Towards Wide-Area Distributed Interfaces

Steve B. Cousins, Andreas Paepcke, Scott W. Hassan, and Terry Winograd

{cousins, paepcke, hassan, winograd}@CS.Stanford.EDU

ABSTRACT

We use our implementation of a distributed interface for digital libraries to explain several design alternatives for distributed Web applications. Our focus is on an analysis of how functionality should be distributed between clients and servers. Based on our prototype and related performance results, we conclude that it is useful to move some user interface related functionality from server to client, but that the bulk of functionality can and should remain with the UI server.

We also argue that at least a minimal set of CORBA features, namely method call semantics and separation of interface and implementation specification, are useful enough to consider for future inclusion in World-Wide Web standards. Beyond that, our experience shows that asychronous method calls with guaranteed, in-sequence delivery would be highly desirable.

Introduction

There are two sides to information access: the user side and the service side. The WWW has done a fantastic job of optimizing the user side: Web browers are ubiquitous, and the number of people who can access Web services is growing exponentially. But the simplicity of HTTP/HTML, which has allowed browsers to become so widespread, has also made it harder for services to provide the functionality they would like. The WWW is now evolving to make it possible for service providers to tailor the user experience, with Java, and with richer alternatives to HTTP on the horizon.

One of the alternatives that is being considered is CORBA. CORBA is a distributed object standard based on the notion of an "Object Request Broker" (ORB). When an ORB is added to an application program, objects in that program can communicate seamlessly with objects in other ORBs. The CORBA distributed object scheme allows programmers of distributed applications to use high-level constructs such as rich data types, objects, and methods with parameters. In other words, CORBA gives distributed application builders programming amenities programmers need to build large systems.

Combining Java with a Java ORB provides a platform with the best of both worlds. In this scheme, the ORB and application are delivered to the client via Java. The objects within the application then communicate with associated application objects elsewhere by way of method calls. Although Java+CORBA provides a good base platform, application developers still need to address a number of problems:

- how to split functionality between distributed objects
- how to get good performance
- how to overcome mismatches between Java and CORBA
- how to provide adequate protection and security

We use a substantial distributed application, DLITE, to explore the first two of these problems. DLITE, the Digital Library Intergrated Task Environment, allows users to interact with multiple, heterogeneous remote services. We describe relevant pieces of this application and then analyze the requirements it places on a widely distributed computing system, such as future versions of the World-Wide Web. We conclude with preliminary performance results. Our goal is to inform Web

application builders and Web infrastructure development with experiences from our current system.

DLITE and Its Distribution Technology

DLITE is a distributed interface to one implementation of a Digital Library. We expect Digital Libraries to include large numbers of autonomously maintained information repositories and publication-related services. Repositories might include traditional collections, such as citation catalogs, electronic newspaper archives, reference works, or tabular data, such as census results. They could also include smaller, individually maintained collections, such as news articles on particular topics, hotlists, individually collected bibliographies, or collections of electronic designs. Services include document summarization, format translation, copyright services [1], payment management [2], and electronically mediated human consulting.

This vision poses both user interface and architectural challenges: how can the interface be made to help users harness small subsets of these large numbers of services and collections to accomplish their tasks? For example, a product manager responsible for marketing and planning future products must often monitor the press and other sources for news about the portfolio of products she supports. She needs to summarize this material and maintain a local collection of the results.

Here is a summary list of five key requirements for Digital Library systems that emerge from our vision and from the example:

1. Digital Library resources must be accessible from anywhere on the Internet. Digital Library interfaces must therefore be supported for multiple software and hardware platforms
2. It must be easy to send the output of one resource as input to another (e.g. search results to summarization service)
3. Digital Library systems need to be extensible enough to allow existing and future resources to be integrated easily, without requiring any changes to those resources.
4. Library interfaces must persist over time because the human tasks they support may vary in length and because the time required for different resources to do their work may range from milliseconds to days or longer.
5. Interfaces must enable collaboration.

Our transport and integration prototype, the InfoBus [3, 4], uses CORBA distributed objects [5] to access these remote resources. CORBA is an evolving standard which supports the writing of object-oriented programs whose constituent objects may be implemented in different languages and may reside on different hardware platforms.

Figure 1 summarizes our use of CORBA objects as relevant in this context. The left side of the figure shows resources 'out in the world'. These could be any of the resources listed in our scenario. Access protocols and communication paths for connecting with them are pre-determined because they are autonomous from the point of view of the library.

We inject a degree of insulation from this heterogeneity by introducing *library service proxies* (LSPs). These are objects that are located between the outside resources and our client programs. Usually, one LSP represents a given autonomous Digital Library resource, such as a collection or service. Sometimes one LSP provides access to multiple outside resources.

Each LSP supports at least two interfaces. One interface is used to communicate with the target resource. The mechanisms for this depend on the resource. For example, proxies accessing Web-based resources use HTTP. Proxies accessing Knight-Ridder's Dialog Information Service run Telnet sessions instead. Many library resources are available through Z39.50. The second interface is method-based and much more uniform. Client programs call the proxies via remote method calls. Clients are individual applications such as multi-search programs which distribute a single query to multiple, possibly heterogeneous information sources. Other kinds of clients are objects representing user tasks, such as the tracking chore of

our scenario.

