Published in *The X Resource*, #13, Winter, 1995. O'Reilly Publishers, Sebastopol, CA.

A Remote Access Protocol for the X Window System

W. Keith Edwards, Susan H. Liebeskind, Elizabeth D. Mynatt and William D. Walker

Abstract

This paper presents a protocol which communicates information about graphical interfaces between X applications. This protocol, called Remote Access Protocol (RAP), can be used to script or test graphical applications, or, as in our case, translate a graphical application to a non-visual presentation. We present details on how applications begin communicating via RAP and describe the messages in the protocol itself.

Introduction

This paper describes an inter-application communication protocol called Remote Access Protocol, or RAP. RAP is designed to allow a process (called an *agent*) to be notified of changes in the state of another application's interface, and to even cause changes in the state of other applications.

The basic framework needed to support RAP transparently has been included in Release 6 of the X Window System, in the form of a new HooksObject within the Xt toolkit. The RAP protocol itself, along with its rendezvous mechanisms, are being proposed as a new standard to be supported in Release 7.

RAP can facilitate the construction of new classes of applications. One category of systems that RAP supports (and, in fact, the reason that design work on RAP was undertaken in the first place) is interface translators. Via RAP, an agent can receive notifications of changes in an application's interface and can present that interface in a new (and possibly even non-visual) modality. For example, a RAP agent could allow access to existing X Window System applications via a car phone, or could allow blind users to operate graphical applications by translating those applications to an auditory presentation.

Other applications include batch scripting of graphical applications, automated testing, and interactive resource editing, a la Editres. While our focus has been on providing a software infrastructure for interface transformation, we have tried to ensure that RAP is usable for other applications as well.

In this paper, we first discuss a motivational example for RAP, namely Mercator, Georgia Tech's screen reader for X Window applications. We follow with a discussion of the infrastructure that will support the RAP protocol, both existing infrastructure and a proposal for additional infrastructure. Next, we discuss our goals for the RAP protocol, followed by a section covering the messages that form the protocol. Lastly, we discuss the status and future directions for the RAP protocol implementation.

Motivation

As a motivating example of RAP, we shall consider the case of translating an existing graphical interface into a non-visual presentation. This transformation has been the goal of the Mercator Project, a research effort at the Georgia Institute of Technology. The Mercator effort has been concerned with the construction of a system which can permit blind or severely visually-impaired computer users to access graphical applications. The system translates graphical output into nonvisual (primarily speech and non-speech audio), and provides new input modalities for controlling applications (for example, the system translates keyboard input into the mouse input “expected” by most applications). Such a system is also useful for providing access to graphical applications in situations where only a limited visual display, or even no visual display at all is present (car phones and PDAs come to mind).

When we began the Mercator Project, we based our system on a completely external approach for capturing information about changes in the state of an X application. We used a pseudoserver process to both trap and synthesize X protocol to and from applications. The pseudoserver, in conjunction with Editres, allowed us some degree of access to the interface of X applications. However it became clear that programming the system to deal flexibly with highly dynamic graphical interfaces was problematic with the pseudoserver approach [1].

For our second version of the system, we began to investigate a set of modifications to Xt itself which would provide more complete and high-level information about application interfaces. The rationale for this approach was that by instrumenting Xt to notify us about changes in an application’s widget hierarchy and widget resource values, our agent process would be able to construct a more robust model of the application’s graphical interface. Further, this approach would be independent of the various widget sets layered atop Xt (that is, we would not need to provide additional code changes every time a new widget class was written) [2].

RAP Design Goals

Before we begin our discussion of the Remote Access Protocol itself, we must present some of the design goals we felt were important.

Transparency

Transparency means that an application with the requisite set of modifications can participate in the protocol without programmer intervention. That is, no special action is required on the part of the application developer to take part in the services provided by RAP.

To implement transparency, the infrastructure provided by Xt and Xlib must be extended so that the entire process of initiating communication, responding to requests, and generating notifications occurs within Xt and Xlib themselves. However this feature does not mean that "RAP-aware" clients cannot exert control over their participation in the protocol, if they are specifically written to do so. Further, clients can specifically choose to not participate in the protocol, for security or other reasons.

Arbitrary Rendezvous Order

It is important that applications and agents be able to connect to each other in arbitrary order. For example, a blind user may begin an X session by starting the interface translation agent, and then running any desired applications. In the case of an interactive resource editor, the agent may be started after the application is already running.

For maximum flexibility, and to support a range of different agents, arbitrary start-up ordering is a requirement.

Support for Multiple Agents

The protocol and initiation procedures should not prevent multiple agents running at the same time. It is easy to envision a situation in which a given user may be running a testing application, a resource editor, and an interface translator at the same time. The protocol infrastructure must support connection of an arbitrary number of agents to a set of applications, whether those agents are speaking RAP or some other protocol.

Support for Multiple Protocols

While our work has focused primarily on the RAP protocol for communication of interface changes, it is possible that the infrastructure required for RAP may be used for other protocols in the future. Thus, we should provide support for arbitrary multiple protocols to share the connection set-up and hook infrastructure used by RAP.

Non-Xt Specific

Although our current implementation is for the Xt toolkit, and our work has resulted in a number of changes to Xt itself, there is no reason that a RAP implementation could not be provided for other Xt (or even non-Xt) toolkits. For this reason, it is important to not include any constructs in the protocol which are specific to Xt and might not be implementable in other toolkit paradigms.

Information Filtering

Even though RAP is a general tool for communicating changes in application interfaces to an agent process, different agents may require different information even though they all participate in RAP. For example, an interface translation system requires access to all interface change information, while an interactive resource editor only needs to be informed of changes in resource values. Thus

we must provide a mechanism to limit the types of information which are sent from an application to an agent via RAP.

Light-weight

The protocol should be lightweight, and not impact the performance of the clients by its presence. It is possible that some clients will choose not to participate in the protocol; such clients should not have to pay the price for unused functionality.

