13

Applet Ensemble: The Opening

No plan of operation can extend with any prospect of certainty, beyond the first clash with the hostile main force. Only a layman can pretend to trace throughout the course of a campaign the prosecution of a rigid plan, arranged beforehand in all its details and adhered to the last. All successive acts of war are therefore not pre-meditated executions but spontaneous acts guided by military tactics.
—Field Marshal Helmuth von Moltke, "the Elder"

This chapter describes the initiating design question. It identifies a main design thread, along with a more promising but speculative contingency plan. The case study describes an attempt to demonstrate the viability of this contingency plan.

13.1 Where are We?

The close of Chapter 12 posed a number of questions about how to transform the unstructured manifest depicted in Figure 12-3 into something more tangible. At root, each of these questions tries to clarify the implications of a number of prior technology commitments.

During the earliest stage of the design, two overall approaches surfaced. The first approach used HTTP server scripting to produce dynamic Web content and to act as a gateway interface between browsers and back-end DIRS services. The second approach used Java applets running in browsers to communicate with DIRS services. Each approach had its strengths and weaknesses:

- The HTTP server scripting approach promised to be easy to implement, at least in the early stages. On the other hand, there were concerns that the approach would lock the project into a proprietary scripting language. This was dangerous considering how fast that segment of the technology market was changing. There were also worries about the long-term maintainability of this approach.

- The applet approach promised to be a flexible way to deliver applications to geographically distributed users. This design also made fewer commitments to vendor-specific features of either Web browsers or HTTP servers. On the other hand, there were serious concerns about the performance and security attributes of this approach.

After much gnashing of teeth, the project architect committed to the server-side approach, while initiating exploration of the feasibility of the applet approach. If its feasibility could be demonstrated before the server-side approach had progressed too far, a switch-over could be effected without impact to the overall project schedule. There was, after all, considerably more development on other aspects of DIRS. The case study begins in earnest from the situation depicted in Figure 13-1. The main design branch was the Server-Side JavaScript (SSJS) ensemble, while the main contingency was the applet ensemble.

13.2 Risk Analysis

Our exploration of the feasibility of the applet ensemble initiates an R^3 discovery cycle (Risk analysis, Realize model problem, Repair). At this point, so little was known that almost everything presented risk. Would the applet take too long to download? Would it introduce security risks? Could a solution be implemented that worked on both Microsoft and Netscape browsers? How would images be transferred from the image store to the applet? In the aggregate, these unknowns presented an unacceptable level of risk. It would have been impractical to attempt to answer all of the above questions at once. Besides, there were many more questions, as well. To begin, we posed the following design question: Does there exist an applet ensemble that works with Netscape and Microsoft browsers, and that allows users to connect to DIRS and to download an image into a third-party image viewing component?

This question is interesting for the issues it does not address—security and performance. In fact, it only asks for a demonstration of the most basic form of

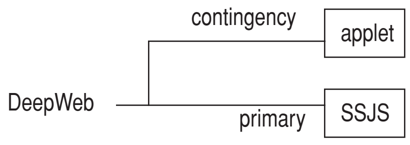

Figure 13-1 Main design path and contingency plan.

feasibility: an existence proof that a collection of components can, in fact, be integrated to achieve a modest objective. With the above question in mind, we moved on to realize a model problem (the second R in R^3).

13.3 Model Problem

We decided to define and implement a use case for retrieving an image from the repository. We did this for several reasons. First, retrieving images is a key capability of the system. Second, image retrieval involves components in all three tiers of the DIRS architecture. Third, the problem was simple to model and communicate. However, there were also several drawbacks. This problem did not, for example, model the transactional properties of the system, but the model problem was deemed sufficient to answer the motivating question, and any further effort would have been inappropriate.

Implementing the image retrieval model solution required that we consider components in all three tiers of the architecture. The middle tier included a component representing the business rule interpreter (BRI). The BRI encapsulated DIRS business rules, supported connections with multiple, simultaneous clients, and coordinated data flow throughout the system. The back-end server contained a component representing the storage manager (SM). This component maintained the actual image files. The client, middle tier, and bottom tiers of the architecture were all hosted on separate platforms. A high bandwidth LAN connected the client to middle tier, although we also considered a low bandwidth WAN. The middle tier to lower tier connection is guaranteed to be a high-bandwidth LAN connection.

