Using Smart Clients to Build Scalable Services

Chad Yoshikawa, Brent Chun, Paul Eastham, Amin Vahdat,
Thomas Anderson, and David Culler
Computer Science Divisio
University of California, Berkeley

Using Smart Clients to Build Scalable Services

Chad Yoshikawa, Brent Chun, Paul Eastham,
Amin Vahdat, Thomas Anderson, and David Culler
Computer Science Division
University of California
Berkeley, CA 94720

Abstract

Individual machines are no longer sufficient to handle the offered load to many Internet sites. To use multiple machines for scalable performance, load balancing, fault transparency, and backward compatibility with URL naming must be addressed. A number of approaches have been developed to provide transparent access to multi-server Internet services including `HTTP` redirect, DNS aliasing, Magic Routers, and Active Networks. Recently however, portable Java code and lightly loaded client machines allow the migration of certain service functionality onto the client. In this paper, we argue that in many instances, a client-side approach to providing transparent access to Internet services provides increased flexibility and performance over the existing solutions. We describe the design and implementation of *Smart Clients* and show how our system can be used to provide transparent access to scalable and/or highly available network services, including prototypes for: `telnet`, `FTP`, and an Internet chat application.

1 Introduction

The explosive growth of the World Wide Web is straining the architecture of many Internet sites. Slow response times, network congestion, and "hot sites of the day" being overrun by millions of requests are fairly commonplace. These problems will only worsen as the Web continues to experience rapid growth. As a result, it has become increasingly important to design and implement network services, such as `HTTP`, `FTP`, and web searching services, to scale gracefully with offered load. Such scalable services must, at minimum, address the following issues:

- Incremental Scalability – If the offered load begins to exceed a service's hardware capacity, it should be a simple operation to add hardware resources to transparently increase system capacity. Further, a service should be able to recruit resources to handle peaks in the load. For example, while the US Geological Survey Web site (`http://quake.usgs.gov`) is normally quite responsive, it was left completely inaccessible immediately after a recent San Francisco Bay Area earthquake.
- Load Balancing – Load should be spread dynamically among server resources so that clients receive the best available quality of service.
- Fault Transparency – When possible, the service should remain available in the face of server and network upgrades or failures.
- Wide Area Service Topology – Individual servers comprising a service are increasingly distributed across the wide area [Net 1994, Dig 1995]. The server machines that comprise a network service should not be required to have a restricted or static topology. In other words, all servers should be allowed to arbitrarily migrate to other machines.
- Scalable Service To Legacy Servers – Adding scalability to existing network services such as `FTP`, `Telnet`, or `HTTP` should not require modifications to existing server code.

This work was supported in part by the Defense Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C-0014), the National Science Foundation (CDA 0401156), California MICRO, the AT&T Foundation, Digital Equipment Corporation, Exabyte, Hewlett Packard, Intel, IBM, Microsoft, Mitsubishi, Siemens Corporation, Sun Microsystems, and Xerox. Anderson was also supported by a National Science Foundation Presidential Faculty Fellowship. Yoshikawa is supported by a National Science Foundation Fellowship. The authors can be contacted at {`chad, bnc, eastham, vahdat, tea, culler`}`@cs.berkeley.edu`.

Unfortunately, providing these properties for network services while remaining compatible with the de facto URL (Uniform Resource Locator) naming scheme has proven difficult. URLs are by definition location dependent and hence are a single point of failure and congestion. A number of efforts address this limitation by hiding the physical location of a particular service behind a logical DNS hostname. Examples of such systems include `HTTP` redirect, DNS Aliasing, Failsafe TCP, Active Networks, and Magic Routers.

We argue that in many cases the client, rather than the server, is the right place to implement transparent access to network services. We will describe limitations associated with each of the above solutions and demonstrate how these limitations can be avoided by moving portions of server functionality onto the client machine. This approach offers the advantage of increased flexibility. For example, clients aware of the relative load on a number of `FTP` mirror sites can connect to the least loaded mirror to deliver the highest throughput to the end user. Ideally, the selection and connection process takes place without any intervention from the end user, unlike the Web today where users must choose among `FTP` mirror sites manually. Note that in this example, clients must take into account available network bandwidth to each mirror site as well as the relative load of the sites to receive optimal performance. Such flexibility would be difficult to provide with existing server-side solutions since individual servers may not have knowledge of mirror site group membership and client location.