RAP Infrastructure

In order to translate graphical interfaces to non-visual equivalents, a certain amount of infrastructure must be supplied within the X Window System. Some of the necessary translation support was added as of Release 6. This support, discussed in the *Existing Infrastructure* section, will permit translation to take place, assuming that the protocol connection has been established and initialized. But support to establish and initialize the protocol is still lacking in the current release. In the *Additional Required Infrastructure* section, we outline our proposal to resolve this deficiency. With this extra functionality, we will have all the necessary components to support the RAP protocol.

Existing Infrastructure

Over the course of this second phase of the project, a set of code changes were proposed to the X Consortium and accepted as a Consortium standard with Release 6. These code changes provide a new, private, implementation-dependent widget to Xt called the HooksObject. The Hooks object is associated with an application's display connection, and maintains a set of callback lists (see Table 1, "HooksObject Callback Lists,"). These callback lists hold callback routines which are fired whenever certain changes occur in the state of the application's interface. Xt has been instrumented to call into the appropriate HooksObject callback list. For example, whenever a new widget is created, Xt will invoke any callbacks present in the WidgetCreation callback list in the HooksObject. Some additional APIs were added to Xt to allow applications to retrieve the HooksObject associated with a display connection [3] and the shells associated with the display.

Callback List	**Description**
CreateHook	Called whenever a new widget instance is created.
ChangeHook	Called whenever a widget resource value is updated.
ConfigHook	Called whenever a widget's configuration (size or position) change.
GeometryHook	Called whenever a widget's geometry is updated.
DestroyHook	Called whenever a widget is destroyed (when a widget's phase 1 destruction is complete).

TABLE 1. HooksObject Callback Lists

The changes to the R6 Xt library do not specify *what* code will be installed in the various callback lists; they merely provide the infrastructure to have Xt call out to some installed set of procedures whenever interface changes occur.

Along with the R6 HooksObject in Xt, a new client-side extension hook was added to Xlib. An extension procedure may be installed via XESetBeforeFlush() which is called whenever the Xlib request buffer is flushed [4]. The motivation for this change was that an interface transformation system (or other applications interested in low-level protocol information) could install a procedure via XESetBeforeFlush to pass the low-level X protocol information which was previously available only through pseudoserver-based approaches.

Like the changes to Xt, the R6 Xlib does not specify any code to be installed in the BeforeFlush hook; it merely provides infrastructure which applications can use to install code.

These changes to the Release 6 Xlib and Xt libraries were based on our experiences with two versions of the Mercator system. The types of information which can be provided by these modifications are necessary to enable translation of graphical interfaces to non-visual modalities at runtime.

With the framework in hand to enable translations, we turn our attention to designing a protocol that can actually pass the information required for translation. The Inter-Client Exchange Protocol (ICE), new to X11R6, facilitates the process of developing inter-client communication protocols, such as RAP. ICE provides functionality common to many protocols (opening connections, validation of requests, closing connections to name but a few). This set of services allows protocol developers to concentrate on designing the protocol-specific messages and message handlers, while leveraging the generic protocol framework in ICE. Additional details on ICE may be found in [5] and [6].

Additional Required Infrastructure

Unfortunately, although we have the necessary framework in Release 6 to support the translation of graphical interfaces, we do not have the sufficient framework to support this translation. We lack a way to jump-start the process, initiating the RAP protocol as soon as the prospective partners can participate in the protocol.

This problem is not specific to the RAP protocol -- all ICE-based protocols face the issue of initiating the communication process. For many custom protocols, this is not a large problem. The clients on either end of the wire are explicitly aware of the protocol, and at the proper point in their execution, can take steps to setup the communication channel, via the appropriate ICE calls. But for the RAP protocol to work with off-the-shelf X clients, written without foreknowledge of the RAP protocol, transparent initiation is a significant problem.

The solution that we will propose for inclusion in Release 7 is the ICE Rendezvous Mechanism [7]. The rendezvous mechanism is designed to establish an ICE connection between an agent and a client in a manner which will not require explicit awareness of the RAP protocol. Briefly, clients wishing to speak ICE-based protocols follow this set of steps to fire up the ICE connection to support the protocol:

1. The client registers interest in the common protocol with the ICE library.
2. The client either actively tries to make an ICE connection to its partner(s) if this client is the Protocol Originator, or passively waits for its partner(s) to make the connection, if this client is the Protocol Acceptor. Authentication at this step is optional.

3. Once the ICE connection is made, the Protocol Setup request and Protocol Reply response messages are exchanged between originator and acceptor. Following the successful, possibly authenticated, exchange of these messages, the custom protocol's messages may be passed back and forth.

The information that must be made known to both sides of the channel is

- the common protocol (in our case, RAP)
- the network connection point (network id), dynamically assigned by the ICE library

To transfer this information, we are proposing the use of the Client Message mechanism to pass the protocol and network ID information, used in conjunction with a new data object, a table of protocol initialization routines. Calls will be provided to install, remove, and retrieve the protocol routines in the table.

The ClientMessage event handler, specific to each protocol, will be responsible for installing the appropriate protocol initialization routine. The protocol initializer routine, supplied by the protocol developer, will handle any required bookkeeping to fire up the protocol. This protocol initializer routine is not necessarily tied to setting up an ICE connection (although the RAP protocol will use its routine for this purpose.) Any client capable of being monitored by an agent must have its routine installed in its address space.

But one question remains: how does the RAP ClientMessage event handler get installed in the first place, to make all this happen without requiring clients to be RAP aware. The answer lies in the VendorShell source file, specific to each toolkit. Just as the Editres [8] Client Message handler is installed in Athena widget set's VendorShell implementation, so that all clients can be queried transparently by the editres program, so would the RAP client message handler be installed in the VendorShell implementation for all toolkits (Motif and Athena). In this way, any Motif or Athena-based client will be RAP aware, simply by being linked against the toolkit libraries.