After brief discussion, three major options presented themselves, as depicted in Figure 13-2. Each model solution provided an alternate approach for implementing control flow and data flow. In all three, control flow is via the IIOP connections. Our interest in data flow is principally with image retrieval, as movement of images is essential to both the model problem and the actual DIRS system. Additionally, the size of image files stored in DIRS could be extremely large—on the magnitude of 10–200 megabytes. As a result, handling these images is a critical, if not overriding consideration. Each design alternative uses the same components, but in a different way. The first ensemble transfers images over IIOP connections between the client and the BRI, and between the BRI and the SM. All communication in this solution travelled through the BRI. There was no direct connection between the client and the SM. The second ensemble uses HTTP to transfer image files directly from the storage manager to the client. The third ensemble transfers images directly from the storage manager to the client over IIOP.

The indirect IIOP ensemble (1) had deficiencies that were readily apparent: it required images to be transferred twice—once between the SM and the BRI

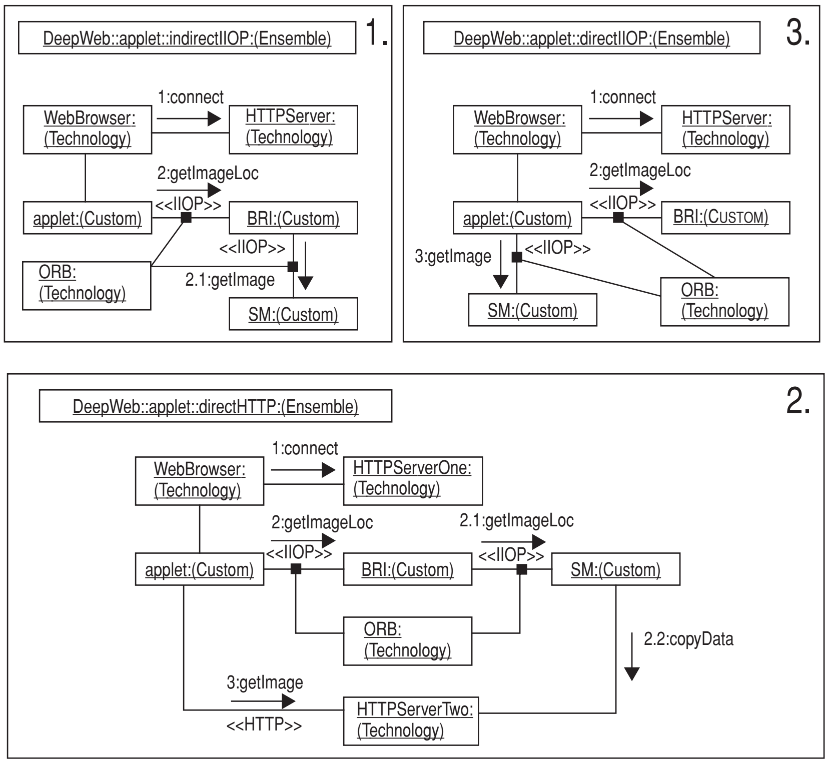

Figure 13-2 Three ensembles.

and again from the BRI to the client. This was unacceptable, due to the potential size of these images and the performance overhead of transferring images a second time. On the other hand, recently we had developed an application using this ensemble to transfer small images. Thus, we were confident that we could produce another implementation. This ensemble could become an interim solution should one of the remaining ensembles appear to be promising but only conditionally feasible. In a sense, then, the indirect IIOP option was a contingency within a contingency.

Before proceeding, we worked with the architect to develop evaluation criteria. That is, how would we know that an ensemble is feasible? We agreed that retrieval of an image from an image store would constitute success, at least at this stage of the design. Then, again working with the architect, we settled on implementation constraints, mainly concerning selection of components.

Java applets were developed using version 1.1.3 of the JDK and OrbixWeb 2.0.1 for communicating with CORBA servers on the middle tier and back-end servers. We often refer to Java applets that communicate with CORBA servers as *orblets*. These orblets were to run on versions 3.0 and 4.0 of the Netscape Browser, and versions 3.0 and 4.0 of the Internet Explorer Browser. The client platform operating system was Windows NT 4.0. The BRI and SM servers were coded in C++, compiled with the SPARCompiler C++, and run on Solaris 2.5. Implementing C++ required installing an additional compiler and dusting off our C++ manuals. We felt this small, but additional, effort enabled us to more closely model the DIRS system. (After all, when looking for a reliable trail guide it is best to find someone who has been down the same trail, and as recently as possible.) Both the BRI and SM used Orbix 2.2 to communicate with each other and the client. Version 2.0.1 of the Netscape FastTrack Server was the HTTP server.

Some of these component selections were consistent with those made for the main design thread (see Figure 13-1), while others were intentionally varied to build team competence in the event that a switch-over would be required.

13.4 Model Solutions

We decided to skip implementing the indirect IIOP ensemble. Prior experience convinced us that this design could be implemented. Therefore we didn't feel this effort would be justified. Instead, we focused our attention on the two remaining ensembles.

