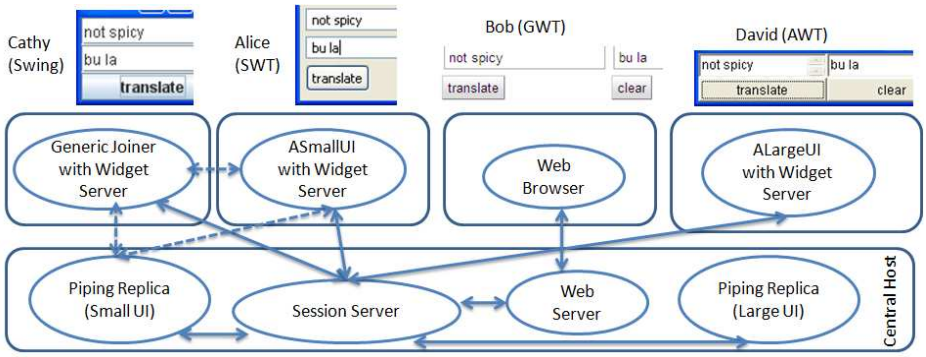

Figure 1 User Interfaces and Physical Process Architecture in Example Session

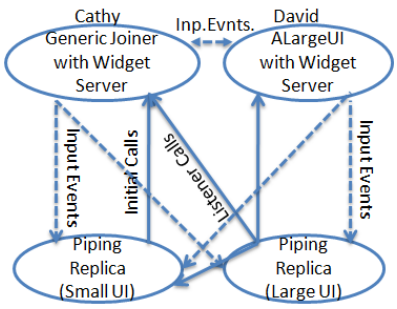

Figure 2 Logical Call/Event Flow

creates a new object to represent the session; and (b) all join requests, the requesters are added to the session object, and the replica address of the joiner, if provided, is sent to existing session members with registered replica addresses.

On the creation of a new session, the session server also starts the two piping replicas for the two user interfaces. A piping replica for a particular kind of user-interface executes the same code as a corresponding user replica, creating the same user-interface, and also has a replica address to communicate with the session server. A configuration file specifies the program(s) that the piping replicas run, runtime arguments to the program(s), and the widgets whose listeners they centralize. In our example, it states that for all widgets, the centralized listeners are located in the piping replica created for the large interface.

Alice next asks Bob, Cathy, and David to join her in testing the new version, communicating the session URL to them. David is on the file system referenced by the web server, so he simply runs the installed code on his machine to create a replica, binding it to AWT and the large interface. The other users are on separate file servers and have not installed the software. This is not a problem for Bob, who interacts through the web browser, which uses the web protocol to download code compiled into JavaScript by GWT, and binds it to the large interface. Cathy creates a stand-alone user-interface, and is also able to use a generic application, the generic session joiner mentioned above, and binds it to Swing and the small interface. David is behind a firewall that prevents incoming connections, so he does not provide a replica address when joining the session. As Bob uses a web browser, he too does not provide such an address. The other two users, not behind firewalls, provide such addresses, which are used to connect them to each other and the two piping replicas. Figure 1 shows the user interfaces and process architecture created for the resulting session, in which multiple users, single-user toolkits, widget compositions, and domains are involved. The dashed lines indicate connections among replicas. Such connections are not shown for the piping replica for the large interface to avoid further cluttering the figure. David and Bob's replicas have no such connections as they have not registered replica addresses.

Factory-based Retargeting

A collaboration toolkit supporting both replication and centralization of widget listeners must trap and distribute events and calls of the underlying user-interface toolkit. Like several previous works, we have developed an abstract user-interface toolkit layer that is mapped to multiple target user-interface toolkits, which allows us to meet the unification requirement by implementing the trapping/distribution support in this layer. There have been two main approaches for such abstraction.

One approach is to create a declarative re-targetable user-interface tool such as an XML-based system [15]. However, this approach fundamentally changes the way developers program, thereby also restricting the set of supported user-interfaces. For instance, it does not allow a program to dynamically add widgets in response to user input.