Note the two differently shaded vertical lines in Figure 1. The left line is a reminder that there is a protocol, and potentially a network boundary between the elements of these programs. The vertical line on the right indicates that there may be a network boundary, but that interactions occur via method calls. The set of possible methods and their signatures are specified using CORBA's *Interface Definition Language* (IDL). This means that implementations on either side of the right-most vertical line in Figure 1 may be changed independently, as long as the agreed-upon interface is not compromised.

Figure 1 also shows how the DLITE user interface interacts with remote services as shown in Figure 1. Note that outside resources, LSPs, instances of the UI hierarchy, and the rendered views may all reside on different machines across the network. The client in Figure 1 is shown within a dark border, and its implementation is depicted in the inheritance hierarchy next to it. Each element on the canvas is managed by an object which inherits from the Component base class. These components often play the role of CORBA Client Objects. For now, this border includes all the elements on the canvas, their control, interaction with LSPs, and rendering.

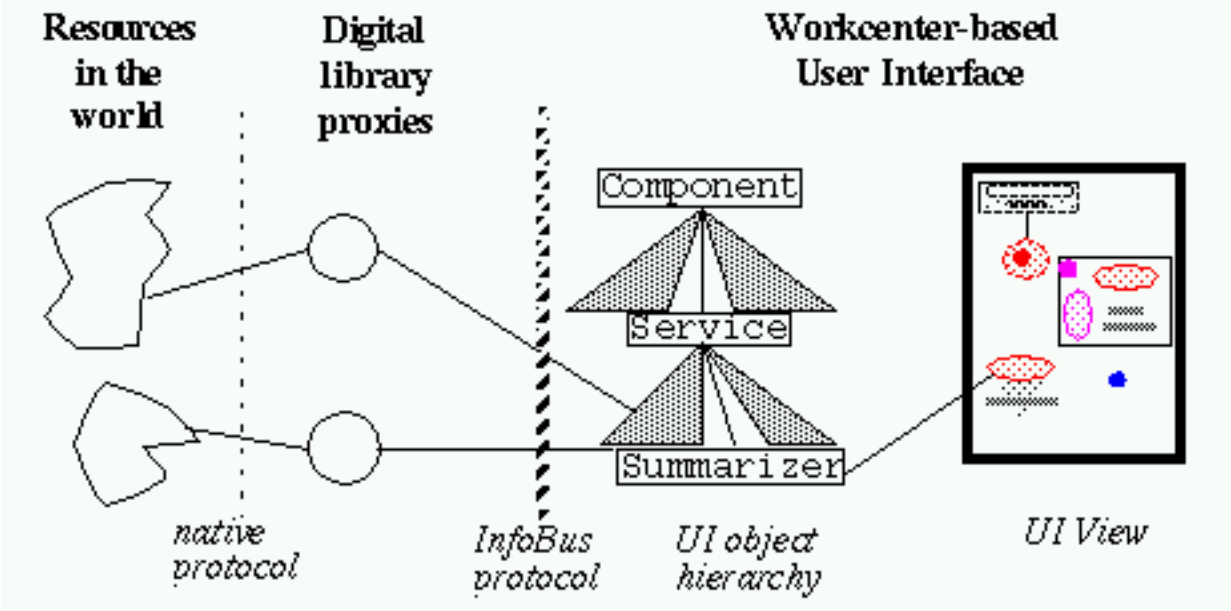

Figure 1: Digital library architecture. The interface consists of everything to the right of the "InfoBus protocol" line. Services to the left of the "native protocol" line may be out of our control.

DLITE affords direct manipulation of entities representing documents, queries, collections, and services. These manipulations may be performed from one or more viewers. Components are represented by icons on a canvas, and can be arranged and manipulated by the user. Services are active, and are invoked by dropping their arguments on them. For example, dropping a query on a search service icon invokes the service. The results of the search are represented as a collection.

Figure 2 shows a very simple example for the product tracking task described earlier. Given the desire to afford real-time collaborative interaction among remote users (requirement 5), multiple instances of the interface may be active at the same time.