MODEL SOLUTION WITH DIRECT HTTP ENSEMBLE

Figure 13-3 shows a deployment diagram for a model solution developed with the direct HTTP transfer ensemble.

The UML sequence diagram shown in Figure 13-4 provides a detailed description of the interactions among components in the solution. The end user first opens the main DIRS HTML page using a browser. The browser contacts the HTTP server that downloads the page content including a Java applet. The applet is loaded into the JVM in the browser and passes control to the applet. The applet uses the CORBA bind operation to bind to the BRI, which in turn binds to the SM. Control returns to the end user who can now request the location of the image by interacting directly with the applet running within the browser. The applet contacts the BRI with this request, which is forwarded to the SM running on the back-end server. The SM returns a URL for the image to the BRI that constructs a new HTML page containing a link to this URL. The URL for this newly constructed page is returned to the applet. The applet sends a request to the browser to display the new HTML page (effectively terminating the applet's existence). The new HTML page containing a link to the image on the storage manager is displayed in the browser. The end user can now select the link to view the image.

Web Browser Evaluation and Risk/Misfit

One of our early interactions with the DIRS design team centered on Web browser evaluation. Prior to our involvement, the design team had a long list of what were deemed essential browser features. Unfortunately, there were no browsers available on the market that possessed all of these features, nor was it clear which features were essential and which were merely desirable. There was considerable controversy, for example, over whether the browser required an on-board JVM of version 1.1 or better, and there were the usual squabbles over whether Netscape Navigator or Microsoft Explorer or both should be supported.

To help matters along we introduced the use of Risk/Misfit, and began by defining the objective of the evaluation. We defined two objectives for the evaluation:

- The evaluation criteria should be satisfiable by Explorer and Navigator, as these possessed (at that time) over 90% of the browser market.
- The evaluation criteria would identify only those browser features that the DIRS design would depend on; DIRS would depend on no other browser features.

Each browser feature that had been identified as "essential" had to be justified in terms of the design risk that would arise without that feature. The aim of this was twofold. First, it forced feature advocates to use constructive rather than categorical arguments to justify their advocacy. Second, it surfaced and documented design assumptions that were held by different members of the design team. Sometimes these assumptions were in conflict, and clarifying the risk statement helped to resolve these conflicts. One other benefit of this approach was that, having identified design risks associated with the absence of features, the design team was provoked into imagining possible mitigators to the risk. In effect, these mitigators expressed design options for DIRS that did not depend on so-called essential features. In fact, the design team discovered that the only truly essential features were those for which no risk mitigation (called "repair" in Risk/Misfit) could be conceived. Thus, building Risk/Misfit evaluation criteria had the effect of identifying those features that defined minimum satisfaction criteria for browsers to be considered "feasible," while all other features expressed mere preference.

A fragment of the resulting evaluation criteria is shown in Table 13-1. Note that we had adapted Risk/Misfit for the purpose just described, and so, at this point, repair cost and residual risk were not yet a concern. Observe, though, that the criteria include something not discussed in the Risk/Misfit chapter: a specification of how a misfit will be detected. Such modifications to Risk/Misfit or any other technique described in this book are encouraged, provided they are sound, in addition to being expedient.

—shissam

Table 13-1 Web Browser

Support Feature	Risk	Mitigator	Assessment Technique	Feature Description
HTTP/1.0 persistent connections	Performance overhead	Use HTTP/1.0 and "cookies"	Product literature	The server can keep a connection with an HTTP/1.1 client open for more than a single request.
Plug-in support for editors	Loss of support for 45 million stored images	None	Model problem	Specify other data and object formats that are supported by the browser.
Object Signing	Loss of authentication and integrity checking	Use 3rd-party security COTS and added integration effort	Model problem	Support of object (applet or ActiveX) signatures with manual and/or automatic acceptance/denial of download of object based on signature.
SSL v3.0	Unencrypted data can be disclosed; also no indication of loss of data integrity	Revert to SSL v2.0	Model problem	The server can communicate using the SSL version 3 protocol.

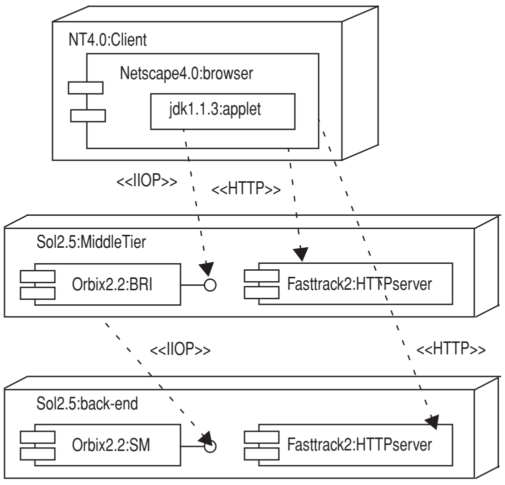

Figure 13-3 Deployment view of Direct HTTP Ensemble.