Another approach is to create a procedural abstract layer and require the programmer to use appropriate subclasses of it to choose the appropriate concrete implementation. This approach has been used in WAHID[13] to map abstract scrollbars and menus to different concrete implementations of them in a stand-alone application and a sketching tool. In a single-inheritance language, the class inheritance approach works only when the target widgets are not related by an inheritance hierarchy, as in the case of the scrollbar and menu widgets. For example, it cannot support a container widget that is a subclass of a component widget, as the former would have to now be a subclass of an abstract container class. Therefore, we have developed an alternate solution based on using (programming) interfaces, and two interface-based design patterns: a) factories [16], that is, objects that create other objects, and (b) abstract factories[16], that is, objects that select among different factories.

We have created an abstract widget interface hierarchy based on the class hierarchy of Swing widget classes. For instance, we have created interfaces, VirtualContainer and VirtualComponent, which declare the public methods of Component and Container, respectively; and made VirtualContainer a subtype of VirtualComponent. For each target toolkit, we have created *proxy toolkit classes* that delegate to corresponding classes of the toolkit. For

instance, we have created the proxy toolkit classes SwingContainer and SWTContainer to provide the Swing and SWT implementations, respectively, of the VirtualContainer interface.

Competing *toolkit-specific factories* are created for different versions of toolkit abstractions in different target toolkits, and are assigned to abstract factory classes, which are used, via static methods, to instantiate toolkit abstractions. For instance, in our example, the following call is used to instantiate the translate button:

ButtonSelector.*createButton*("translate")

It invokes the static method. createButton() on the abstract factory, ButtonSelector, which returns an instance of the abstract interface VirtualButton.

Abstract factories are initialized by functions that choose the toolkit, which in turn results in them being assigned the concrete factories of the chosen toolkit. For instance, if the Swing toolkit is chosen, then the abstract factory, ButtonSelector, is assigned an instance of the factory, SwingButtonFactory. Thus, the call given above asks the instance of SwingButtonFactory to create a button, which, in turn, returns an instance of the proxy class SwingButton. An operation such as addActionListener() or setName() on the proxy class (SwingButton) delegates to the corresponding operation provided by the target toolkit class (JButton). The proxy toolkit classes not only delegate to target toolkit classes, but also, as we see below, distribute toolkit events among collaborators and interpose toolkit-provided proxy listeners between widgets and their application-defined centralized listeners (Figure 3).

Requiring developers to type toolkit abstractions using interfaces and instantiate them using factories is arguably good programming practice, yet *none* of our target Java target toolkits define a factory or even an interface for *any* widget. We might have also ignored them in our API had they not offered a way to support controlled retargeting, a side effect of which is the ability to create better designed programs, consistent with the argument of Heering and Klint that the exercise of trying to unify a set of computer systems can result in enhancement of individual members of the set. To allow developers to use familiar APIs, we have built adapters that translate calls of an existing user-interface toolkit to our abstract toolkit-independent top layer (Figure 3). To use them, developers must change the name of the top-level package in their imports of classes/interfaces provided by the existing toolkit.

Not all of our target toolkits support the same set of abstractions. How one translates between heterogeneous toolkit abstractions is an issue that modern multi-platform toolkits [15] have addressed and is, thus, beyond the scope of our research. In our current implementation, if a target toolkit does not support the equivalent of a supported abstraction or operation, we simply return a null object or do nothing when the abstraction is instantiated and the operation is invoked, respectively. If an application uses some functionality that is common to a subset of the target user-interface toolkits, then our implementation is able to bind it to all of these toolkits without loss of functionality. An important implication of this guarantee is that existing programs bound to some target toolkit would suffer no loss of functionality when they are ported to our abstract layer and then targeted back at the original toolkit. A more robust system would implement missing functionality.