The migration of service functionality onto client machines is enabled by two recent developments. Today, most popular Internet services, such as `FTP`, `HTTP`, and search engines are universally accessed through extensible Web browsers. This extensibility allows insertion of service-specific code onto client machines. In addition, the advent of Java [Gosling & McGilton 1995] allows such code to be easily distributed to multiple platforms. Next, network latency and bandwidth are increasingly the bottleneck to the performance delivered to clients. Thus, client processors can be left relatively idle. We will demonstrate that offloading service functionality onto these idle cycles can substantially improve the quality of service along the axis described above.

Motivated by the above observations, we describe the design and implementation of *Smart Clients* to support our argument for client-side location of code for scalability and transparency. The central idea behind Smart Clients is migrating server functionality to the client machine to improve the overall quality of service in the ways described above. This approach contrasts the traditional "thin-client" model where clients are responsible largely for displaying the results of server operations. While our approach is general, this paper concentrates on augmenting the client-side architecture to provide benefits such as fault transparency and load balancing to the end user.

The rest of this paper is organized as follows. Section 2 discusses existing solutions to providing scalable services. The limitations of the existing solutions motivates the Smart Client architecture, described in Section 3. Section 4 demonstrates the utility of the architecture by describing the implementation and performance of interfaces for `telnet`, `FTP`, and a scalable chat service. Section 5 evaluates our requirements above in the context of the Smart Client architecture. Section 6 describes related work, and Section 7 concludes.

2 Alternative Solutions

Existing architectures include DNS Aliasing [Brisco 1995, Katz et al. 1994], `HTTP` redirect [Berners-Lee 1995], Magic Routers [Anderson et al. 1996], failsafe TCP [Goldstein & Dale 1995], and Active Networks [Wetherall & Tennenhouse 1995]. Figure 1 describes how Smart Clients fits in the space of existing solutions. We will describe each of the existing solutions in turn leading to a description of the Smart Client architecture.

A number of Web servers use Domain Name Server (DNS) aliasing to distribute load across a number of machines cooperating to provide a service. A single logical hostname for the service is mapped onto multiple IP addresses, representing each of the physical machines comprising the service. When a client resolves a hostname, alternative IP addresses are provided in a round-robin fashion. DNS aliasing has been demonstrated to show relatively good load balancing, however the approach also has a number of disadvantages. First, random load balancing will not work as well for requests demonstrating wide variance in processing time. Second, DNS aliasing cannot account for geographic load balancing since DNS does not possess knowledge of client location/server capabilities.

On a client request, `HTTP` redirect allows a server to instruct the client to send the request to another location instead of returning the requested data. Thus, a server machine can perform load balancing among a number of slave machines. However, this approach has a number of limitations: latency to the client is doubled for small requests, a single point of failure is still present (if the machine serving redirects is un-

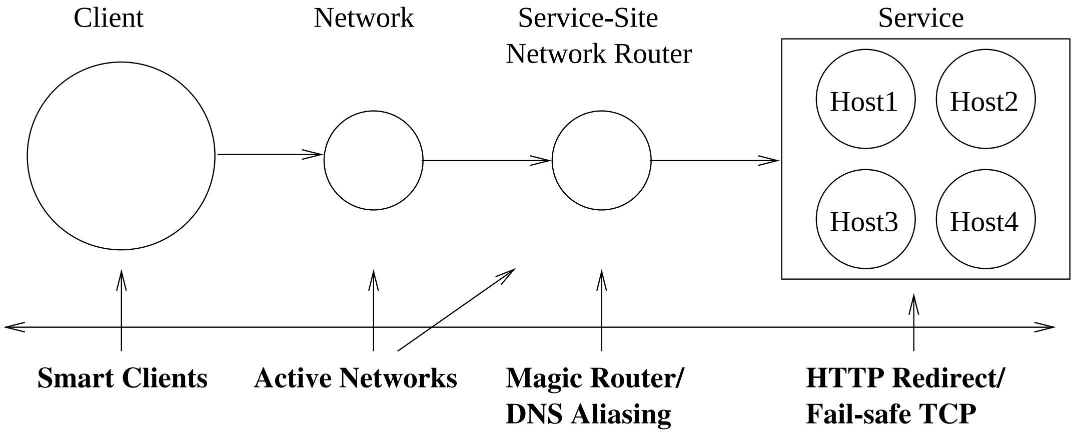

Figure 1: This figure describes the design space for providing transparent access to scalable network services. Transparency mechanisms can be implemented in a number of places, including the client, network, network routers, or at the service site.

available, the entire service appears unavailable), and servers can still be overloaded attempting to serve redirects. Further, this mechanism is currently only available for `HTTP`; it does not work with legacy services nor does it optimize wide-area access.