The implementation went well until we needed to generate a request to the browser to open the new page. At this time, we did not have any experience communicating from the applet to a browser. Luckily, we acquired a copy of the book *Java Network Programming* by Rusty Harold [Harold 97] and learned about the `showDocument()` method in the `java.applet.AppletContext` class. On page 131 of this book we discovered that:

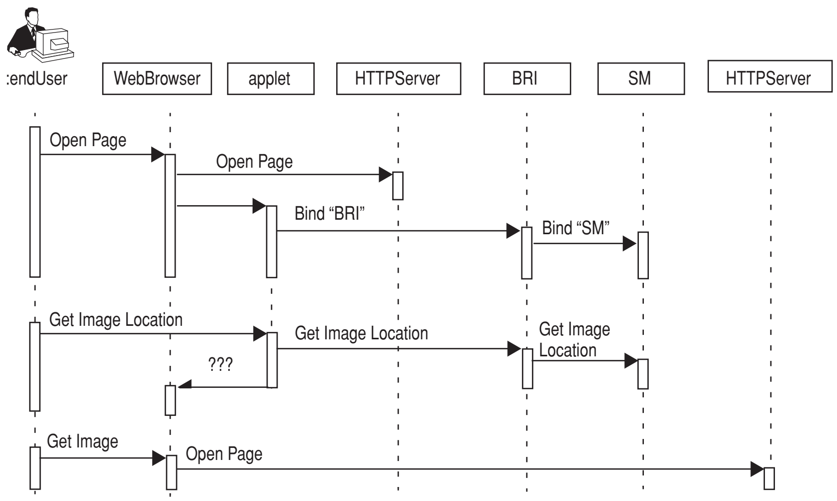

Figure 13-4 Direct HTTP Ensemble sequence diagram.

> This method shows the document at `URL u` in the `AppletContext's` window. It is not supported by all web browsers and applet viewers, but it is supported by Netscape and HotJava.

We added a call to the `showDocument()` method from the applet to display the HTML page in the browser. As promised, this worked with Netscape Navigator, but it was not known to work with Internet Explorer. By using Netscape Navigator to implement the browser component, we had proved, at a minimum, that one implementation of the ensemble was possible. However, we had not satisfied the evaluation criterion that the solution work with both Navigator and Explorer. This was an important criterion, since the DIRS user community used both browsers. After implementing the model solution, we modified our blackboard as shown in Figure 13-5.

The first and most obvious change is that technologies have been replaced with components. CORBA has been replaced with Orbix 2.2, used to enable communication with the legacy C++ servers and OrbixWeb 2.0.1, used to enable communication with the Java applets within the browser. The interaction between applet and BRI is associated with both ORBs since it involves different implementations of the CORBA standard, albeit by the same vendor. The diagram also includes a credential for the show document method. Its equivalent on Explorer was only a postulate.[1]

[1] A later version of Microsoft IE did, in fact, support the show document method.

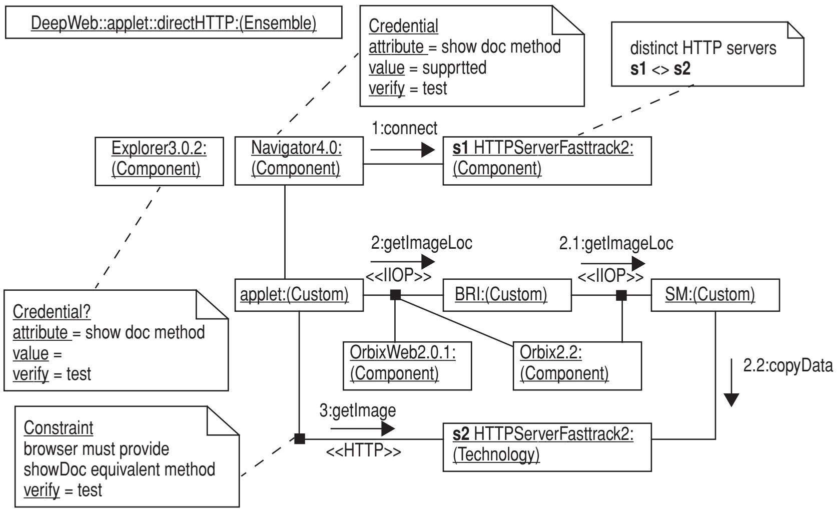
DeepWeb::applet::directHTTP:(Ensemble)
Credential
attribute = show doc method
value = supprtted
verify = test
distinct HTTP servers
s1 <> s2
Explorer3.0.2: (Component)
Navigator4.0: (Component)
1:connect
s1 HTTPServerFasttrack2: (Component)
2:getImageLoc
<<IIOP>>
2.1:getImageLoc
<<IIOP>>
applet:(Custom)
BRI:(Custom)
SM:(Custom)
Credential?
attribute = show doc method
value =
verify = test
OrbixWeb2.0.1: (Component)
Orbix2.2: (Component)
2.2:copyData
Constraint
browser must provide showDoc equivalent method
verify = test
3:getImage
<<HTTP>>
s2 HTTPServerFasttrack2: (Technology)