Our factory-based approach allows toolkit API calls to be redirected to another process rather than a target toolkit. To do such redirection, the redirector must set the target "user-interface toolkit" to an address that represents the remote process. In response, the call sets the abstract factories to *factories* rather than toolkit-specific factories. These factories return, not toolkit proxy objects, but *forwarder objects*, which forward calls to the target process (and store local un-displayed state). This feature is used by the piping replicas to forward calls to the session server. Thus, when the translator program is run by a piping replica, ButtonSelector.createButton() returns an instance of ForwarderButton, which forwards setActionListener() and other calls to the session server.

In summary, we support a three layered approach for abstracting the user-interface toolkit. The top layer defines widget interfaces, factory interfaces, and abstract factory classes, which define the API used by the programmer. The second layer provides, for each target user-interface toolkit and the forwarder "user-interface toolkit," implementations of the factory and widget interfaces defined by the topmost layer. This layer references instances of the actual classes of the target toolkits.

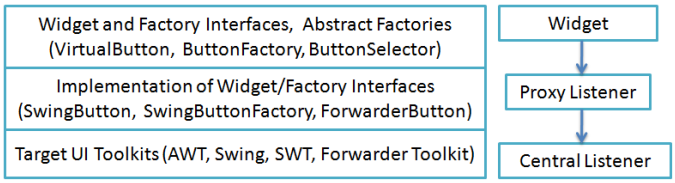

Figure 3 Layers and Modules Processing Calls/Events

Widget Servers and Piping Replica Clients

Systems such as X[17] and NeWS[18] have shown the usefulness and possibility of distributing the window layer. An interactive computer runs a special local process, called a *window server*, which manages the windows on that computer, and accepts connections from multiple, possibly remote, processes, called *window clients*. A window client can ask a window server to create windows on the computer it manages, and is sent user events associated with these windows. This is called the *inverted server* architecture as the local machine hosts a server, and remote machines the clients.

As mentioned above, we distribute the user-interface toolkit layer, which seems like a simple variation of the idea of distributing the window layer, where the main difference is in the types of the objects on the user's computer and the

events that are fired by these objects. Using the window system inverted server terminology, we will use the terms *widget server* and *widget client* to refer to a module that serves and issues, respectively, remote requests to create and manipulate toolkit abstractions. As in the case of a window system, the widget server runs in a process on the user's computer, while a widget client is a remote process.

However, there are some subtle related differences between the motivations for and nature of the distribution of the window and toolkit layers. One difference has to do with the number of user-interface servers on a user's computer. A window system running on a user's computer provides a single process serving all possible window clients that wish to create windows on that computer. Our approach, on the other hand, creates a separate widget server process on a computer for each remote widget client process that creates (synchronized) widgets on that computer.

In a network window system, the single window server is guaranteed to be up before any of its window clients starts, and allows connections to be made to it by clients. Moreover, a client can be on an arbitrary host, and pushes requests to a server. In our context, a piping replica is a client. It is located on a well known host, and is started when a session is created. The server is typically started after this client, when its user joins the session. Furthermore, to respect firewall restrictions, it is not guaranteed to allow inward connections. Therefore, we interpose a central process, the session server, between the server and the piping replicas, which accepts connections from both. The calls of the piping replicas are piped to the session server instead of a target toolkit, which stores them. Later, when a widget server joins a session, it pulls these calls. Any subsequent call made by the piping client is immediately sent to the widget server, again through the session server.

Remote window systems differ on where event processing code is executed. X [17] requires events to be passed back to the window client that made the API calls, while NeWS [18] allows the client to download some event-processing code, written in a special language, into the window system. Our user-interface toolkit model requires us to support listener-based event handling, which means deciding where event processing is done reduces to locating the listeners, discussed next.

Flexible Listener Placement

Like traditional single-user toolkits, we allow a widget to be bound to a local listener. In addition, we allow it to be bound to a central listener in a piping replica, as shown in the call below:

```
translate.setCentralizedComputation(true);
```

which centralizes the listener for the translate button, as it accesses a database located only on the central host.