The Remote Access Protocol

This section of the paper describes the format of the Remote Access Protocol itself. We describe the semantics of each message in the protocol (that is, what each message *means*), the circumstances under which each message is sent, and the format of the messages. This list is not meant to be definitive or final; it represents the current state of the protocol as used by Mercator [9][10].

Whereever the term "widget ID" is used in the description of a particular message, we are referring to a 32-bit unique identifier for a toolkit object in the application. While in the Xt implementation of RAP these 32-bit values will map directly onto widget IDs, other toolkits may use them to identify their particular constructs.

Further, some messages may not be implemented for certain toolkits. See the DoActionRequest for example.

There are three basic types of messages in RAP: requests, replies, and notifications.

Requests are RAP messages which travel from an external agent process to an application. They are used to ask the application for information about its interface, or to control some aspect of the application.

Replies are responses to specific requests for information sent from the agent. Replies travel from the application to the agent.

Notifications are asynchronous messages which are used to notify an interested agent in a change in the status of the application's interface. Via a set of notification-control requests, the generation of notifications within the application can be selectively filtered.

GetResourcesRequest

The GetResourcesRequest message passes a list of widget IDs to the application; the application responds with a list of the names and types of the resources in the specified widgets (see GetResourcesReply).

QueryTreeRequest

The QueryTreeRequest asks the application to transmit its entire widget hierarchy back to the calling agent via a QueryTreeReply.

GetValuesRequest

GetValuesRequest is used to ask for the values of a set of resources on a set of widgets. The message passes a list of structures; each structure includes a widget ID and a list of resources to retrieve the values of for the specified widget. The application generates a GetValuesReply with the requested information.

SetValuesRequest

The SetValuesRequest message is used by an external agent to change the value of a particular set of resources in a particular set of widgets. The message encodes a list of structures; each structure contains a widget ID, and a list of resource-value-type tuples specifying all of the resources to change in the particular widget, the new values, and the type the value is expressed in.

AddNotifyRequest

The AddNotifyRequest is used to control filtering of notifications sent from the application. The message passes a list of names of callback lists in the HooksObject. Any callback lists specified in the request will be "turned on," meaning that they will generate notifications to the external agent.

RemoveNotifyRequest

RemoveNotifyRequest is used to disable particular notifications by "turning off" callbacks in the HooksObject. Like AddNotifyRequest, it takes a list of names of callback lists to disable.

ObjectToWindowRequest

The ObjectToWindowRequest is used to resolve a widget ID into the primary window used by that widget. The message takes a list of widget IDs to resolve, and generates an ObjectToWindowReply message from the application.

WindowToObjectRequest

WindowToObjectRequest it used to resolve a window ID into the primary Xt widget associated with that window. The message sends a list of window IDs to resolve, and generates a WindowToObjectReply from the application.

LocateObjectRequest

The LocateObjectRequest message is used to find the on-screen locations of a specified list of objects. The message generates a LocateObjectReply return from the application.

GetActionsRequest

The GetActions Request is used to retrieve a list of actions associated with a group of widgets. The message contains a list of widgets.

DoActionRequest

This request is used to invoke an action. It contains a widget and an action name which together specify the action to run.

This message will most probably not be provided in the Xt implementation of RAP, because there are situations in which Xt-based applications may become unstable if actions were invoked directly.

SelectEventRequest

The SelectEventRequest message is used to control which event types are sent from the application to the agent. Whenever connection between an agent and a client application is initiated, a procedure is installed in the WireToEvent client-side extension slot which can transmit X Events to an agent. This message can be used to filter what events are actually sent.

SelectRequestRequest

This message is used to filter the transmission of X Protocol requests from the application to the agent. Requests are “caught” by a routine installed in the BeforeFlush client-side extension and are sent to the agent if their types have been selected by SelectRequestRequest.

CloseConnectionRequest

This message is generated by an agent whenever it wishes to terminate its communication with a client application. The client should take any appropriate clean-up action and then disconnect.

GetResourcesReply

This message is generated in response to a GetResourcesRequest; it contains a list of structures. Each structure contains the ID of a requested widget, and a list of the resource names, classes, and types for that widget.

QueryTreeReply

QueryTreeReply is generated in response to a QueryTreeRequest and returns the widget tree of the application. The information is formatted as a list of tuples, with each tuple containing the widget ID, name class, window ID, and parent widget ID.

GetValuesReply

This message returns a list of resource values for specified widgets to the caller. The format of the message is a list of structures, with each structure containing the ID of a requested widget, and a list of the requested resource names and values for that widget.

ObjectToWindowReply

The ObjectToWindowReply message returns a list of the windows corresponding to the objects requested via ObjectToWindowRequest.

WindowToObjectReply

WindowToObjectReply returns a list of the objects associated with the windows passed via WindowToObjectRequest.

LocateObjectReply

The LocateObjectReply message returns an X, Y coordinate pair for each of the objects requested via LocateObjectRequest.

GetActionsReply

The GetActionsReply is generated in response to a GetActionsRequest and contains a list of action names for each widget specified in the GetActionsRequest message.

CreateNotify

The CreateNotify message is generated by code installed in the CreateHook callback list in the HooksObject. CreateNotify is used to inform an agent whenever a new widget has been created. The messages passes the widget ID, name, class, window ID, and parent widget ID to the agent.

ChangeNotify

The ChangeNotify message is generated via the ChangeHook whenever a resource value is updated. The message sends the widget ID, resource name, and new resource value to the agent.

ConfigNotify

A ConfigNotify message is sent to any interested agents whenever the ConfigHook callback is executed. This message contains information on widget configuration changes: the widget ID, a geometry mask specifying the actual changes which have taken place, and an XWindowChanges structure describing the changes. The message is sent after the changes have taken place.

GeometryNotify

GeometryNotify is generated whenever the GeometryHook callback is executed. This occurs when Xt application makes a geometry request. The message contains the resulting widget geometry, expressed as an XtWidgetGeometry structure. The message is sent after the geometry negotiation has completed.

DestroyNotify

DestroyNotify is used to inform an agent that a widget has been destroyed. The message contains the ID of the newly-destroyed widget, and is generated via the DestroyHook callback list.