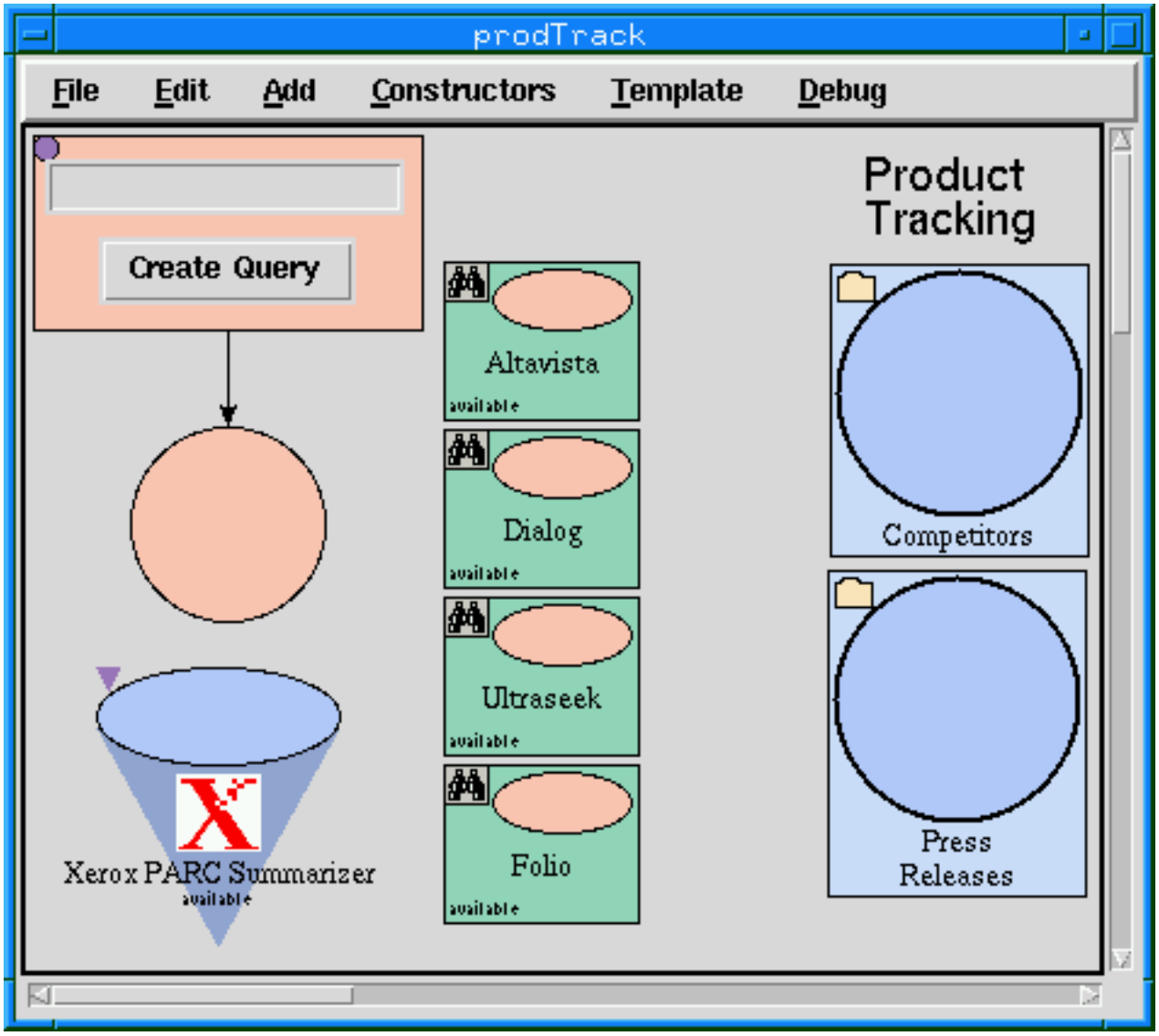

Figure 2: A sample digital library task interface for searching various services, summarizing documents, and building result collections.

Users may move interface components around; they may delete some of them, or add new ones. Inputs are type sensitive and reject inappropriate parameters. Complete details of the intraction may be found in [6].

Underlying all these components and their manipulation are, of course, CORBA objects spread across the computational environment. The interface of Figure 2 implies underlying system activities at two levels. At one level are the coordinated activities of the task as a whole. These include management of interface persistence (requirement 4), the dragging of interface items, or constraints a task might want to place on the use of any individual component. For example, it might be required that a task limit the total number of resources used to stay within some budget.

At a second level, each individual interface component needs to communicate with its associated proxy. The Altavista icon in Figure 2, for example, needs to interact with the Altavista proxy whenever a query object is dropped on it.

The details of how all this functionality is distributed are critical for the success of the interface.

How to Split Functionality

To keep the terminology clear, we divide the user interface into two parts and refer to the part closest to the user as the *UI-client* and the part between the UI-client and the rest of the InfoBus as the *UI-server* .

The challenge is to determine where to place different pieces of functionality: UI-client or UI-server. The UI-client is special because any functionality placed there needs to be supported on many software and hardware platforms, and it must connect to the UI-server functionality through a variety of network technologies that may vary greatly in latency and bandwidth (e.g. local area network vs. wireless links). In the simplest situation, the UI-server contains *no* functionality.

Design Tensions

The decision of how to distribute functionality between the UI-client and the UI-server greatly influences the speed of the application. When considering speed in this paper, we are concerned with latencies in user interface manipulations, rather than the speed issues between other parts of the system, such as the real-world services and the LSPs. For example, if we want a document object to turn color when dragged over an inappropriate interface component, we cannot afford to send numerous events from the 'dumb' rendering to the remote 'smart' portions of the system. This may pose no difficulties if both are within the same process boundary, or at least within the same machine. But if there are network latencies as is the case in distributed interfaces, latency quickly becomes problematic.