How the widget-listener binding is done depends not only on the listener location but also on whether a user-interface is created in response to requests issued by a local application replica or a central piping replica. Thus, we have four cases: (1) local replica, local listener; (2) local replica, central listener; (3) piping replica, central listener; (4) piping replica, local listener. The first case is the traditional approach, and can be supported directly by associating the listener with the target toolkit widget. In the other cases, the widgets and listeners are in different address spaces, and in the last two cases, the binding is being made in one process, the widget server, by another process, the piping replica.

Consider case (2). When the local replica makes the call to bind a widget to a local (application-defined) listener, the proxy class that intercepts this call binds the widget to a toolkit-defined *proxy listener*, which forwards received events to the piping replica that hosts the central listener for that widget (Figure 3). Case (3) is similar except that the call to bind the widget to the central listener is invoked by the remote piping replica, through the session manager. This call does not send a listener object to the replica. Instead, it sends a directive to bind the widget to the toolkit-defined proxy listener.

Case (4) is the most difficult, as a listener must be created in the local replica in response to a request from a remote process. One approach, which we support, is to make the requesting process instantiate the listener and send a copy to the widget server. However, the listener may have location-specific references and thus, in Java's terminology, not be serializable. Moreover, this approach does not allow listeners for different widgets to share objects, as copies rather than references of objects are shared. Therefore, our collaboration toolkit allows the application to bind a widget, not only to a listener, but also a factory that returns a listener. The toolkit uses the factory to obtain the real listener. When a piping replica makes such a binding call, it sends a copy of a factory rather than that of a listener. The widget server uses this copy to obtain the listener. The factory can construct the listener locally and, optionally, bind it to parts of listeners constructed by previously sent factories. This approach is illustrated in the following code, where the translate button is associated with a listener factory:

```
translate.addActionListener(new ATranslateListenerFactory(from,
to).createActionListener());
```

Here, "from" and "to" are the text components containing the original and translated text, respectively. A factory can be expected to be serializable as its job is relatively simple. As we see from the call above, a factory or listener may have references to widgets, which are not normally serializable. We define special "serialization" procedures for them that send their global ids, which are de-serialized by mapping them to the corresponding widgets installed in the local process.

Case (4) requires the widget server to access application-defined (listener/factory) classes. If the underlying RPC

system allows classes of objects to be sent along with objects, then this is not an issue. However, the ones we use in our implementation – GWT RPC for communication between web clients and servers and RMI for all other communication - do not currently include this capability. This is not a problem in the GWT case as it allows client-side classes to be specified at the web server, which are automatically compiled and downloaded in the browser. In the RMI case, we use a network loader to obtain such a class, if such an operation does not cause access violations, and centralize the listener otherwise. Creating application-defined factories increases program burden, and dynamically sending them and loading their classes over the network increases the time required to create a user-interface. These problems are consistent with extensible network window systems such as NeWS, which allow only certain kinds of code to be executed in the window system, and incur the cost of creating and sending event handlers.

A widget event can go to a local listener (case 1 and 4) without the need for an intermediary. Yet, even in these cases, we interpose a proxy between the two, which forwards events to the local listener, and also implements widget synchronization, discussed below.

Synchronization

One challenge in automatically synchronizing different widget composition is determining the correspondence between widgets in these compositions. The traditional approach is to assume that identical widget compositions are created by identical replicated code, and use the order in which widgets are created to determine the correspondence. We use a more general approach in which the creation order is used by default, unless the programmer explicitly names the widgets, in which case the name is used to resolve the correspondence. The standard Swing setName() method is also provided by our toolkit to name the widgets:

```
translate.setName("translate")
```

Unlike the Swing toolkit, our synchronization approach requires the name of a widget to be unique.