RequestNotify

RequestNotify passes X protocol requests to the agent. The actual requests which are sent may be selected by the SelectRequestRequest message. The message contains one or more wire-format X protocol requests; it is generated by code installed in the BeforeFlush client-side extension.

EventNotify

EventNotify passes X events to the agent. The event types which are sent may be selected via the SelectEventRequest message. The message contains one or more wire-format X events, and is generated by code installed in the WireToEvent client-side extension.

Status and future directions

Our current implementation of the Mercator system is built using a protocol which predates (but is essentially similar) to RAP. This protocol runs over standard TCP/IP sockets. Implementation of RAP itself, as well as implementation of the changes to Mercator to use RAP, are ongoing.

Will Walker is serving as the RAP architect, and has begun implementation work on the ICE rendezvous mechanism, and the RAP protocol messages. The public forum on which RAP-related issues are discussed is the x-agent public mailing list, x-agent@x.org. This mailing list is sponsored by the X Consortium for the purposes of developing protocols like RAP for external agent software. Anyone who is interested in contributing to the effort may join the list by sending a message to

```
x-agent-request@x.org
```

The body of this message should contain the single word “subscribe”. The subject line of the message will be ignored.

At last year’s X Technical conference, Philippe Kaplan and Anselm Baird-Smith of Bull France announced their efforts in a next generation Editres protocol, called K-Edit. K-Edit is “an attempt to provide a simple multipurpose protocol, suitable (in particular) to export an application user interface state” [11]. There is enough overlap between the goals of the K-Edit protocol and the RAP protocol that there has been continuing discussion over the last year, with the possibility of combining the two efforts into a common protocol.

The ICE rendezvous mechanism and ClientMessage handler outlined in the Additional Infrastructure section will be proposed for inclusion in Release 7. However, those proposed

changes can be added to R6 clients today. For dynamically-linked applications, all that must be done is to rebuild the Intrinsics library from source modified to support the rendezvous and ClientMessage. The next invocation of the dynamically linked clients will be RAP aware, as the library they utilize dynamically will have been made RAP aware. Statically linked clients will need to be relinked against a newly compiled static Intrinsics library, but in keeping with the goals for the RAP protocol, the clients themselves will not need to be modified at the source code level in any way.

Since the format of the RAP protocol is still evolving, we expect changes in the protocol between now and the Release 7 cut-off date. One area where we need to focus is resource type negotiation. Certain types used by resources within an application may not be known to a particular external agent; yet the application may have the ability to convert these types to a format which the agent can understand. Conversely, the agent itself may have the ability to convert certain types internally, even if the agent does not have the required converters linked into it. Thus a process where resource types are negotiated between agents and applications may be useful.

Summary

We have presented a protocol for communicating changes in interface state between X applications and agent processes. This protocol is powerful in that it can support a variety of useful agents including interactive resource editing, scripting, testing, and interface translation.

The protocol builds on a set of hooks, some of which are already present in Release 6 of the X Window System, and some of which are still pending.

We acknowledge the contributions of many people to this project: Tom Rodriguez (formerly of Georgia Tech, now at Sun Microsystems, who designed the protocol on which RAP is based; Bob Scheifler, Donna Converse, Ralph Swick, Daniel Dardailler, and Kaleb Keithley (of the X Consortium) who provided much-needed design guidance; and the members of the Disability Action Committee on X (DACX). Thanks also to Sun Microsystems and the NASA Marshall Space Flight Center for their sponsorship of Mercator.

Author Information

Keith Edwards, Susan Liebeskind, and Beth Mynatt work at the Graphics, Visualization, & Usability Center at Georgia Tech. Their email addresses are keith@cc.gatech.edu, shl@cc.gatech.edu, and beth@cc.gatech.edu; they can be reached via telephone at (404) 894-3658. Will Walker is a Software Engineer at Digital Equipment Corporation. His email address is wwalker@zk3.dec.com.

References

[1] Elizabeth Mynatt and W. Keith Edwards. *Mapping GUIs to Auditory Interfaces*. In Proceedings of the Fifth ACM Symposium on User Interface Software and Technology (UIST), Monterey, CA, November, 1992.

[2] Elizabeth Mynatt and W. Keith Edwards. A*n Architecture for Transforming Graphical Interfaces*. In Proceedings of the Seventh ACM Symposium on User Interface Software and Technology (UIST), Marina Del Rey, CA, November, 1994.

[3] Joel McCormack, Paul Asente, Ralph Swick. *X Toolkit Intrinsics C Language Interface.* X Consortium Standard. Version 11, Release 6.

[4] James Gettys and Robert Scheifler. *Xlib C Language Interface*. X Consortium Standard. Version 11, Release 6.

[5] Robert Scheifler and Jordan Brown. *Inter-Client Exchange (ICE) Protocol, Version 1.0*. X Consortium Standard. Version 11, Release 6. 1994.

[6] Ralph Mor. *Inter-Client Exchange (ICE) Library, Version 1.0.* X Consortium Standard. Version 11, Release 6. 1994.

[7] William D. Walker. *An ICE Rendezvous Mechanism for X Window System Clients*. Unpublished report, Disability Action Committee for X, 1994.

[8] Chris D. Peterson. *Editres: A Graphical Resource Editor for X Toolkit Applications.* In Proceedings of the Fifth Annual X Technical Conference, Boston, MA, January, 1991.

[9] W. Keith Edwards and Tom Rodriguez. *Runtime Translation of X Interfaces to Support Visually-Impaired Users.* In Proceedings of the 7th Annual X Technical Conference, Boston, MA, January 18-20, 1993.

[10] William D. Walker. *Remote Access Protocol (RAP), Version 0.2.* DACX Work In Progress. November 11, 1993.

[11] Philippe Kaplan and Anselm Baird-Smith. *The K-edit System.* Unpublished report available via http://zenon.inria.fr:8003/koala/k-edit.html.