The Magic Router provides transparent access by placing a modified router on a separate subnet from machines implementing a service. The Magic Router inspects and possibly modifies all IP packets before routing the packets to their destination. Thus, it can perform load balancing and fault transparency by mapping a logical IP address to multiple server machines. If a packet is destined for the designated service IP address, the Magic Router can dynamically modify the packet to be sent to an alternative host. Unresolved questions with Magic Routers include how much load can be handled by the router machine before the dynamic redirection of the packets becomes the bottleneck (since it must process every packet destined for a particular subnet). Magic Routers also require a special network topology which may not be feasible in all situations. Finally, the Magic Router is not aware of the load metrics relevant to individual services, i.e. it would have to perform remappings based on a generic notion of load such as CPU utilization.

Fail-safe TCP replicates TCP state across two independent machines. On a server failure, the peer machine can transparently take over for the failed machine. In this fashion, fail-safe TCP provides fault transparency. However, it requires a dedicated backup machine to mirror the primary server, and it does not address the problem of the front-end becoming a bottleneck. Finally, both fail-safe TCP and Magic Routers are relatively heavy-weight solutions requiring extra hardware.

Proposals for Active Networks allow for computation to take place in network routers as packets are routed to their destination. This approach could potentially provide fault transparency and load balancing functionality inside of the routers. We believe Active Networks, if widely deployed, can provide a mechanism for implementing Smart Client functionality.

All of the above solutions provide a level of transparent access to network services with respect to load balancing and fault transparency. However, they are all limited by the fact that they are divorced from the characteristics and implementations of individual services. We observe that the greatest functionality and flexibility can often be provided by adding service-specific customization to the client, rather than service-independent functionality on the server.

3 Smart Client Architecture

In this section, we describe how the Smart Client architecture allows for the construction of scalable services. For the purposes of this paper, we assume the service is implemented by a number of peer servers, each capable of handling individual client requests[1]. The key idea behind Smart Clients is the migration of certain server functionality and state to the client machine. This approach provides a number of advantages: (i) offloading server load and decreasing imple-

[1]This assumption holds for many widely-used Internet services such as `HTTP`, `FTP`, and Web searching services.

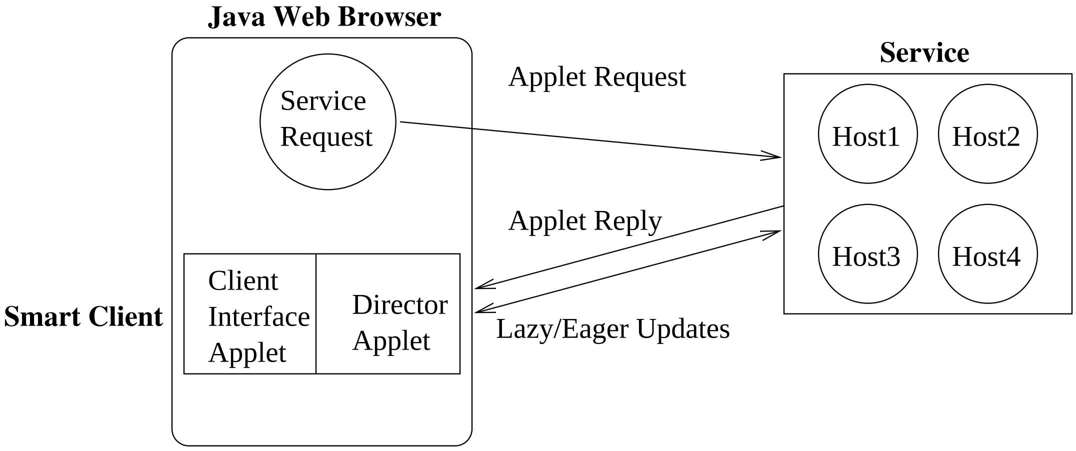

Figure 2: This Figure describes the Smart Client service access model. Two service-specific Java applets are supplied to mediate server access. The *client interface* applet provides the interface to the user and makes requests of the service. The *director* applet is responsible for providing transparency to the client applet; it makes server requests to the appropriate (e.g. least loaded) server, and updates its notion of server state.