There are a number of reasons for splitting the user interface:

- Increase the ease of maintaining and porting functionality to different platforms (requirement 1). We might want implementations based on X Windows, Java applets, Visual Basic, NeWS [7] or InterViews [8] to ensure that as many clients as possible can have access to the library.
- Only part of the functionality needs to be re-implemented for legacy applications, or those which need to run on multiple platforms.
- Splitting affords remote consultation or collaborative work in the Digital Library, by making it possible to display one interface view at multiple locations.
- New services can be added without affecting any deployed clients.
- The UI-server part of the interface can receive results while the UI-client is off-line.
- Part of the interface may reside on machines which are faster, or have faster connections to the outside world.
- It takes less time to download partial applets.
- The application makes use of existing libraries which are not part of Java.
- If the application code contains proprietary algorithms or concepts, the developer may be reluctant to deploy it in a language which is readily decompiled.

Figure 3 shows a spectrum of solutions. The left-most solution places all functionality in the UI-client. The right-most one shifts all functionality to the UI-server. In the following sections we explore this spectrum in more detail.

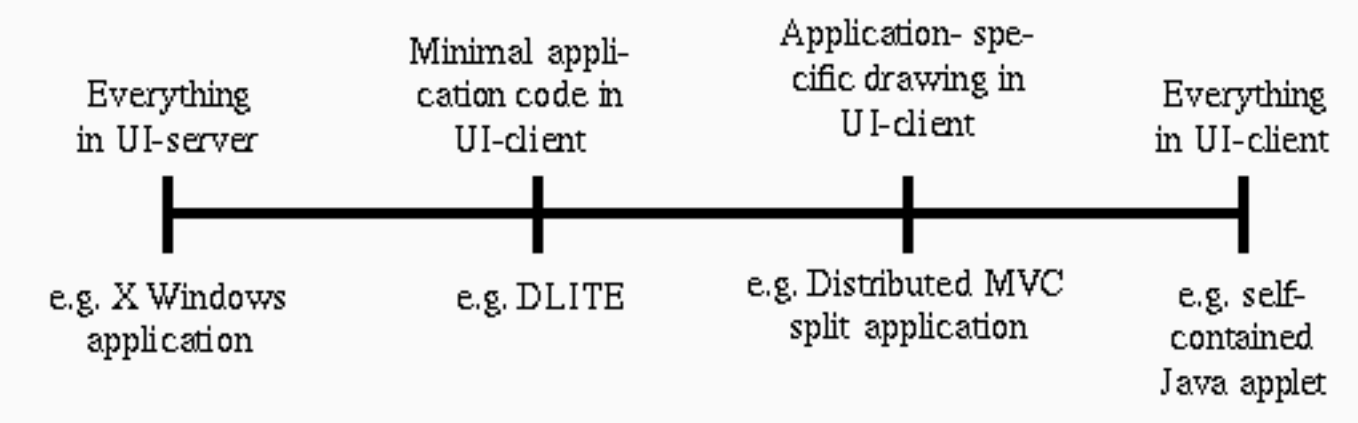

Figure 3: Spectrum of ways to distribute functionality and examples of each.

Everything in UI-client

The simplest interface to design and build is one which is not distributed at all. In our framework this means that all functionality is in the UI-client. Our first prototype implemented this architecture. The interface was encapsulated in a single process, written in Python and using the Tk graphics toolkit. We considered addressing the problem of ubiquitous access (requirement 1) by rewriting this interface in Java with its Abstract Windowing Toolkit (AWT). However, for a number of the reasons listed above we decided against this solution.

Distributed Model/View/Controller

Following object-oriented principles, one idea for effecting an appropriate UI split is along the lines of Smalltalk's Model-View-Controller (MVC) architecture [9]. We would then have a view subclass for each component subclass. Figure 4 shows this alternative.

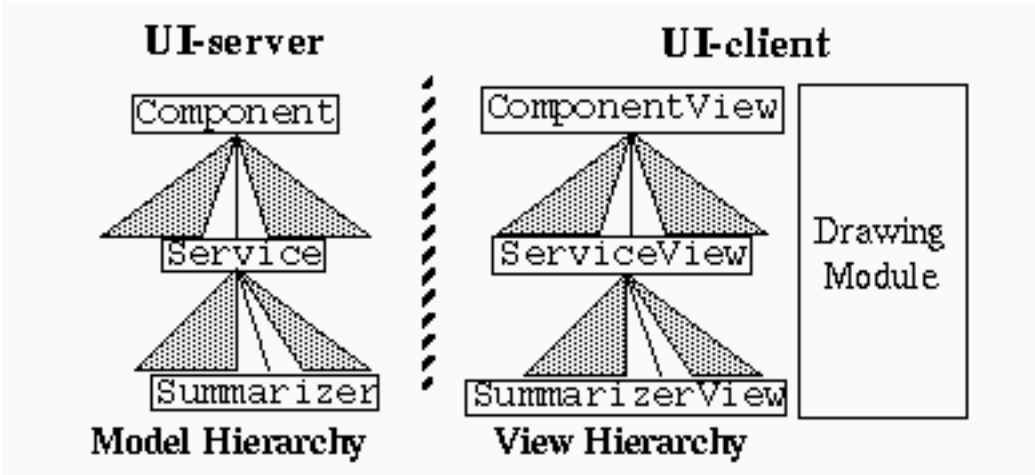

Figure 4: Distributed-MVC-split architecture. Use the natural separation between Model and View to decide how functionality is distributed.