A toolkit such as ours that supports replication and centralization must support the following synchronization invariants. (a) An event sent to a listener in one process must also be sent to all corresponding replicated listeners in other processes. For instance, in our example, when the user presses the clear button, the event should be sent to all replicated listeners of the widget. (b) A state-changing API call made by a centralized listener must be invoked on all widget servers in the user replicas. Thus, in our example, when the listener in the large UI piping replica sets the text of the "to" widget, the call should be forwarded to all user replicas. (c) Corresponding local toolkit calls made by listeners in different replicas must return the same value. For instance, when the translate button is pressed, the replicated listeners in different replicas must see the same state in the "from" and "to" text widgets.

We support three coupling modes that offer these invariants. In all of these modes, each change to a widget causes the widget and event to be stored in a global buffer. The buffer is sent to all remote (piping and user) replicas whenever a proxy listener (Figure 3) forwards an event to an application-defined listener. In the default/action/incremental coupling mode, no other event/action events/all events cause buffer transmission. We have included these coupling modes to show the range of existing coupling modes [5] that can be supported in our collaboration toolkit. They subsume those offered by current collaboration toolkits, and thus help meet the no superfluous constraint requirement.

On receiving an input event, an interactive replica makes an equivalent call to update the local widget, and also forwards it to local listeners of the event. Thus, when a user edits the "from" text component, the resulting input event, when transmitted to another interactive replica, causes the corresponding local "from" component to be updated with the edited value. The calls made by a centralized listener are sent to all interactive replicas. A replica ignores an event or call referencing a widget that does not exist in its user interface.

Synchronization events and calls are sent to the session manager by the piping replicas through the forwarder user-interface toolkit (Figure 3). These are buffered in the session object in the session manager for replicas that join the session late. While compression techniques [19] can be applied to them, we have not so far provided them in our implementation. Buffered actions are separated into the initial calls for creating the user interface, which are forwarded to generic session joiners, and the subsequent actions, which are forwarded to all replicas. The initial calls received from different piping replicas are kept in separate buffers as these replicas create different user interfaces. All subsequent events and listener calls are kept in a single global buffer. All calls made before the first synchronization input event is received are considered initial calls. A piping replica generates no input events but receives all synchronization events. The synchronization events received by only one of these replicas are stored in the global buffer. Moreover, a call made by a listener in a piping replica in response to a synchronization event is stored in the global buffer only if the (a) the listener is centralized, determined by the API call given earlier, and (b) the replica is responsible for centralizing it, determined by the configuration file, also mentioned earlier.

Figure 2 illustrates and summarizes the logical flow of events and API calls among the interactive and piping replicas. As Cathy joins the session with a generic session joiner, the piping replica for the small user interface sends it calls to create the interface. David's replica does not receive such calls from the piping replica to create the large interface, as he joins the session with a custom replica that makes local calls to create it. As the listener for the translate

button is centralized in the large piping replica, the calls issued by the listener to update the "to" widget are sent to all other (user and piping) replicas. Finally, each user replica sends input to all other replicas.

This logical flow does not reflect the physical flow of messages discussed in earlier sections, which may involve the session manager. It is involved in delivery of (a) all communication to interactive replicas such as David's that have not registered addresses, (b) all calls made by piping replicas, and (c) all latecomer input events.

DISCUSSION

Our abstraction of the target user-interface toolkits and implementation of the sharing mechanisms in terms of this abstraction allows us to meet the unification requirement.

By allowing communication to be centralized in the session server, and some listeners to run on a central machine, we ensure that constraints of our target domains are not violated, thereby meeting the compatibility requirement. Moreover, by allowing processes that can do so to execute listeners locally and communicate directly with each other, we meet the no superfluous constraint requirement. By allowing generic programs to be used to join collaborative sessions and allowing programmers to determine if computation/communication is centralized or replicated allows us to meet the cross fertilization requirements. By allowing programs to also choose the target user-interface toolkit, we meet the controlled retargeting requirement.