Published in *The X Resource*, #13, Winter, 1995. O'Reilly Publishers, Sebastopol, CA.

A Remote Access Protocol for the X Window System

W. Keith Edwards, Susan H. Liebeskind, Elizabeth D. Mynatt and William D. Walker

Abstract

This paper presents a protocol which communicates information about graphical interfaces between X applications. This protocol, called Remote Access Protocol (RAP), can be used to script or test graphical applications, or, as in our case, translate a graphical application to a non-visual presentation. We present details on how applications begin communicating via RAP and describe the messages in the protocol itself.

Introduction

This paper describes an inter-application communication protocol called Remote Access Protocol, or RAP. RAP is designed to allow a process (called an *agent*) to be notified of changes in the state of another application's interface, and to even cause changes in the state of other applications.

The basic framework needed to support RAP transparently has been included in Release 6 of the X Window System, in the form of a new HooksObject within the Xt toolkit. The RAP protocol itself, along with its rendezvous mechanisms, are being proposed as a new standard to be supported in Release 7.

RAP can facilitate the construction of new classes of applications. One category of systems that RAP supports (and, in fact, the reason that design work on RAP was undertaken in the first place) is interface translators. Via RAP, an agent can receive notifications of changes in an application's interface and can present that interface in a new (and possibly even non-visual) modality. For example, a RAP agent could allow access to existing X Window System applications via a car phone, or could allow blind users to operate graphical applications by translating those applications to an auditory presentation.

Other applications include batch scripting of graphical applications, automated testing, and interactive resource editing, a la Editres. While our focus has been on providing a software infrastructure for interface transformation, we have tried to ensure that RAP is usable for other applications as well.

In this paper, we first discuss a motivational example for RAP, namely Mercator, Georgia Tech's screen reader for X Window applications. We follow with a discussion of the infrastructure that will support the RAP protocol, both existing infrastructure and a proposal for additional infrastructure. Next, we discuss our goals for the RAP protocol, followed by a section covering the messages that form the protocol. Lastly, we discuss the status and future directions for the RAP protocol implementation.

Motivation

As a motivating example of RAP, we shall consider the case of translating an existing graphical interface into a non-visual presentation. This transformation has been the goal of the Mercator Project, a research effort at the Georgia Institute of Technology. The Mercator effort has been concerned with the construction of a system which can permit blind or severely visually-impaired computer users to access graphical applications. The system translates graphical output into nonvisual (primarily speech and non-speech audio), and provides new input modalities for controlling applications (for example, the system translates keyboard input into the mouse input “expected” by most applications). Such a system is also useful for providing access to graphical applications in situations where only a limited visual display, or even no visual display at all is present (car phones and PDAs come to mind).

When we began the Mercator Project, we based our system on a completely external approach for capturing information about changes in the state of an X application. We used a pseudoserver process to both trap and synthesize X protocol to and from applications. The pseudoserver, in conjunction with Editres, allowed us some degree of access to the interface of X applications. However it became clear that programming the system to deal flexibly with highly dynamic graphical interfaces was problematic with the pseudoserver approach [1].

For our second version of the system, we began to investigate a set of modifications to Xt itself which would provide more complete and high-level information about application interfaces. The rationale for this approach was that by instrumenting Xt to notify us about changes in an application’s widget hierarchy and widget resource values, our agent process would be able to construct a more robust model of the application’s graphical interface. Further, this approach would be independent of the various widget sets layered atop Xt (that is, we would not need to provide additional code changes every time a new widget class was written) [2].

RAP Design Goals

Before we begin our discussion of the Remote Access Protocol itself, we must present some of the design goals we felt were important.

Transparency

Transparency means that an application with the requisite set of modifications can participate in the protocol without programmer intervention. That is, no special action is required on the part of the application developer to take part in the services provided by RAP.

To implement transparency, the infrastructure provided by Xt and Xlib must be extended so that the entire process of initiating communication, responding to requests, and generating notifications occurs within Xt and Xlib themselves. However this feature does not mean that "RAP-aware" clients cannot exert control over their participation in the protocol, if they are specifically written to do so. Further, clients can specifically choose to not participate in the protocol, for security or other reasons.

Arbitrary Rendezvous Order

It is important that applications and agents be able to connect to each other in arbitrary order. For example, a blind user may begin an X session by starting the interface translation agent, and then running any desired applications. In the case of an interactive resource editor, the agent may be started after the application is already running.

For maximum flexibility, and to support a range of different agents, arbitrary start-up ordering is a requirement.

Support for Multiple Agents

The protocol and initiation procedures should not prevent multiple agents running at the same time. It is easy to envision a situation in which a given user may be running a testing application, a resource editor, and an interface translator at the same time. The protocol infrastructure must support connection of an arbitrary number of agents to a set of applications, whether those agents are speaking RAP or some other protocol.

Support for Multiple Protocols

While our work has focused primarily on the RAP protocol for communication of interface changes, it is possible that the infrastructure required for RAP may be used for other protocols in the future. Thus, we should provide support for arbitrary multiple protocols to share the connection set-up and hook infrastructure used by RAP.

Non-Xt Specific

Although our current implementation is for the Xt toolkit, and our work has resulted in a number of changes to Xt itself, there is no reason that a RAP implementation could not be provided for other Xt (or even non-Xt) toolkits. For this reason, it is important to not include any constructs in the protocol which are specific to Xt and might not be implementable in other toolkit paradigms.

Information Filtering

Even though RAP is a general tool for communicating changes in application interfaces to an agent process, different agents may require different information even though they all participate in RAP. For example, an interface translation system requires access to all interface change information, while an interactive resource editor only needs to be informed of changes in resource values. Thus

we must provide a mechanism to limit the types of information which are sent from an application to an agent via RAP.

Light-weight

The protocol should be lightweight, and not impact the performance of the clients by its presence. It is possible that some clients will choose not to participate in the protocol; such clients should not have to pay the price for unused functionality.