For example, the SummarizerView class would have methods for drawing a Summarizer object. The Summarizer object would reside with the other components on the UI-server side of the boundary, and the SummarizerView would reside on

the Viewer side of the boundary.

Although this architecture can be implemented efficiently using caching [10], this design would make it very difficult to maintain multiple versions of the viewer code. Furthermore, it would be difficult to add new services, since each new service would require the interface specification to be updated (violating requirement 3). This approach is very similar to traditional client-server architectures, but for those systems the specifications are usually more static and the goal is not to support the wide variety of hardware and software platforms we need to support.

Everything in UI-server

Another idea is to make the viewer a pure drawing tool, with no application-specific knowledge at all. This leads to the architecture in Figure 5. X Window distributed interfaces are built this way, but latency problems have kept X Window interfaces from being widely used over modem and other high-latency links, even though protocols such as PPP make its use possible. There is work in progress at the X consortium to build an X interface which runs over the Web, but it is not yet available, and its latency tolerances are not yet known.

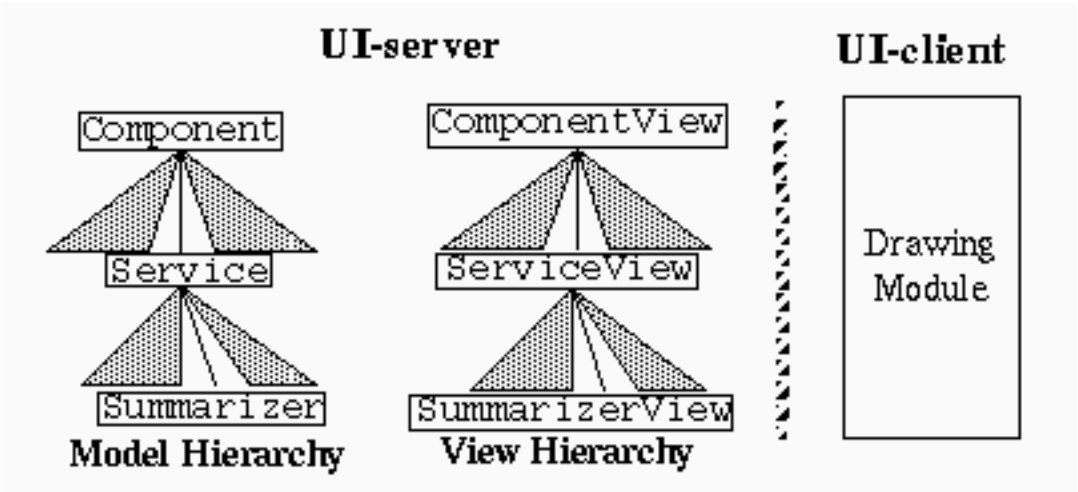

Figure 5: Everything in UI-server. Keep the Models and Views together, but distribute the Drawing Module (e.g. X Windows). Note that it is no longer strictly necessary to split the Model and View hierarchies in this architecture.

In addition, personal computers are becoming more powerful and prevalent, and it makes sense to off-load as much processing to the UI-client as possible given other constraints.

It may still be a good idea to have two hierarchies for software engineering reasons, but the two hierarchies are not required in the X-split architecture.

Hybrid Architecture

The solution we settled on is between these extremes. Some application specific functionality is placed at the viewer, but the viewer does not need to have a class for each component (Figure 6). The requirements of the viewer are completely specified in an interface specification, so implementation of the interface in a new system is at least a well-defined task. Our viewer encapsulates all of the functionality that is specific to the Digital Library application in a single class. This class inherits behavior from any drawing classes which are available on the view side. This keeps the problem of porting to different UI-client platforms even more contained.

Our UI-client code is about 3000 lines of code. The UI-server code is roughly 24,000 lines. This implies that necessary porting effort for clients is cut to about 1/8 by the UI split.

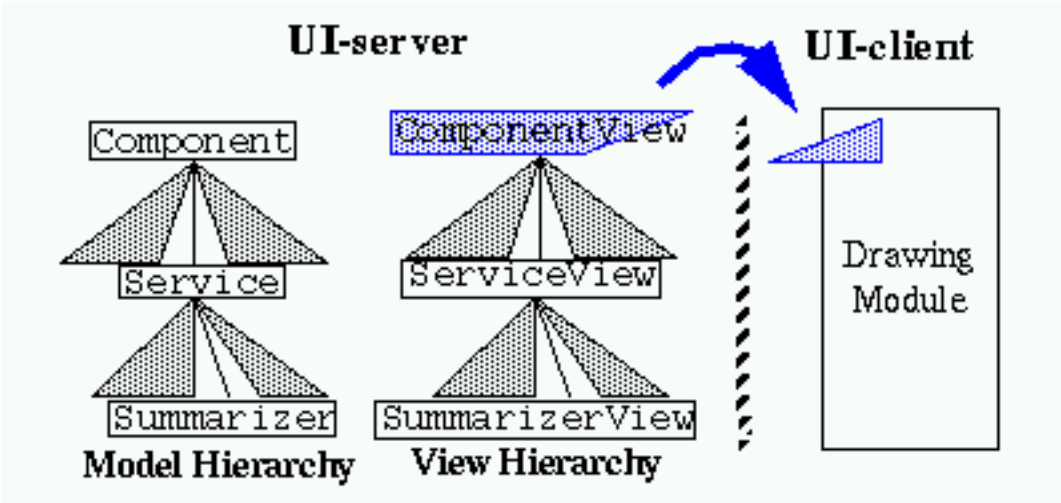

Figure 6: Hybrid architecture. Move a small piece of application specific code across the CORBA boundary. This piece is designed to reduce latencies while keeping the CORBA interface simple and the Drawing Module extensions minimal.