By allowing synchronization among heterogeneous compositions of widgets of different target toolkits, we meet the requirements of interoperation and multiple widget-compositions. Modulo a few extra levels of indirection (proxy widgets and listeners), we support the existing functionality and communication and computation architectures of these toolkits. Our preliminary performance measurements show, not surprisingly, that these indirection levels cause un-measurable changes to the performance. Thus, we meet the performance requirement.

The functionality unique to our collaboration toolkit is optional. Defining a configuration file, listener factories and overriding of the communication, centralization, and coupling modes is necessary only if the semantics supported by traditional stand-alone collaboration toolkits, which are replicated, is unacceptable. Thus, arguably, we meet the proportional effort requirement.

Dynamic binding of an application program to a concrete user-interface – also called *plasticity* [20] - is the goal of several collaborative and non collaborative systems. As mentioned earlier, a particularly relevant example of a procedural system supporting this idea is WAHID [13]. However, it is not a collaboration toolkit, and thus does not meet any of our other requirements. There are numerous examples of previous collaboration toolkits such as [3, 8, 12]. However none of these supports heterogeneous domains or user-interface toolkits.

Several other forms of collaboration tools meet one or more of our requirements. Like us, Cooperative Teresa [14] allows synchronization of heterogeneous widget compositions. However, it is targeted only at the web domain. More important, it shares the view rather than the user-interface toolkit layer. As mentioned before, a higher-level tool such as this one supports a non standard and restrictive programming paradigm, and does not allow certain useful coupling modes that can be implemented only in a collaboration toolkit. Some model-based [4] or layer-independent[21] tools support multiple communication and/or computation architectures. One difference between these systems and ours is that the former assume all computation can be safely replicated, and support multiple architectures only for performance.

As mentioned before, sharing of each layer provides a unique set of advantages and disadvantages. For instance, as also mentioned before, sharing of a layer higher than the user-interface toolkit allows more divergence in the synchronized widget compositions, and also, in view-sharing systems such as Cooperative Teresa, allows automatic generation of these compositions. Thus, our system has several disadvantages that collaboration toolkits suffer in general. As mentioned before, the unique advantages and disadvantages of sharing each layer are discussed in depth in [2].

In summary, no other system meets all of the requirements our toolkit was designed to meet. More strongly, none of them supports: a generic session-joining program in domains other than the web, programmer-controlled retargeting of an entire user-interface toolkit, widget-grained control over whether communication and computation is centralized, and no superfluous constraints.

Novel Mechanism	Supported Requirements
Factory-based retargeting	Unification, controlled targeting
Widget server	Generic session joining tool, listener centralization, latecomer
Synchronization modes	No extraneous constraints
Flexible listener placement	Widget-grained computation centralization/replication, no extraneous constraints
Piping replicas	Computation centralization, Generic session joining tool, Hetero. widget compositions
Process architecture	All requirements

Table 1 Novel Mechanisms vs. Requirements

The novelty of a system must be judged not only by the requirements it meets but also the mechanisms it offers. Table 1 identifies the major novel mechanisms and the collaboration requirements they support. Some of these mechanisms have applications beyond collaboration. Factory-based retargeting can be used to create plastic

single-user interfaces, and piping replicas could be used for monitoring user-interface activity.

Perhaps more important than the novel aspects of our collaboration toolkit are the requirements themselves, which, for the first time, define what it means to support a collaboration toolkit that can be targeted at multiple domains and associated user-interface toolkits. In particular, they show that web features can be an asset rather than a liability in collaboration toolkits.

Experience with our system is necessary to refine our requirements and identify alternative mechanisms to meet them. It would be useful to identify: (a) other uses of some our mechanisms, (b) support for multi-language user-interface toolkits, (c) how collaboration functions other than widget synchronization, such as concurrency control, operation transformation, awareness, and collaborative undo can be added to our mechanisms, (d) practical ways to attach our adapters to existing code, without any modification to it, and (e) adapt our requirements and mechanisms to higher-level collaboration tools exhibiting heterogeneity.