Figure 13-5 Revised Direct HTTP.

Without the show document method we would have to abandon this ensemble. When dealing with emerging technologies, the critical role played by low-level component features may not be apparent until implementation time, which is what makes early prototyping so important. In our experience, model problems are the most efficient way of implementing these prototypes.

MODEL SOLUTION WITH DIRECT IIOP ENSEMBLE

The second model solution we implemented used the direct IIOP ensemble. This model solution used the same components employed in the direct HTTP transfer solution except it did not require an HTTP server on the back-end. We originally intended to deploy this model solution as depicted in Figure 13-6.

We did have some apprehensions about implementing this ensemble. The model solution requires that a Java applet, running within a browser on the client platform, directly calls remote methods on both the BRI server running in the middle tier and on the SM server running on the back-end. However, for reasons of security, the JVMs within both the Netscape and IE browsers prevent an applet from connecting to a second machine. To circumvent this restriction, a little creativity was required. We created a second "helper" applet that is loaded with the HTML page generated by the BRI and served up from a HTTP server on the back-end platform. So we needed a second HTTP server after all! Figure 13-7 shows the sequence diagram for this model solution. (The second HTTP server is not shown, to make the diagram easier to read.)

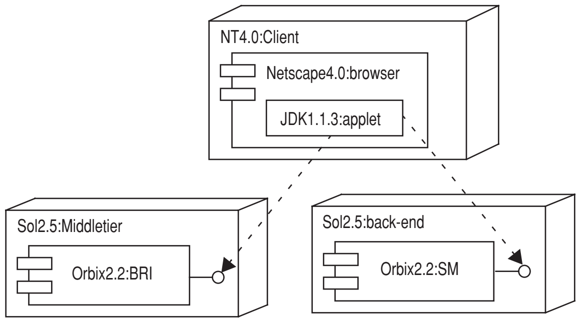

Figure 13-6 Deployment view of Direct IIOP Ensemble.

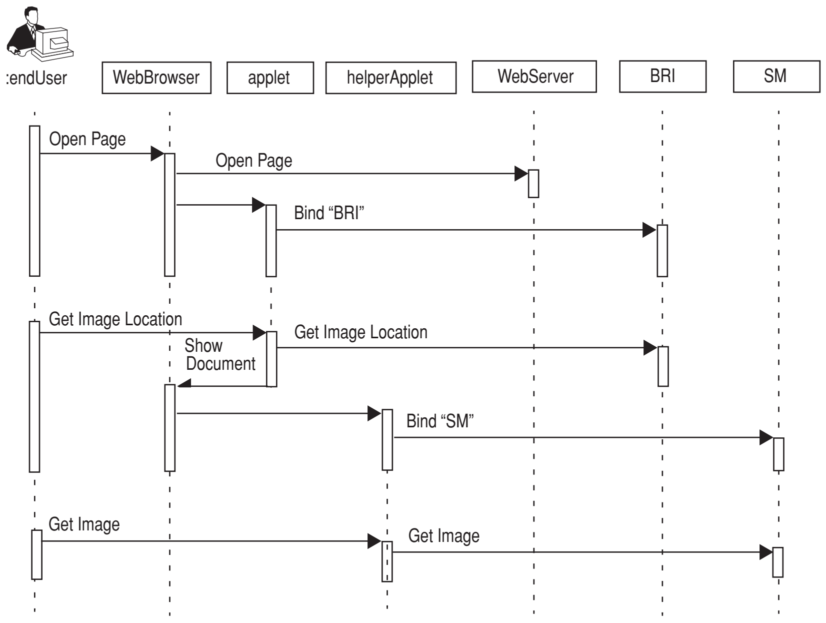

Figure 13-7 Direct IIOP transfer sequence diagram.