mentation complexity, (ii) allowing clients to utilize multiple peer servers distributed across the wide area without the knowledge of individual servers, and (iii) improving the load distribution and fault transparency of the service as a whole.

When a user wishes to use a service, a bootstrapping mechanism is used to retrieve service-specific applets designed to access the service. Two cooperating applets, a *client interface applet* and a *director applet*, provide the interface and mask the details of contacting individual servers respectively. Client-side functionality is partitioned in this fashion to separate the service's interface design from the mechanisms necessary to deliver client requests to servers in a load-balanced, fault tolerant manner.

The client interface applet is responsible for accepting user input and packaging these requests to the director applet. The director applet encapsulates knowledge of the service member set and the service-specific meta-information allowing the director applet to send requests to the appropriate server. For every user request, the Smart Client uses the director applet to invoke a service-specific mechanism for determining the correct destination server for the request. Figure 3 shows the interaction of the two applets in a Java-enabled Web browser. A number of issues are associated with this approach: naming mechanisms for choosing among machines implementing a service, procedures for receiving updates with new information about a service (e.g., changes in load, or the availability of a new machine), and bootstrapping retrieval of the Smart Client applets. We will discuss each of these issues in turn leading to a description of the Smart Clients API.

3.1 Transparent Service Access

3.1.1 Load Balancing and Fault Transparency

We begin our discussion of the Smart Client architecture by describing the techniques used to provide load balanced and fault tolerant access to network services. Discussion of bootstrapping the retrieval of the Smart Client is deferred until Section 3.2. We assume that services accessed by Smart Clients are implemented by a number of peer servers. In other words, any of a list of machines are capable of serving individual client requests. Thus, the director applet makes a service-specific choice of a physical host to contact based on an internal list of (dynamically changing) server sites. Ideally, this choice should balance load among servers while maximizing performance to the end user.

While the choice of load balancing algorithm is service specific, we enumerate a number of sample techniques. The simplest approach is to randomly pick among service providers. While this approach is simple to implemement and does not require server modifications, it can result in both poor load balancing and poor performance to the end user. For example, an `FTP` applet picking randomly among a list of service providers may pick an under-powered mirror site on another continent. A refinement on random load balancing would bias future random choices based on how quickly requests to a particular server are pro-