RAP Infrastructure

In order to translate graphical interfaces to non-visual equivalents, a certain amount of infrastructure must be supplied within the X Window System. Some of the necessary translation support was added as of Release 6. This support, discussed in the *Existing Infrastructure* section, will permit translation to take place, assuming that the protocol connection has been established and initialized. But support to establish and initialize the protocol is still lacking in the current release. In the *Additional Required Infrastructure* section, we outline our proposal to resolve this deficiency. With this extra functionality, we will have all the necessary components to support the RAP protocol.

Existing Infrastructure

Over the course of this second phase of the project, a set of code changes were proposed to the X Consortium and accepted as a Consortium standard with Release 6. These code changes provide a new, private, implementation-dependent widget to Xt called the HooksObject. The Hooks object is associated with an application's display connection, and maintains a set of callback lists (see Table 1, "HooksObject Callback Lists,"). These callback lists hold callback routines which are fired whenever certain changes occur in the state of the application's interface. Xt has been instrumented to call into the appropriate HooksObject callback list. For example, whenever a new widget is created, Xt will invoke any callbacks present in the WidgetCreation callback list in the HooksObject. Some additional APIs were added to Xt to allow applications to retrieve the HooksObject associated with a display connection [3] and the shells associated with the display.

Callback List	**Description**
CreateHook	Called whenever a new widget instance is created.
ChangeHook	Called whenever a widget resource value is updated.
ConfigHook	Called whenever a widget's configuration (size or position) change.
GeometryHook	Called whenever a widget's geometry is updated.
DestroyHook	Called whenever a widget is destroyed (when a widget's phase 1 destruction is complete).

TABLE 1. HooksObject Callback Lists

The changes to the R6 Xt library do not specify *what* code will be installed in the various callback lists; they merely provide the infrastructure to have Xt call out to some installed set of procedures whenever interface changes occur.

Along with the R6 HooksObject in Xt, a new client-side extension hook was added to Xlib. An extension procedure may be installed via XESetBeforeFlush() which is called whenever the Xlib request buffer is flushed [4]. The motivation for this change was that an interface transformation system (or other applications interested in low-level protocol information) could install a procedure via XESetBeforeFlush to pass the low-level X protocol information which was previously available only through pseudoserver-based approaches.

Like the changes to Xt, the R6 Xlib does not specify any code to be installed in the BeforeFlush hook; it merely provides infrastructure which applications can use to install code.

These changes to the Release 6 Xlib and Xt libraries were based on our experiences with two versions of the Mercator system. The types of information which can be provided by these modifications are necessary to enable translation of graphical interfaces to non-visual modalities at runtime.

With the framework in hand to enable translations, we turn our attention to designing a protocol that can actually pass the information required for translation. The Inter-Client Exchange Protocol (ICE), new to X11R6, facilitates the process of developing inter-client communication protocols, such as RAP. ICE provides functionality common to many protocols (opening connections, validation of requests, closing connections to name but a few). This set of services allows protocol developers to concentrate on designing the protocol-specific messages and message handlers, while leveraging the generic protocol framework in ICE. Additional details on ICE may be found in [5] and [6].

Additional Required Infrastructure

Unfortunately, although we have the necessary framework in Release 6 to support the translation of graphical interfaces, we do not have the sufficient framework to support this translation. We lack a way to jump-start the process, initiating the RAP protocol as soon as the prospective partners can participate in the protocol.

This problem is not specific to the RAP protocol -- all ICE-based protocols face the issue of initiating the communication process. For many custom protocols, this is not a large problem. The clients on either end of the wire are explicitly aware of the protocol, and at the proper point in their execution, can take steps to setup the communication channel, via the appropriate ICE calls. But for the RAP protocol to work with off-the-shelf X clients, written without foreknowledge of the RAP protocol, transparent initiation is a significant problem.

The solution that we will propose for inclusion in Release 7 is the ICE Rendezvous Mechanism [7]. The rendezvous mechanism is designed to establish an ICE connection between an agent and a client in a manner which will not require explicit awareness of the RAP protocol. Briefly, clients wishing to speak ICE-based protocols follow this set of steps to fire up the ICE connection to support the protocol:

1. The client registers interest in the common protocol with the ICE library.
2. The client either actively tries to make an ICE connection to its partner(s) if this client is the Protocol Originator, or passively waits for its partner(s) to make the connection, if this client is the Protocol Acceptor. Authentication at this step is optional.

3. Once the ICE connection is made, the Protocol Setup request and Protocol Reply response messages are exchanged between originator and acceptor. Following the successful, possibly authenticated, exchange of these messages, the custom protocol's messages may be passed back and forth.

The information that must be made known to both sides of the channel is

- the common protocol (in our case, RAP)
- the network connection point (network id), dynamically assigned by the ICE library

To transfer this information, we are proposing the use of the Client Message mechanism to pass the protocol and network ID information, used in conjunction with a new data object, a table of protocol initialization routines. Calls will be provided to install, remove, and retrieve the protocol routines in the table.

The ClientMessage event handler, specific to each protocol, will be responsible for installing the appropriate protocol initialization routine. The protocol initializer routine, supplied by the protocol developer, will handle any required bookkeeping to fire up the protocol. This protocol initializer routine is not necessarily tied to setting up an ICE connection (although the RAP protocol will use its routine for this purpose.) Any client capable of being monitored by an agent must have its routine installed in its address space.

But one question remains: how does the RAP ClientMessage event handler get installed in the first place, to make all this happen without requiring clients to be RAP aware. The answer lies in the VendorShell source file, specific to each toolkit. Just as the Editres [8] Client Message handler is installed in Athena widget set's VendorShell implementation, so that all clients can be queried transparently by the editres program, so would the RAP client message handler be installed in the VendorShell implementation for all toolkits (Motif and Athena). In this way, any Motif or Athena-based client will be RAP aware, simply by being linked against the toolkit libraries.

