Programmable API

As figure 17.3 shows, the next approach for distributing the user interface is to create a programmable user interface layer. This allows portions of the view code and portions of the controller code to be downloaded to the client. The client is then capable of handling many interactive inputs locally without incurring network latency costs as part of a tight interactive loop.

Network latency has long been a problem for distributed interaction. One of the first approaches for dealing with this problem was the *smart terminal*, notably characterized by the IBM 3270. This approach was created in the days of text-only interaction and extremely slow bandwidth. The server would assemble a *screen* of textual information and send it to the terminal for display. Embedded in the screen description were special codes that would indicate which characters could be changed and which could not. The terminal device retained a buffer of all of the characters on the screen and allowed the user to move among the changeable characters making any modifications that the user desired. When the user hit the Enter key all of the modified characters were packaged up and shipped back to the server where they could be analyzed.

The key advantage here is that the interaction mostly happened in the terminal and thus was very fast and imposed no burden on the server. There was no round-trip over the network to process each user event. In the days of very slow machines and slow disk there was no swapping in of the server application every time the user provided input. Exploiting the intelligence of the client device has important benefits when distributing the user interface.

Display list graphics

An early architecture for graphics devices was the vector refresh display[1]. In such displays the server would provide the graphics device with a *display list*. A display list is a machine-readable list of drawing commands. The display controller would refresh from this list rather than from a frame buffer. Many such systems added matrix transformations to the display list interpreter and many forms of interaction such as dragging, rotating or scaling were performed by modifying the matrices. A new matrix takes much less bandwidth to transmit than all of a display.

The display list architecture would still have the input/latency problems of the previous distribution schemes. Inputs would be sent to the server, translated into display list/matrix modifications which were sent back to the client device and then displayed. The Evans and Sutherland PS 300 extended the display list architecture to include *function nets*. Function nets were like hardware-

interpreted spreadsheet formulas. Transformations and other display list data could be computed in the display device as functions of inputs and other data. This meant that many interactions could be handled completely in the display processor without involving the server application. By linking the inputs to the display, interaction techniques such as rubber-band rectangles, dragging or scaling could be done at interactive speeds despite slow processor/graphics communications.

NeWS

As a response to the interactive problems of X-Windows the NeWS[2] (New Windowing System) system was developed, based on Postscript. Postscript[3] was developed by Adobe as a mechanism to reduce the bandwidth requirement for laser printers. The idea behind PostScript was to encode graphics information as Forth[4] programs. Forth is a very efficient interpreted language. The PostScript drawing commands were implemented as procedures called from Forth programs. A Postscript drawing is just a Forth program that executes on the laser printer.

NeWS added input event processing and windowing to the underlying drawing model of PostScript. This allowed interactive applications to be built in PostScript. This not only added the superior drawing model of PostScript but opened opportunities in the windowing system. Because the foundation was the language Forth it was possible to send small Forth programs to the windowing system that could process events and change the screen without going back to the application.

In a networked implementation such as NeWS, interactive techniques that need tight interaction with the user but less integration with the application could be downloaded as code into the remote windowing system. Scroll-bars, dragging, highlighting buttons and a variety of other techniques could be performed in the NeWS server without a network round trip. NeWS did solve the network latency problem for interactive distribution, but it suffered a variety of other setbacks. At the time computers were too slow to handle the PostScript drawing model at interactive speeds. Personal computing started to replace terminal/server computing. Programming in Forth is rather painful and highly error-prone making development of interactive techniques difficult.

WWW Interaction

When the World Wide Web appeared it created a model of interaction based on HTML. A server would distribute HTML pages to a browser that would allow

the user to interact with what they saw. Early browsers provided only “click to follow a link” style of interaction. Early servers mapped URLs directly to some part of the local file system. However, two innovations occurred simultaneously. The first was the inclusion of <form>s in HTML, which would add user inputs to the URL and the second was the addition of CGI (Common Gateway Interface) to servers. CGI allows web pages to be generated by programs rather than simply service the contents of the file system. This created a new model of distributed interaction. Server-based applications implemented using CGI would interact with users by generating interactive page descriptions for a standard browser. This allowed applications to distribute their interfaces without needing to install special software on the client. The first truly world-wide distribution of interactivity was born.