As in the direct HTTP transfer ensemble, the end user opens the main DIRS page with the browser. The browser contacts the server that downloads the page content including a Java applet. The applet is loaded into the browser JVM and control is passed to the applet. The applet uses the CORBA bind operation to bind to the BRI. Unlike the direct HTTP transfer solution, the BRI no longer binds directly to the SM. Control is returned to the end user who sends a request for the image location to the applet. The applet forwards this request to the BRI that constructs an HTML page containing a second, helper, applet. The location of the image is included in the new page as a parameter to the helper applet. When control is returned to the applet, it calls the show document method as in the direct HTTP transfer solution. The new page, including the helper applet, is loaded by the browser and control is passed to the new helper applet that binds to the storage manager. The end user now sends a request to the helper applet to retrieve the image, which is forwarded to the SM. The SM converts the requested image into a sequence of bytes, or *octets* in CORBA terminology, and returns it via IIOP to the helper applet running within the browser.

Figure 13-8 shows the revised deployment diagram for this model solution. This is a good example of how actual experience gained from implementing model problems can alter a design. As suggested by von Moltke, in the epigraph that opens this chapter, it is impossible to predict the utility of a plan or design beyond the initial phases. This makes it incumbent on the designer to insist on increasingly stringent proofs of design feasibility.

EXTENDING THE SANDBOX

We succeeded in transferring an image from the storage manager to the client in the direct IIOP transfer ensemble, and, by logical extension, in the indirect IIOP ensemble as well. However, we have come short of a full solution in both cases. The image data is stored as a byte array within the JVM and cannot be directly displayed. We still need to get the image data out of the JVM and into the third-party image viewing component. Although other commercial image viewers were available, they did not support some of the nonstandard image formats used by DIRS. The image viewer is invoked on a command with an argument specifying

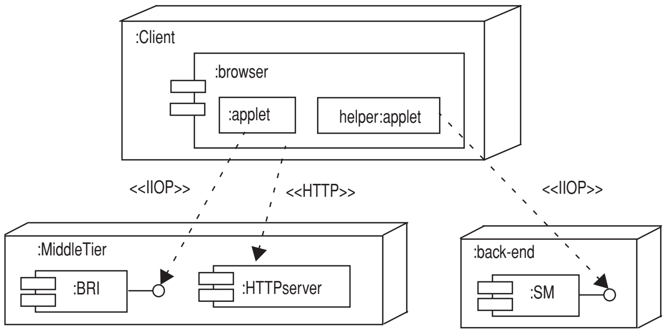

Figure 13-8 Revised direct HTTP transfer abstract model solution.

the location of the file containing the image to be viewed. To invoke this image viewer from our thin client we had to:

1. Write the image data to a temporary file on the client platform.
2. Launch an external application on the client platform.

Unfortunately, these are both privileged operations under Java. To prevent downloaded applets from malicious behavior, version 1.1 of the JDK, operating in a browser environment, enforces a security mechanism known as the *sandbox*. An applet running in a sandbox is restricted from performing operations that might compromise the security of the platform. The sandbox, for example, restricts reading or writing files on the local file system.

The options available in 1997 for allowing applets to operate outside of the browser sandbox were extremely limited. Netscape provided the Netscape capabilities classes that allowed an applet to operate outside of the sandbox. The capabilities classes added facilities to refine the control provided by the standard Java security manager class. These classes could then be used to exercise fine-grained control over an applet's activities outside of the sandbox.

A full discussion of the use of digital certificates and security is contained in the following chapter, but a brief introduction is required here. Access control decisions boil down to who is allowed to do what. In the capabilities model, a principal represents the "who," a target represents the "what," and the privileges associated with a principal represent the authorization for a principal to access a specific target. Using digital certificates, the *principal* is represented by a signing certificate while the *target* is one or more system resources, such as files stored on a local disk. The capabilities classes make it possible to determine whether any given principal (that is, signing certificate) is allowed to access the local system resources represented by a given target. The answer is expressed by a *privilege*, which states whether access is allowed and, if so, for how long.

We extended our IIOP transfer solutions to use the Netscape capabilities classes to write image data out to a temporary file and invoke an external application to view them. Additionally, we used the capabilities classes to circumvent the restriction that a Java client could only communicate with the host from which it was downloaded. The capability of connecting to multiple hosts also eliminated the need for an additional helper applet to communicate with the SM.

Unfortunately, the capability classes also had drawbacks. Since they are proprietary, we could not use them in non-Netscape browsers, for example, Microsoft Internet Explorer. The revised blackboard for the Direct IIOP ensemble, illustrated in Figure 13-9, shows that applets must be digitally signed (refer back to Figure 5-6 for the details of the signed applet ensemble). It also clearly indicates, via a postulated credential on Explorer 3.0.2, that it is uncertain whether Explorer provides an equivalent feature. As will be seen when we resume the case study narrative in Chapter 15, this postulate introduces a whole range of rather complex issues pertaining to interoperability of digital certificates.