The decision of which functionality to move to the UI-client is driven by performance and portability considerations. Any highly interactive behavior which does not require control from the UI-server is a good candidate for transfer. However, any code associated with this behavior must then be re-written during a port of the UI-client.

For example, our architecture allows a user to drag items around the screen with good response. This is because move tracking has been shifted to the UI-client. When a user moves an object around, no network traffic is generated, and response is immediate. When the user releases the mouse, a single message is sent from UI-client to UI-server, and the state of the UI-server is updated. In contrast, with typical latencies of dial-up users of between 200 and 400 milliseconds, the animation rate would only be 2 to 5 frames per second without the UI split.

Sometimes it is not sufficient to transfer completely abstract behavior, such as dragging, to the UI-client. For example, in DLITE, we display help 'bubbles' when users slide the cursor over objects on the screen. In order to enable this facility, the UI-client also needed to be made aware of this bit of application semantics. Obviously, such awareness should be kept to a minimum. Just how much semantics to make UI-clients aware of is again driven by performance consideration.

Occasionally, the transfer of functionality is even more complicated. The most difficult behavior we have put into the UI-client is "copy off". The interaction we wanted was for the user to be able to mouse down on an object and drag a copy of it away. Since copying can be somewhat time consuming, we needed a trick to avoid the perceived delay by the user during the onset of the move. The UI-client therefore creates a new temporary object which is dragged by the user. The temporary object contains a pointer to the original Component object, and when it arrives at its destination, the temporary object invokes over the network a copy operation of that Component. The new Component copy is created and placed at the new location, and the temporary object is then deleted. The message traffic occurs on mouse up, when the user expects a slight delay while the computer works, but it does not interfere with the sensation that dragging objects is a smooth operation.

If the UI-client and UI-server need to communicate frequently in spite of careful partitioning of functionality, some

additional optimizations must be used. For such cases we have been able to improve performance significantly without giving up our abstractions or greatly changing our code. We have done this by using asynchronous method calls.

From the programmer's point of view, asynchronous methods are procedure calls which return immediately and return no values. Since asynchronous method calls can be pipelined, they speed up interactions greatly when many method calls are made in a row.

In our non-distributed prototype of DLITE, all of our drawing was accomplished by function calls to a drawing library. When we converted the application, we replaced the native graphics library calls with remote CORBA method calls. As a consequence, when objects are drawn, many consecutive method calls are issued.

Synchronous method calls take CD+2*L+PD time each (Figure 7), where CD is the call delay (including the time taken by the underlying system to marshall the parameters for the function call and any other processing done between calls), PD is the processing delay (including the time to decypher the parameters and the time actually spent in the method), and L is the link latency. From the client's point of view, an asynchronous method returns after only CD, which is practically immediate. To be fair, we should consider asynchronous methods which return a value by sending another asynchronous message back. In this case, the time taken for one asynchronous method and its return would be the same as for synchronous calls. However, if multiple calls are present, pipelining remains an advantage. In our application, many calls did not require return values.