The Remote Access Protocol

This section of the paper describes the format of the Remote Access Protocol itself. We describe the semantics of each message in the protocol (that is, what each message *means*), the circumstances under which each message is sent, and the format of the messages. This list is not meant to be definitive or final; it represents the current state of the protocol as used by Mercator [9][10].

Whereever the term "widget ID" is used in the description of a particular message, we are referring to a 32-bit unique identifier for a toolkit object in the application. While in the Xt implementation of RAP these 32-bit values will map directly onto widget IDs, other toolkits may use them to identify their particular constructs.

Further, some messages may not be implemented for certain toolkits. See the DoActionRequest for example.

There are three basic types of messages in RAP: requests, replies, and notifications.

Requests are RAP messages which travel from an external agent process to an application. They are used to ask the application for information about its interface, or to control some aspect of the application.

Replies are responses to specific requests for information sent from the agent. Replies travel from the application to the agent.

Notifications are asynchronous messages which are used to notify an interested agent in a change in the status of the application's interface. Via a set of notification-control requests, the generation of notifications within the application can be selectively filtered.

GetResourcesRequest

The GetResourcesRequest message passes a list of widget IDs to the application; the application responds with a list of the names and types of the resources in the specified widgets (see GetResourcesReply).

QueryTreeRequest

The QueryTreeRequest asks the application to transmit its entire widget hierarchy back to the calling agent via a QueryTreeReply.

GetValuesRequest

GetValuesRequest is used to ask for the values of a set of resources on a set of widgets. The message passes a list of structures; each structure includes a widget ID and a list of resources to retrieve the values of for the specified widget. The application generates a GetValuesReply with the requested information.

SetValuesRequest

The SetValuesRequest message is used by an external agent to change the value of a particular set of resources in a particular set of widgets. The message encodes a list of structures; each structure contains a widget ID, and a list of resource-value-type tuples specifying all of the resources to change in the particular widget, the new values, and the type the value is expressed in.

AddNotifyRequest

The AddNotifyRequest is used to control filtering of notifications sent from the application. The message passes a list of names of callback lists in the HooksObject. Any callback lists specified in the request will be "turned on," meaning that they will generate notifications to the external agent.

RemoveNotifyRequest

RemoveNotifyRequest is used to disable particular notifications by "turning off" callbacks in the HooksObject. Like AddNotifyRequest, it takes a list of names of callback lists to disable.

ObjectToWindowRequest

The ObjectToWindowRequest is used to resolve a widget ID into the primary window used by that widget. The message takes a list of widget IDs to resolve, and generates an ObjectToWindowReply message from the application.

WindowToObjectRequest

WindowToObjectRequest it used to resolve a window ID into the primary Xt widget associated with that window. The message sends a list of window IDs to resolve, and generates a WindowToObjectReply from the application.

LocateObjectRequest

The LocateObjectRequest message is used to find the on-screen locations of a specified list of objects. The message generates a LocateObjectReply return from the application.

GetActionsRequest

The GetActions Request is used to retrieve a list of actions associated with a group of widgets. The message contains a list of widgets.

DoActionRequest

This request is used to invoke an action. It contains a widget and an action name which together specify the action to run.

This message will most probably not be provided in the Xt implementation of RAP, because there are situations in which Xt-based applications may become unstable if actions were invoked directly.

SelectEventRequest

The SelectEventRequest message is used to control which event types are sent from the application to the agent. Whenever connection between an agent and a client application is initiated, a procedure is installed in the WireToEvent client-side extension slot which can transmit X Events to an agent. This message can be used to filter what events are actually sent.

SelectRequestRequest

This message is used to filter the transmission of X Protocol requests from the application to the agent. Requests are “caught” by a routine installed in the BeforeFlush client-side extension and are sent to the agent if their types have been selected by SelectRequestRequest.

CloseConnectionRequest

This message is generated by an agent whenever it wishes to terminate its communication with a client application. The client should take any appropriate clean-up action and then disconnect.

GetResourcesReply

This message is generated in response to a GetResourcesRequest; it contains a list of structures. Each structure contains the ID of a requested widget, and a list of the resource names, classes, and types for that widget.

QueryTreeReply

QueryTreeReply is generated in response to a QueryTreeRequest and returns the widget tree of the application. The information is formatted as a list of tuples, with each tuple containing the widget ID, name class, window ID, and parent widget ID.

GetValuesReply

This message returns a list of resource values for specified widgets to the caller. The format of the message is a list of structures, with each structure containing the ID of a requested widget, and a list of the requested resource names and values for that widget.

ObjectToWindowReply

The ObjectToWindowReply message returns a list of the windows corresponding to the objects requested via ObjectToWindowRequest.

WindowToObjectReply

WindowToObjectReply returns a list of the objects associated with the windows passed via WindowToObjectRequest.

LocateObjectReply

The LocateObjectReply message returns an X, Y coordinate pair for each of the objects requested via LocateObjectRequest.

GetActionsReply

The GetActionsReply is generated in response to a GetActionsRequest and contains a list of action names for each widget specified in the GetActionsRequest message.

CreateNotify

The CreateNotify message is generated by code installed in the CreateHook callback list in the HooksObject. CreateNotify is used to inform an agent whenever a new widget has been created. The messages passes the widget ID, name, class, window ID, and parent widget ID to the agent.

ChangeNotify

The ChangeNotify message is generated via the ChangeHook whenever a resource value is updated. The message sends the widget ID, resource name, and new resource value to the agent.

ConfigNotify

A ConfigNotify message is sent to any interested agents whenever the ConfigHook callback is executed. This message contains information on widget configuration changes: the widget ID, a geometry mask specifying the actual changes which have taken place, and an XWindowChanges structure describing the changes. The message is sent after the changes have taken place.

GeometryNotify

GeometryNotify is generated whenever the GeometryHook callback is executed. This occurs when Xt application makes a geometry request. The message contains the resulting widget geometry, expressed as an XtWidgetGeometry structure. The message is sent after the geometry negotiation has completed.