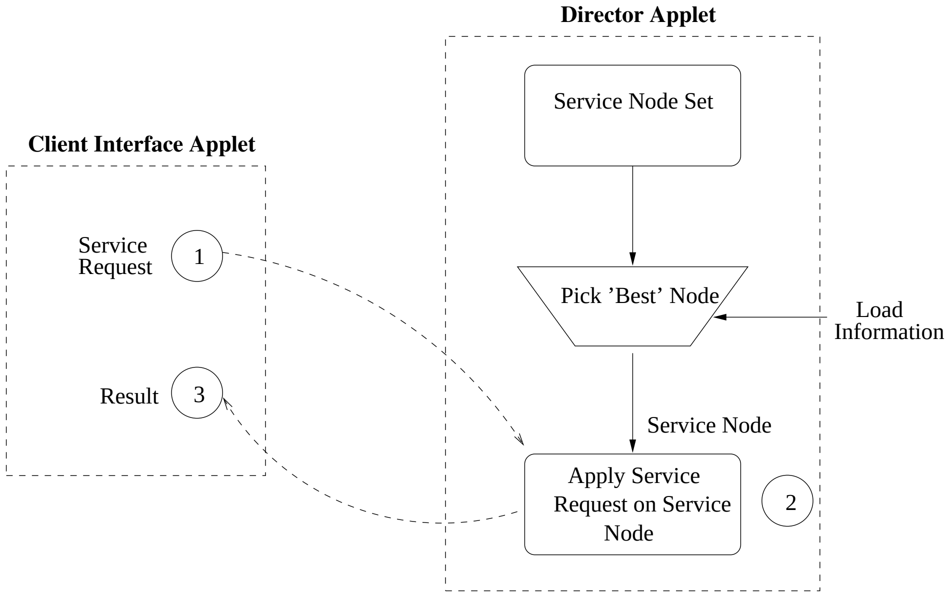

Figure 3: This Figure describes the Smart Client architecture. (1) The client interface applet first makes a request to the director applet. (2) The director applet, given outside information such as load and changes to the service group membership, picks the best node to apply the request to. The director will also re-apply the request if the operation fails. (3) The result of the operation, including a success/failure code, is returned to the client interface applet.

cessed. For services where multiple successive requests are likely, we believe this technique should result in good performance while maintaining implementation simplicity.

Another technique involves maintaining service-specific profiles of servers. In the `FTP` example above, a description of hardware performance and network connectivity (perhaps using techniques similar to the Internet Weather Report [Mat 1996]) may be associated with each server. The Smart Client director applet can then use this information to evaluate available bandwidth to each server based on the client's location. A further improvement requires maintaining load information for each server. In this case, the client is able to maximize performance by weighing a combination of network connectivity, server performance, and current server load.

The mechanisms used for load balancing can be adapted to provide fault transparency to the end user. Techniques such as keep-alive messages or time outs can be utilized to determine server failure. Upon failure, the director applet can reinvoke the load balancing mechanism to choose an alternate server and reapply the request. By storing all uncompleted server requests and necessary client state information, the director applet can connect to an alternative site to retransmit all outstanding requests transparently to the user.

3.1.2 Updating Applet State

In order to make load balancing decisions, the client may need a reasonably current profile of the individual servers providing the service. Depending on the application, updating the director of changes in service state can be achieved through either lazy or eager techniques, presenting both performance and semantic tradeoffs for maintaining consistency.

Examples of eager update techniques include client polling and server callbacks. Using client polling of servers to maintain load information has the disadvantage of severely loading server machines. Server callback techniques can be more scalable than client polling, however they require server modifications and increase implementation complexity. Neither client polling nor server callbacks is likely to scale to the level of thousands of clients necessary for some Web services. Eager update methods are appropriate when accurate information is required and the scale of the service is small enough to support eager protocols.

Lazy update techniques [Ladin et al. 1992] are likely to be more appropriate in the context of the Web. Lazy updates reduce network traffic by sending information only occasionally, after a number of updates have been collected. One particularly attractive mode of lazy updates is piggy-backing update information with server replies to client requests. For example, a server can inform Smart Client directors applets of the addition of new server machines comprising the service when replying to a director request.

3.1.3 Director Architecture

Smart Clients provide a very flexible mechanism for implementing service-specific transparency. The Smart Client director provides the illusion of a single, highly-available machine to the programmer of the client interface applet. Requests made by the Smart Client client interface applet are written to operate on a single machine. The director applet chooses the destination server based on service-specific information such as load, availability, processor speed, or connection speed.

The director accepts arbitrary requests of the form "perform this action on a server node". The director applet sends the request to the server determined to deliver the best performance to the client. If the request fails, the next server in the director applet's ranked list is contacted with the request. In this way, the director applet provides transparent access to arbitrary server groups. As a result of the well-defined interface between the client interface applet and the director applet (as described in Section 3.3), individual director applets are easily interchanged for many different services. For example, the director applet providing transparent Telnet access to a cluster of workstations can also be used for services such as chat or `FTP`.

3.2 Bootstrapping Applet Retrieval

The goal of transparent access to network services would be compromised if the Smart Client applets necessary for service access must be downloaded from a single hostname before every service access. We have created a scalable bootstrapping mechanism to circumvent this single point of failure. To remove the single point of failure associated with a single hostname, we have modified jfox [Wendt 1996], an existing Java Web browser, to support a new *service* name space, e.g. *service://now chat service*. For the service name space, the browser contacts one of many well-known search engines with a query. These well-known search engines serve the same purpose as the root name servers in DNS.