A careful look at the forms model of HTML interaction reveals that the style of interaction is identical to the old IBM 3270. Pages of information are sent to the client, the user fills in the changeable parts and the changes are returned to the server. The only real interactive innovations are that the client is software rather than hardware-based, which greatly extends its use and that the formatting/graphics are much more pleasant than the old 3270 block text screens.

The next interactive innovation was the Java Applet. Using this mechanism, interactive programs could be embedded in web pages and downloaded to the client workstation. This achieved the NeWS vision of downloadable interaction. By the time Java appeared in web browsers personal computers were fast enough to handle the graphical displays. Java also is a modern object-oriented language rather than a machine-code replacement such as Forth. Consequently development of applets was much easier.

The most recent development is AJAX or Asynchronous JavaScript And XML. JavaScript was introduced into HTML as a smaller language that could handle simple dynamic formatting of web page content. JavaScript is tied to Java in name only. Its most notable characteristic is direct access to the Domain Object Model (DOM) which is a tree representation of the HTML page. Using this access JavaScript code can dynamically modify the HTML being presented to the user. Using event handlers on a variety of HTML tags, JavaScript can embed a high degree of interactivity directly into the page going far beyond the 3270 model of forms interaction.

Model Semantics Distribution

The next approach for distributing interaction is to place the network between the View/Controller and the Model as shown in figure 17.3. There are many variations on this approach. One such strategy is to provide each interactive client with the application code and then have the server function as a synchronizer of the various models. Because each client has a copy of the application virtually all of the interaction takes place in the client. This has excellent local performance. This technique is rarely used for distribution of server interfaces to clients because it requires installation of the application on all possible client machines. It is most often used for multiple users to collaborate with each other. Such users are already likely to be using the same application and this facilitates their collaboration.

One such architecture uses the Command objects defined in chapter 16. Before a command is executed by invoking doIt(), the object is serialized (converted to a byte stream) and sent to all of the other collaborators. The application for each user takes the received command object and invokes doIt(). The model in each case cannot distinguish between command objects received from the controller and those received over the network. The one constraint that this imposes on command objects is that their references to the model must by symbolic rather than by pointers. Obviously a pointer on one machine will not be valid on another. The same techniques from chapter 16 that were used by command-object scripting systems will suffice for this purpose.

A related approach was pioneered by Greenberg's GroupKit[5]. GroupKit is based on TCL[6] and TK. The TK graphical toolkit interfaces with its model by means of the TCL scripting language. GroupKit handles much of its distribution of the interface by forwarding the TCL scripts generated by the user interface to all participants in the collaboration. This has an effect similar to the distribution of command objects. In fact command objects that have been adapted for scripting could use their scripting facilities as their mechanism for serializing and executing distributed commands.

Synchronization

One of the problems with the command or script distribution approach is synchronization of the commands. Let us take the case of two users editing the string "Hello World". User A has positioned the editing cursor just after the "H" and user B has positioned the cursor just after the "W". Suppose both of them hit backspace at the same time. User A's application sends a command to delete the

first character of the string. At the same time user B's application sends a command to delete the seventh character. When user A receives user B's command it will delete the seventh character giving the result "ello Wrld" because the seventh character (after deleting "H") is the "o" in World. User B will end up with the right string "ello orld" because its deletion did not confuse the indices of the command from user A. There are a huge variety of such problems that can arise from order confusion in the command stream.

One way to resolve this is to create a collaboration server that manages the ordering of commands. The server's job is to define the command order to which all clients must conform. This means that eventually every machine will have an identical command order and thus an identical model. The server defines this order by giving every command that it receives a sequence number. Each client retains the sequence number of the last confirmed command that it received from the server.

When a client performs a command it assigns that command the next sequence number and sends it to the server. The client also retains the command in a history list just in case the command is rejected. If the server agrees that the received command's sequence number is the next one in order then it confirms the command to the client that created it, adds the command to its server history, distributes the command to all other collaborators and increments its own command sequence number. If the server already has a command with that sequence number it rejects the command and notifies the client that created it.