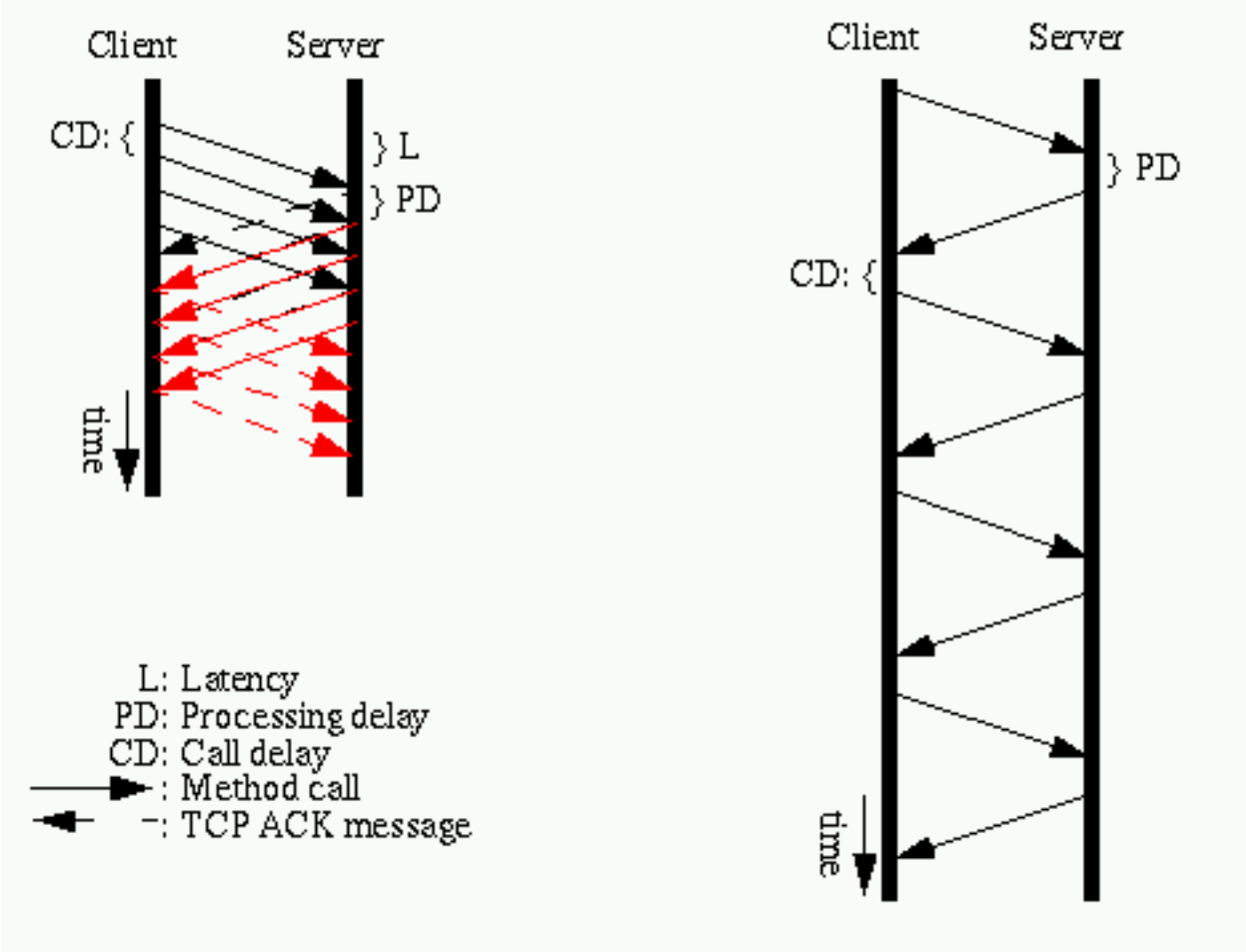

Figure 7: Performance of Asynchronous vs. Synchronous Method calling. Asynchronous method calls overlap in time, while synchronous calls are serialized. If return values are required, two asynchronous messages, one in each

direction, may be used.

Figure 7 also shows how asynchronous method calls are pipelined. The time for 10 asynchronous method calls, from the first call until the 10th response is received, is bounded by 11*MAX(CD,PD)+2*L. By contrast, 10 synchronous method calls takes 10*(CD+2*L+PD). Clearly, if either the latency or processing delay is significant, and the number of objects to draw is significant, using asynchronous method calls will be much faster.

In ILU, changing from synchronous to asynchronous methods is just a matter of adding the keyword "ASYNCHRONOUS" to the method definition in the interface description, so at a syntactic level, the code changes are minimal.

Table 1 shows the difference between the synchronous and asynchronous versions under different latencies. The experiment was done by drawing an object with 100 graphical elements, compared with the 5-10 graphical elements of typical components in DLITE. Note that because this experiment makes use of an experimental, un-optimized CORBA implementation for Java, the reader should look at the relative differences between the asynchronous and synchronous, rather than the absolute numbers.

Table 1: Performance of DLITE using Synchronous vs. Asynchronous method calls.

Communication Mechanism	Latency	Synchronous	Asynchronous
Local ethernet	0.7ms	17.75s	4.90s
Remote ethernet	80ms	62.0s	19.5s

In practice, making methods asynchronous raises two issues: Since asynchronous methods cannot return values, the application may need to be modified; and asynchronous methods are not guaranteed to be serialized.

Handling return values. In our application, UI-servers occasionally need handles to the graphical primitives produced by the UI client so the primitives can be modified later. We do this by introducing a simple callback scheme: We added a parameter to the drawing call requesting the callback. The parameter provides a cookie, which is used in the callback handler to properly associate the result value with the graphical primitive.

This solution works because we do not use the return values immediately. If we did, we would need to find another solution, or perhaps a synchronous method would be required.

Serialization. Unfortunately, neither the ILU implementation nor the CORBA standard guarantee that asynchronous methods will be called on the destination object in the order that they were invoked. The CORBA specifications are based on datagrams, and do not even guarantee delivery of the message.

This problem required us to add a serialization parameter to each drawing method, and fix the serialization at the receiving end. Our two implementations handled this in two ways. In Python, when the serialization number on the method indicated that it was out of order, we simply put the message into a queue until all the preceeding methods had been handled. In Java, since we do not have pointers to functions, this approach would have required a large case statement for dispatching. Instead, the Java version uses the serialization parameter to enter the drawing elements into the proper position in a display

list.

Implementing the Alternatives

We have discussed four ways that the functionality could be distributed across different processes. Implementing these alternatives could be done in a number of ways. Since our Component objects already need to comunicate using CORBA distributed object protocols, CORBA was a natural choice for implementing the communications protocol between the UI-client and UI-server, especially because of CORBA's multiple language bindings.