Currently, the browser contacts Altavista [Dig 1995] with a query requesting an HTML page whose title matches the service name, e.g. "now chat service". In this way, Smart Clients leverages highly-available search engines to provide translations from well-known service names to a URL. The URL points to a page containing a *service certificate*. The certificate includes references to both the client interface and director applets. In addition, the certificate contains some initial guess as to service group membership. This hint initializes the director applet, allowing the applet to validate the list by contacting one of the nodes. Figure 4 shows a certificate used for the NOW chat service.

Jfox has been extended to cache the Smart Client applets associated with individual services, the location of the service certificate, the certificate itself and any additional state that the Smart Client director needs for the next access to the service. While the client interface applet and service-certificate are cached using normal browser disk caching mechanisms, the director state is saved by serializing the director applet (and any relevant instance variables) to disk using Java Object serialization [Jav 1996]. Thus, on subsequent service accesses, the director applet need not rely on the initial group membership contained in the service certificate. Instead, it can use the last known service group membership. With this bootstrapping mechanism, no network communication is necessary to load the service applets after the initial access.

Currently, the service certificate and applets are cached indefinitely. In the future, we plan on adding a time-out period to the server certificate. After the timeout, the browser can revalidate both its service certificate and the associated applets. If either the certificate or applets are inaccessible, the decision to proceed with the cached state can be made on a service-specific basis.

Note that with the exception of bootstapping, the implemented applets work on unmodified Java-capable Web browsers such as Netscape Navigator and Internet Explorer. Further, mainstream browsers such as Internet Explorer allow for installation of filters over the entire browser [Leach 1996]. Such a filter would allow our bootstrapping mechanism to be implemented in widely used Web browsers.

The bootstrapping problem has been addressed in other contexts. For example, distributed applications need access to DNS without a name server. Such applications fall back to sending queries well-known root name servers when it is unable to resolve a hostname. As another example, applications which communicate through RPCs must bind to a server without using an RPC. This problem is also addressed by using broadcast to initiate binding to RPC servers on the network.

```
<HTML>
<TITLE>now chat service</TITLE>
<META name="description" content="now chat service">
<META name="keywords" content="now chat service">
<APPLET name="now_chat"
        codebase="Chat" code="Chat.class">
<param  name="director" value="now_chat_director">
</APPLET>
<APPLET name="now_chat_director"
        codebase="Chat" code="ChatDirector.class">
<param  name="nodes"
 value="u81.cs.berkeley.edu, u82.cs.berkeley.edu, u83.cs.berkeley.edu">
</APPLET>
</HTML>
```

Figure 4: This example of a *service certificate* references both the client interface applet (Chat.class) and the director applet (ChatDirector.class). Initial service group membership (u81,u82 and u83) is fed to the director applet for bootstrapping purposes. The director applet contacts one of these machines to obtain group membership updates.

3.3 Smart Clients API

In this subsection, we will describe the Smart Clients API. The goal of the API is to provide a generic interface for service providers to develop transparent access to their servers and to make it easier for programmers to implement applications for distributed services. In the interests of brevity, we do not document the interface in its entirety. Interested readers can download the Java classes implementing the API to see how the classes are used to implement a number of sample applications (as described in Section 4).

Figure 5 presents a high-level overview of the Java methods which make up the Smart Clients API. The `IDirector` interface provides a simple abstraction of a service to the application programmer. The programmer makes director requests through the IDirector interface. The requests are then sent by the director applet to one of the service nodes; note that the application programmer is not concerned with managing server nodes. If the request fails, a director exception is raised. In response, the director will first allow the request to clean up any state, then resend the request to another server. The director applet takes a best effort approach in delivering the request. Thus, a return of false from the delivery request indicates a catastrophic failure of the service, i.e. all servers have failed.

4 Sample Applications

4.1 Telnet Front-End for a NOW

The NOW (Network of Workstations) Project [Anderson et al. 1995a] at UC Berkeley provides approximately 100 workstations for use within the department; however, it is difficult for users to know which of the 100 machines is least loaded. To address this problem, we developed a Web page containing a single button which, when pressed, opens a telnet window onto the least loaded machine in the NOW cluster.