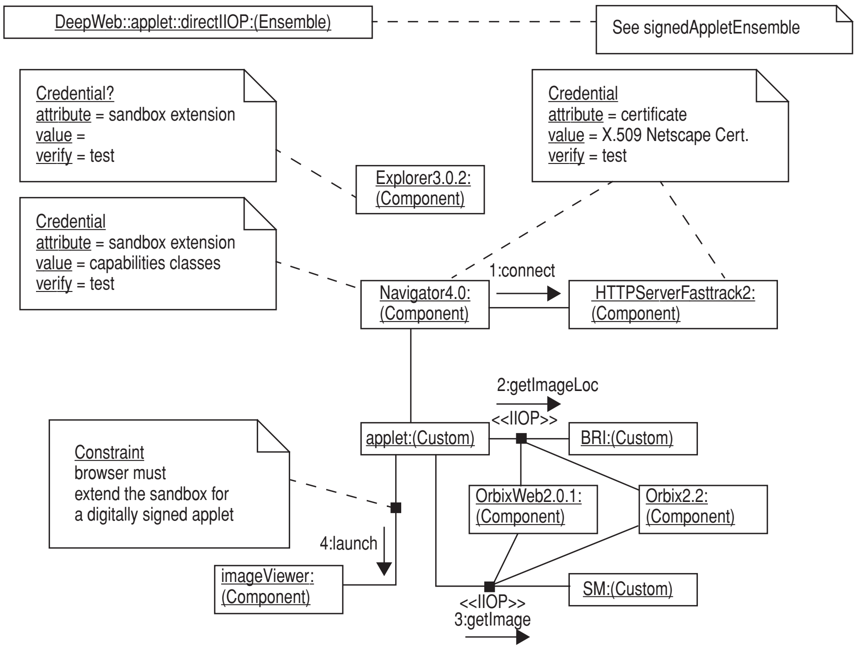

Figure 13-9 Revised Direct IIOP.

13.5 Evaluation

We had completed two model solutions but had obtained decidedly mixed results. On the one hand, the Direct HTTP Ensemble proved feasible (we had, by this time, verified that Explorer supported the show document method). This was good news except for the fact that this ensemble, shown in Figure 13-4, is too complex for the simple function it performs. There also remained questions about how to manage error propagation considering the fact that distinct control and data channels are used. Another detail concerned finalization of the retrieval scenario. Specifically, data on the second Web server should be deleted immediately upon retrieval; this involves one more interaction, and still more complexity.

On the other hand, the Direct IIOP solution was simple and elegant. Unfortunately, our model solution relied on proprietary Netscape interfaces. Since the model solution did not work with both Navigator and Internet Explorer, this ensemble was, at best, only conditionally feasible. It had failed to satisfy the evaluation criteria established for the model problem. However, we already had in mind several repair strategies and were not ready to concede defeat. We had heard rumors of a product from JavaSoft called Java Protection Domains that promised to be a portable alternative to Netscape's Capability Classes, and we were willing to wait on this market event while continuing to explore the ramifications of this ensemble. (Persistence is an important quality when building component-based systems. Optimism also helps.)

Figure 13-10 depicts the situation in terms of contingency management. Work was proceeding on the main design option, while more work remained to prove that the applet ensemble was a viable alternative. The direct HTTP ensemble was feasible, but ungainly. We were suspicious that anything that ugly could be correct. Therefore, we considered its feasibility to be unproven. The direct IIOP ensemble was the favorite, but it was infeasible without repair. Java Protection Domains, we thought, would repair the Navigator dependency, and accordingly we decided to "wait" on its release from Sun Microsystems. This did not mean, however, that further repair options were foreclosed.

Figure 13-11 summarizes what we discovered in a fragment of the manifests associated with the direct IIOP and direct HTTP ensembles. Roll-up diagrams like this are exceedingly useful as the number of components and potential ensembles explodes.