ACKNOWLEDGEMENTS

This research was funded in part by NSF grants IIS 0712794 and IIS-0810861.

REFERENCES

1. Heering, J. and P. Klint, *Towards Monolingual Programming Environments.* ACM Transactions on Programming Languages and Systems, April 1985. **7**(2).
2. Dewan, P., *Architectures for Collaborative Applications.* Trends in Software: Computer Supported Co-operative Work, 1998. **7**: p. 165-194.
3. Roseman, M. and S. Greenberg, *Building Real-Time Groupware with GroupKit, A Groupware Toolkit.* ACM Transactions on Computer-Human Interaction, 1996. **3**(1).
4. Wolfe, C., T.C.N. Graham, W.G. Phillips, and B. Roy. *Fiia: User-Centered Development of Adaptive Groupware Systems*. in *ACM EICS*. 2009.
5. Dewan, P. and R. Choudhary, *Coupling the User Interfaces of a Multiuser Program.* ACM Transactions on Computer Human Interaction, March 1995. **2**(1).
6. Sun, C., S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai, *Transparent adaptation of single-user applications for multi-user real-time collaboration.* ACM Transactions on Computer Human Interaction, 2006. **13**(4).
7. Cheng, L.-T., S. Hupfer, S. Ross, and J. Patterson. *Jazzing up Eclipse with collaborative tools*. in *Proceedings of the OOPSLA workshop on eclipse technology eXchange*. 2003.
8. Greenberg, S. and M. Roseman. *GroupWeb: A WWW Browser as Real Time Groupware*. in *Proc. CHI*. 1996: ACM.
9. Gutwin, C., M. Lippold, and N. Graham. *Real-Time Groupware in the Browser: Testing the Performance of Web-Based Networking*. in *Proc. CSCW*. 2011: ACM.
10. Shao, B., D. Li, T. Lu, and N. Gu. *An operational transformation based synchronization protocol for Web 2.0 applications*. in *Proc. CSCW*. 2011.
11. Nichols, D., P. Curtis, M. Dixon, and J. Lamping. *High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System*. in *UIST*. 1995.
12. Begole, J., Rosson, M., and Shaffer, C. *Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems*. in ACM Transactions on Computer Human Interaction, 1999. **6**(2).
13. Jabarin, B. and N. Graham. *Architectures for Widget-Level Plasticity*. in *Proc. DSV-IS*. 2003.
14. Paternò, F. and I. Santos. *Designing and Developing Multi-User, Multi-Device Web Interfaces*. in *Proc. CADU*. 2006: Springer.
15. Bishop, J. *Multi-platform User Interface Construction – a Challenge for Software Engineering-in-the-Small*. in *Proc. ICSE*. 2006.
16. Gamma, E., R. Helm, R. Johnson, and J. Vlissedes, *Design Patterns, Elements of Object-Oriented Software*, Reading, MA.: Addison Wesley, 1995.
17. Scheifler, R.W. and J. Gettys, *The X Window System.* ACM Transactions on Graphics, Aug. 1983. **16**(8).
18. Gosling, J. *"SunDew?: a distributed and extensible window system*. in *Proceedings of an Alvey Workshop on Methodology of window management,*. 1986.
19. Chung, G., P. Dewan, and S. Rajaram. *Generic and composable latecomer accommodation service for centralized shared systems* in *Proc. IFIP Conference on Engineering for Human Computer Interaction, Chatty and Dewan, editors*. 1998: Kluwer Academic Publishers.
20. Calvary, G., J. Coutaz, David Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt, *A unifying reference framework for multi-target user interfaces.* Interacting with Computers, 2003. **15**(3).
21. Chung, G. and P. Dewan. *Towards dynamic collaboration architectures*. in *Proc. ACM CSCW*. 2004.