The implementation of the telnet application is straightforward. The telnet Web page encapsulates the necessary Smart Clients applets. The director applet periodically polls the NOW's operating system, GLUnix [Ghormley et al. 1995], to retrieve the load averages of machines in the cluster through a simple Common Gateway Interface (CGI) program. When the user clicks on the telnet button (provided by the client interface applet), a request is sent to start a telnet window on the least loaded machine in the cluster. If the director applet notices that a machine has failed it will not submit telnet requests to that node. We are currently investigating a fault-tolerant telnet service which re-opens a telnet window (with saved state such as the current working directory and environment variables) in the event of node failure. The fault-tolerant telnet would pass this saved state through the `RequestException` object (as described in Figure 5.

4.2 Scalable FTP Interface

We have also used Smart Clients to build a scalable frontend for FTP sites. As a motivating example, the Netscape Navigator FTP download page[2] contains twelve hyperlinks for netscape FTP hosts. Users choose among netscape sites or mirrors to perform

[2] http://www.netscape.com/comprod/mirror-/client_download.html

```java
//  Interface to encapsulate all client interface applet requests
public interface IRequest {
        //  Downcall from the director to the request object.  Perform the action
        //  on 'hostname'.  Throw RequestException if an error occurs
        public void action(String hostname) throws RequestException;
        //  Downcall from the director to the request object upon failure.  Perform
        //  any necessary cleanup code.  The state of the failed request consists
        //  of the 'oldhostname' and the RequestException that was thrown from the
        //  action method
        public void cleanup(String oldhostname, RequestException t);
}

//  Generic interface for all director applets
public interface IDirector {
        //  Execute the request r on a hostname of the director's choosing.  If the
        //  request object throws a RequestException, assume failure of the node
        //  and reapply the request after calling the request's cleanup method.
        //  If there are no remaining nodes, return false.  Otherwise, return true.
        public boolean apply(IRequest r);
}
```

Figure 5: This Figure describes some of the interfaces in the Smart Clients API. Classes that implement the director interface have been written to provide much of the functionality necessary to simple directors, including randomized directors (picking a random machine) and directors based on choosing the least loaded server.

manual load balancing. To improve on this interface, Smart Client applets present a single download button to the user. The client interface applet delivers requests to the director to retrieve a file, while the director picks a machine at random from a static set of nodes. When the user presses the button, the applet transparently determines the best site for file retrieval.

To demonstrate the scalability available from using Smart Clients, we measure delivered bandwidth to Smart Client applets running in Netscape Navigator from a varying number of FTP servers. We emphasize that the choice of FTP site is transparent to the end user (a single button is pressed to begin file retrieval) and that our FTP application can be downloaded to run with unmodified servers and Java-compliant browsers. The tests were run on a cluster of Sun Sparcstation 10's and 20's interconnected by a 10 Mbps Ethernet switch. The Ethernet switch allows each machine in the cluster to simultaneously deliver 10 Mb of aggregate bandwidth to the rest of the cluster without the the contention associated with shared Ethernet networks. Either one, two, or four of the machines are designated FTP servers, while the rest of the machines in the cluster attempt 40 consecutive retrievals of a 512 KB file. This experimental setup approximates multiple FTP mirror sites spread across the wide area.

Figure 6 summarizes the results of the FTP scalability tests. The graph shows aggregate delivered bandwidth in megabytes per second as a function of the number of client machines making simultaneous file requests. For one FTP server, 8 clients are able to saturate the single available Ethernet link at 1.2 MB/s[3]. The results for two and four FTP servers demonstrate that the random selection of an FTP server used within the applet delivers reasonable scalability. Sixteen clients are able to retrieve approximately 2 MB/s from two servers, while 16 clients saturate four servers at approximately 3 MB/s.

For small number of clients, a single FTP server demonstrates the best performance because all 40 file downloads are made during a single connection. For the multi-server tests, multiple connections and disconnections take place as the clients attempt to randomly balance load across the servers. In the future, this problem can be avoided by implementing site affinity with successive file requests (if delivered bandwidth on the previous was deemed satisfactory), implementing a load daemon on the nodes to allow the clients to continuously choose lightly loaded machines, or by using services such as the Internet Weather Map [Mat 1996] to choose low-latency hosts. This information can be used to incrementally scale connections to available FTP servers (i.e. allow some machines to be recruited only when needed).

[3]We were unable to take measurements for more than 16 simultaneous clients making requests to a single server because the FTP server would not allow more than 16 simultaneous file retrievals. We plan to investigate this limitation further.