Clients receive commands, confirmations and rejections from the server. If the client receives a command that matches its next sequence number it performs the command locally and increments its sequence number. If a client receives a confirmation it can discard the command or save it for undo purposes. Either way the client knows that the command was accepted by the server. If the client receives a command rejection from the server, the client undoes all saved commands up to and including the command with the matching sequence number.

This synchronization mechanism allows a client to optimistically move ahead with its interface processing assuming that the server will eventually catch up. If collaborators are talking with each other the likelihood that command conflicts will occur is low and this optimistic approach will be correct. By being optimistic the interactivity of the client is not impaired by network latency. The undo capability associated with command objects allows the server to help overly optimistic clients bring themselves into line.

There is one other minor detail to this strategy. With each command the client must send the sequence number of the last confirmed or forwarded command that it received. This will allow the server to know how many commands will need to be discarded as overly optimistic if a command rejection occurs.

The developers of GroupKit determined that social mechanisms and good design of the command set made synchronization errors extremely rare or non-existent. They therefore, did not use any synchronization control and relied upon the users to resolve any problems in the unlikely event that they occurred.

Data Layer Distribution

The last approach shown in figure 17.3 is distribution at the data layer. This approach has many of the advantages of distribution at the model interface layer. By distributing in the model, latency problems for tight interactions are reduced. By distributing at the data layer, some of the synchronization issues are simplified. Suppose, for example, that a data model consisted of numbers, strings, records, arrays and trees. This is a small number of concepts that spans a wide number of applications. The protocol for distribution can be uniform across all applications. When one user makes changes to the model, the changes are distributed to the other clients. The other clients modify their models and implicitly update their presentations in response.

This approach was used in the XWeb[7] system. XWeb extended HTTP by adding the, CHANGE, SUBSCRIBE and UNSUBSCRIBE commands to the standard GET and POST commands. The shared XWeb data model was XML. As with the WWW, XML was a communication façade over whatever underlying code was associated with a URL. A client would download XWeb information and make it available to the user interactively. As the user interacted with the client CHANGE commands were sent back to the site to modify the data. Clients would also SUBSCRIBE to sites with which they were interacting. When another user would use the same site, CHANGE commands would be forwarded to other clients so that they could update their presentation. By building the distribution around an abstract data model, a variety of applications could share the mechanisms without reimplementing them.

XWeb also supported a variety of clients for interacting with XWeb data models. Each client was designed for a particular combination of interactive devices. Thus the form of the interaction was tailored to the particular devices available. Clients were built for desktops, tablet pens, speech, laser pointers and

whiteboard pens. XWeb interfaces are highly adaptive to the particular devices and situations in ways that are not possible in the other collaborative architectures.

Summary

Distributing the interface allows users to work on systems that are remote from where applications are running. Distribution also supports collaboration among users on the same work. There are a variety of points within the application architecture where the distribution mechanism can be placed. Each such location supports a different collection of advantages and disadvantages.

---

[1] Newman, W. F. and Sproull, R. F. *Principles of Interactive Computer Graphics (2nd Ed.)*. McGraw-Hill, Inc. (1979)

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

[3] Adobe Systems Inc, *PostScript Language Reference (3rd Ed.)*, Addison-Wesley Longman Publishing Co. (1999).

[4] Bell, J. R. "Threaded code". *Commun. ACM* 16, 6 (Jun. 1973), pp 370-372.

[5] Roseman, M. and Greenberg, S., "Building real-time groupware with GroupKit, a groupware toolkit." *ACM Trans. Comput.-Hum. Interact.* 3, 1 (Mar. 1996), 66-106

[6] Ousterhout, J. K., *Tcl and the Tk Toolkit*, Addision-Wesley, (1994).

[7] Olsen, D. R., Jefferies, S., Nielsen, T., Moyes, W., and Fredrickson, P. "Cross-modal interaction using XWeb." *User interface Software and Technology* (*UIST '00)*. ACM Press, New York, NY, (2002) 191-200.