Our first implementation was done in Python using the ILU implementation of CORBA and the Tk graphics package. When we split the interface, we first did so by creating an ILU interface specification which defined the boundary between the UI-server and the UI-client. We then moved existing drawing code into a separate program which became the UI-client.

Once we had split the first implementation, we wrote a new UI-client in Java using the AWT windowing toolkit. We used a locally-developed implementation of ILU for Java for the communication with the UI-server. Bringing up the UI-client in Java took less than 3 person-months, which is significantly less work than would have been required to rewrite the entire DLITE interface in Java.

Other technologies could have been used to implement the architecture we have designed. For example, the NeWS system [7] was a competitor to X Windows in the early 1990s which allowed functionality to be moved to the UI-client. NeWS required that all local functionality be written in Postscript, and supplied by the UI-server. If NeWS was a viable platform today, we could use it to implement another UI-client for our UI-server.

This distributed interface could also be implemented using low-level networking calls. For example, it could be built using sockets directly. Doing so would force us to define a protocol for the connection down to the byte level. For some applications, such a step may be necessary, but we view it as an optimization analogous in costs and benefits to programming in assembly language.

Conclusions

We have designed and implemented a prototype distributed Digital Library interface. CORBA technology has given us the convenience of high-level programming languages across wide-area networks. It has also allowed us to move user interface functionality to the places in the network where it is most efficient.

There are several lessons we derive as input to the design of highly interactive World-Wide Web applications, and to future protocol developments for the Web.

If both interactive response times and portability of client-side capabilities are important, the choice of placing UI functionality with the UI server or the UI client is critical. Based on our experience, we suggest leaving the bulk of the functionality with the UI server, but moving very select elements of functionality and semantic application knowledge to the UI client. In our prototype this cut the bulk of code to be ported by a factor of about 8 while retaining good response time for critical actions such as dragging.

The use of asychronous method calls significantly improved overall performance. This is in part due to the fact that our UI operations frequently did not require return values. But even when return values are needed, the pipelining effect of asynchronous calls was beneficial. However, in contrast to CORBA specifications, we recommend that asynchronous

calling facilities guarantee delivery and preserve sequence.

Our use of distributed object technology (CORBA in this case) served us well. While simpler facilities such as CGI would have required less setup and learning time, the size of our project made amenities such as automated parameter mashalling indispensable. The free ILU 2.0 implementation from Xerox (ftp://parcftp.xerox.com/pub/ilu/ilu.html) has become robust enough that it is now possible to build the kind of distributed interfaces we have described. As new languages and platforms become available, ports of CORBA-like specifications are quite feasible, as was demonstrated by our Java port.

We found that the formal separation of interface and implementation class hierarchies helped us separate and identify code that potentially needed to be ported to other platforms. The discipline of keeping interfaces separate served us well in handling the complexities of our large and widely distributed system.

Acknowledgments

This work is supported by the NSF under Cooperative Agreement IRI-9411306. Funding for this cooperative agreement is also provided by ARPA, NASA, and the industrial partners of the Stanford Digital Libraries Project. Alan Steremberg implemented the Java UI-client. Thanks to Keith Marrs for doing the cross-country experiments from St. Louis.

References

[1] N. Shivakumar and H. Garcia-Molina. The SCAM Approach to Copy Detection in Digital Libraries. *CNRI D-Lib Magazine*, November, 1995.

[2] S. Cousins, S. Ketchpel, A. Paepcke, H. Garcia-Molina, S. Hassan, and M. Röscheisen. InterPay: Managing Multiple Payment Mechanisms in Digital Libraries. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*, 1995.

[3] Andreas Paepcke, Steve B. Cousins, Héctor García-Molina, Scott W. Hassan, Steven K. Ketchpel, Martin Röscheisen, and Terry Winograd. Towards Interoperability in Digital Libraries: Overview and Selected Highlights of the Stanford Digital Library Project. *IEEE Computer Magazine*, May, 1996.

[4] Edward Chang and Héctor García-Molina. Reducing Initial Latency in a Multimedia Storage System. In *Submitted for publication*, 1996.

[5] *The Common Object Request Broker: Architecture and Specification*. Object Management Group, http://www.omg.org, December, 1993.

[6] Steve B. Cousins, Andreas Paepcke, Terry Winograd, Eric A. Bier, and Ken Pier. The Digital Library Integrated Task Environment (DLITE). 1997. Accessible at http://www-diglib.stanford.edu/cgi-bin/WP/get/SIDL-WP-1996-0049.

[7] James Gosling, David S.H. Rosenthal, and Michelle J. Arden. *The NeWS Book : An Introduction to the Network/Extensible Window System*. Springer Verlag, 1989.

[8] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, 22(2), 1989.

[9] Glenn E. Krasner and Stephen T. Pope. A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26-49, 1988.

[10] T. C. Nicholas Graham, Tore Urnes, and Roy Nejabi. Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware. In *Proceedings of UIST '96*, 1996.>

Stanford Digital Library Project Homepage: <http://www-diglib.stanford.edu/diglib/>