DestroyNotify

DestroyNotify is used to inform an agent that a widget has been destroyed. The message contains the ID of the newly-destroyed widget, and is generated via the DestroyHook callback list.

RequestNotify

RequestNotify passes X protocol requests to the agent. The actual requests which are sent may be selected by the SelectRequestRequest message. The message contains one or more wire-format X protocol requests; it is generated by code installed in the BeforeFlush client-side extension.

EventNotify

EventNotify passes X events to the agent. The event types which are sent may be selected via the SelectEventRequest message. The message contains one or more wire-format X events, and is generated by code installed in the WireToEvent client-side extension.

Status and future directions

Our current implementation of the Mercator system is built using a protocol which predates (but is essentially similar) to RAP. This protocol runs over standard TCP/IP sockets. Implementation of RAP itself, as well as implementation of the changes to Mercator to use RAP, are ongoing.

Will Walker is serving as the RAP architect, and has begun implementation work on the ICE rendezvous mechanism, and the RAP protocol messages. The public forum on which RAP-related issues are discussed is the x-agent public mailing list, x-agent@x.org. This mailing list is sponsored by the X Consortium for the purposes of developing protocols like RAP for external agent software. Anyone who is interested in contributing to the effort may join the list by sending a message to

```
x-agent-request@x.org
```

The body of this message should contain the single word “subscribe”. The subject line of the message will be ignored.

At last year’s X Technical conference, Philippe Kaplan and Anselm Baird-Smith of Bull France announced their efforts in a next generation Editres protocol, called K-Edit. K-Edit is “an attempt to provide a simple multipurpose protocol, suitable (in particular) to export an application user interface state” [11]. There is enough overlap between the goals of the K-Edit protocol and the RAP protocol that there has been continuing discussion over the last year, with the possibility of combining the two efforts into a common protocol.

The ICE rendezvous mechanism and ClientMessage handler outlined in the Additional Infrastructure section will be proposed for inclusion in Release 7. However, those proposed

changes can be added to R6 clients today. For dynamically-linked applications, all that must be done is to rebuild the Intrinsics library from source modified to support the rendezvous and ClientMessage. The next invocation of the dynamically linked clients will be RAP aware, as the library they utilize dynamically will have been made RAP aware. Statically linked clients will need to be relinked against a newly compiled static Intrinsics library, but in keeping with the goals for the RAP protocol, the clients themselves will not need to be modified at the source code level in any way.

Since the format of the RAP protocol is still evolving, we expect changes in the protocol between now and the Release 7 cut-off date. One area where we need to focus is resource type negotiation. Certain types used by resources within an application may not be known to a particular external agent; yet the application may have the ability to convert these types to a format which the agent can understand. Conversely, the agent itself may have the ability to convert certain types internally, even if the agent does not have the required converters linked into it. Thus a process where resource types are negotiated between agents and applications may be useful.

Summary

We have presented a protocol for communicating changes in interface state between X applications and agent processes. This protocol is powerful in that it can support a variety of useful agents including interactive resource editing, scripting, testing, and interface translation.

The protocol builds on a set of hooks, some of which are already present in Release 6 of the X Window System, and some of which are still pending.

We acknowledge the contributions of many people to this project: Tom Rodriguez (formerly of Georgia Tech, now at Sun Microsystems, who designed the protocol on which RAP is based; Bob Scheifler, Donna Converse, Ralph Swick, Daniel Dardailler, and Kaleb Keithley (of the X Consortium) who provided much-needed design guidance; and the members of the Disability Action Committee on X (DACX). Thanks also to Sun Microsystems and the NASA Marshall Space Flight Center for their sponsorship of Mercator.

Author Information

Keith Edwards, Susan Liebeskind, and Beth Mynatt work at the Graphics, Visualization, & Usability Center at Georgia Tech. Their email addresses are keith@cc.gatech.edu, shl@cc.gatech.edu, and beth@cc.gatech.edu; they can be reached via telephone at (404) 894-3658. Will Walker is a Software Engineer at Digital Equipment Corporation. His email address is wwalker@zk3.dec.com.

References

[1] Elizabeth Mynatt and W. Keith Edwards. *Mapping GUIs to Auditory Interfaces*. In Proceedings of the Fifth ACM Symposium on User Interface Software and Technology (UIST), Monterey, CA, November, 1992.

[2] Elizabeth Mynatt and W. Keith Edwards. A*n Architecture for Transforming Graphical Interfaces*. In Proceedings of the Seventh ACM Symposium on User Interface Software and Technology (UIST), Marina Del Rey, CA, November, 1994.

[3] Joel McCormack, Paul Asente, Ralph Swick. *X Toolkit Intrinsics C Language Interface.* X Consortium Standard. Version 11, Release 6.

[4] James Gettys and Robert Scheifler. *Xlib C Language Interface*. X Consortium Standard. Version 11, Release 6.

[5] Robert Scheifler and Jordan Brown. *Inter-Client Exchange (ICE) Protocol, Version 1.0*. X Consortium Standard. Version 11, Release 6. 1994.

[6] Ralph Mor. *Inter-Client Exchange (ICE) Library, Version 1.0.* X Consortium Standard. Version 11, Release 6. 1994.

[7] William D. Walker. *An ICE Rendezvous Mechanism for X Window System Clients*. Unpublished report, Disability Action Committee for X, 1994.

[8] Chris D. Peterson. *Editres: A Graphical Resource Editor for X Toolkit Applications.* In Proceedings of the Fifth Annual X Technical Conference, Boston, MA, January, 1991.

[9] W. Keith Edwards and Tom Rodriguez. *Runtime Translation of X Interfaces to Support Visually-Impaired Users.* In Proceedings of the 7th Annual X Technical Conference, Boston, MA, January 18-20, 1993.

[10] William D. Walker. *Remote Access Protocol (RAP), Version 0.2.* DACX Work In Progress. November 11, 1993.

[11] Philippe Kaplan and Anselm Baird-Smith. *The K-edit System.* Unpublished report available via http://zenon.inria.fr:8003/koala/k-edit.html.