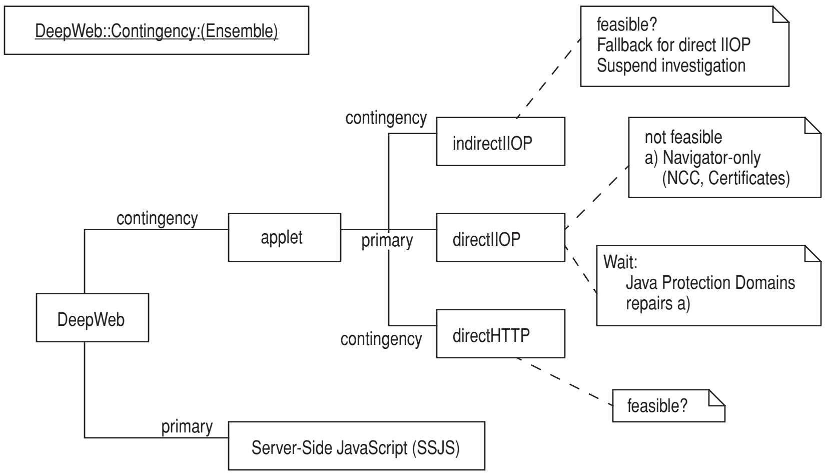

Figure 13-10 DeepWeb contingency.

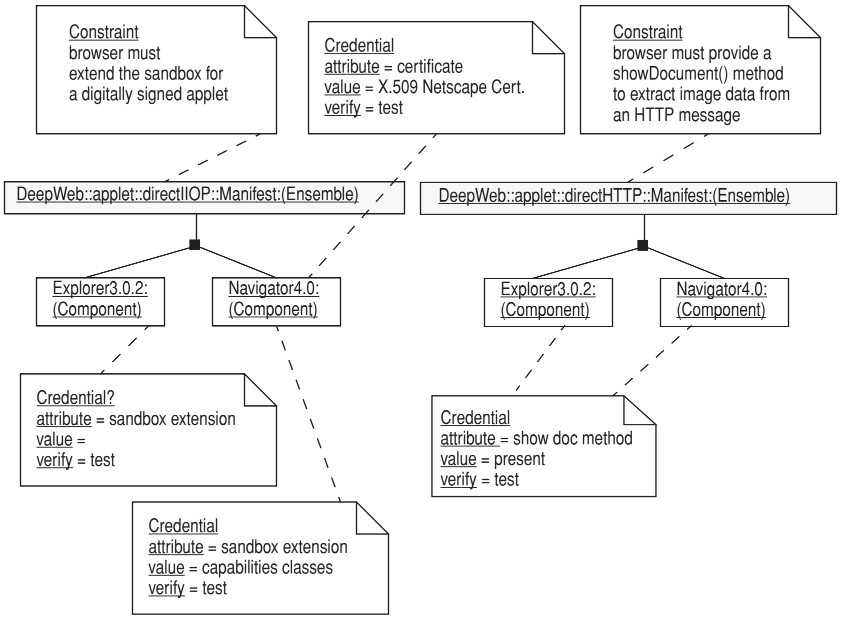
Constraint
browser must extend the sandbox for a digitally signed applet
Credential
attribute = certificate
value = X.509 Netscape Cert.
verify = test
Constraint
browser must provide a showDocument() method to extract image data from an HTTP message
DeepWeb::applet::directIIOP::Manifest:(Ensemble)
DeepWeb::applet::directHTTP::Manifest:(Ensemble)
Explorer3.0.2: (Component)
Navigator4.0: (Component)
Explorer3.0.2: (Component)
Navigator4.0: (Component)
Credential?
attribute = sandbox extension
value =
verify = test
Credential
attribute = show doc method
value = present
verify = test
Credential
attribute = sandbox extension
value = capabilities classes
verify = test

Figure 13-11 Manifest after model solutions.

13.6 Summary

This chapter illustrates our approach to the design of component-based systems. The design space is partitioned into branches corresponding to a main design option (the trunk) and one or more contingencies. Each contingency may itself be partitioned, recursively, with effort allocated to each design option. Model problems are used to guide the exploration of contingencies. In this chapter, the design question for one model problem was defined: were there any applet ensembles that would work with both Internet Explorer and Netscape Navigator? Three ensembles were specified, two of which were realized as model solutions.

The model problem and their solutions significantly extended our knowledge of the design space. We discovered problems that revised our initial ideas about how these ensembles worked. However, as is the norm for any field of exploration, we were left with more questions. Was there a mechanism that can be used to provide fine-grained access outside of the sandbox that would work with both Netscape Navigator and Microsoft Internet Explorer? Would the transfer of images over IIOP be the most efficient mechanism? We explore these questions, along with others, in the remaining chapters of this part of the book.

13.7 Discussion Questions

1. Direct transfer of images over HTTP appeared to provide adequate performance. What are the arguments for and against this abstract model solution?

2. Late selection of products that implement technologies provides additional time for evaluation and experimentation through model problems. Early selection of a product helps to solidify the design and provide focus to the development. Discuss other advantages and disadvantages of these conflicting approaches. When is one approach better than the other?