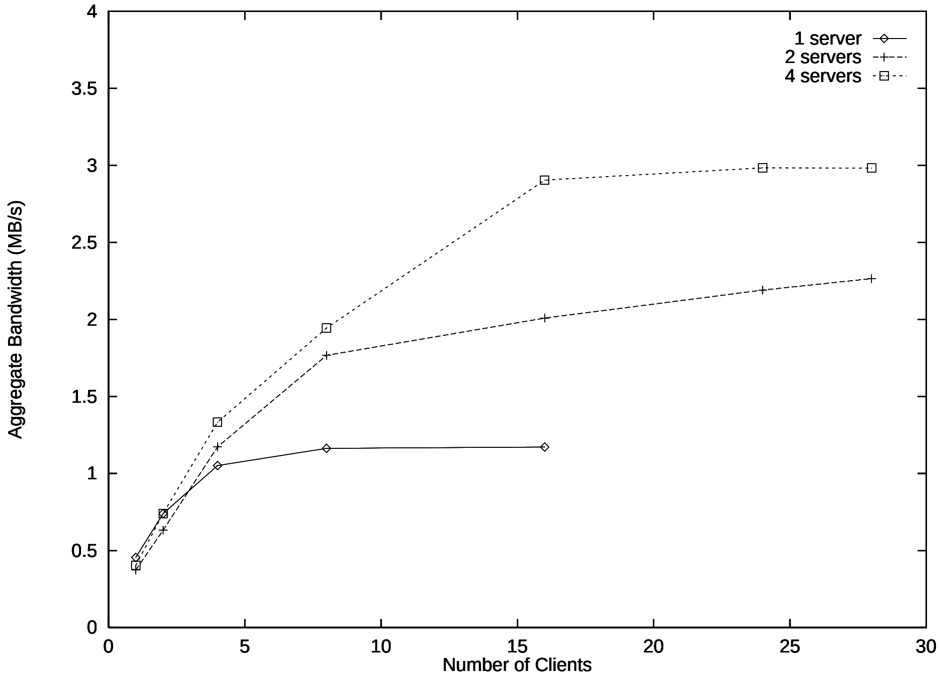

Figure 6: This figure demonstrates how a Smart Client interface to `FTP` delivers scalable performance. The graph shows delivered aggregate bandwidth as a function of number of clients making simultaneous requests.

4.3 Scalable Chat

The next application we implement is Internet chat. The application allows for individuals to enter and leave chat rooms to converse with others co-located in the same logical room. The chat application is implemented as Java applets run through a Web browser. Figure 7 depicts our implementation of the application. Individual chat rooms are modeled as files exported through WebFS [Vahdat et al. 1996], a file system allowing global URL read/write access. WebFS provides for negotiation of various cache consistency protocols on file open.

We extended WebFS to implement a scalable caching policy suitable to the chat application. In this model, when a user wishes to enter a chat room, the client simply opens a well-known file associated with the room. This operation registers the client with WebFS. Read and write operations on the file correspond to receiving messages from other chatters and sending a message out to the room, respectively. On receiving a file update (new message), WebFS sends the update to all clients which had opened the file for reading (i.e., all chatters in a room). In this case, the client interface applet consists of two threads, a read thread continuously polling the chat file and an event thread writing user input to the chat file. These read/write requests are sent to the chat server via the director applet.

The director sends the request to the hostname that represents the best service node at the time. If the request does not complete, the request raises an exception to the director applet. The director applet then calls the service-specific cleanup routine for the request, and resends it to another service node. Note that the request takes a service specific failure event, such as chat file not found or WebFS server is down, and translates it into a general exception. Thus, the director applet can be written for a cluster of machines and reused for many different protocols: `FTP`, Telnet and chat.

From the above discussion, it is clear that a single WebFS server can quickly become a performance bottleneck as the number of simultaneous users is scaled. To provide system scalability, we allow multiple WebFS servers to handle client requests for a single file. Each server keeps a local copy of the chat file. Upon receiving a client update, WebFS distributes the updates to each of the chat clients connected to it. WebFS also accumulates updates, and every 300 ms propagates the updates to other servers in the WebFS group. This caching model allows for out of order message delivery, but we deemed such semantics to be acceptable for a chat application. If it is determined that such semantics are insufficient, well-known dis-

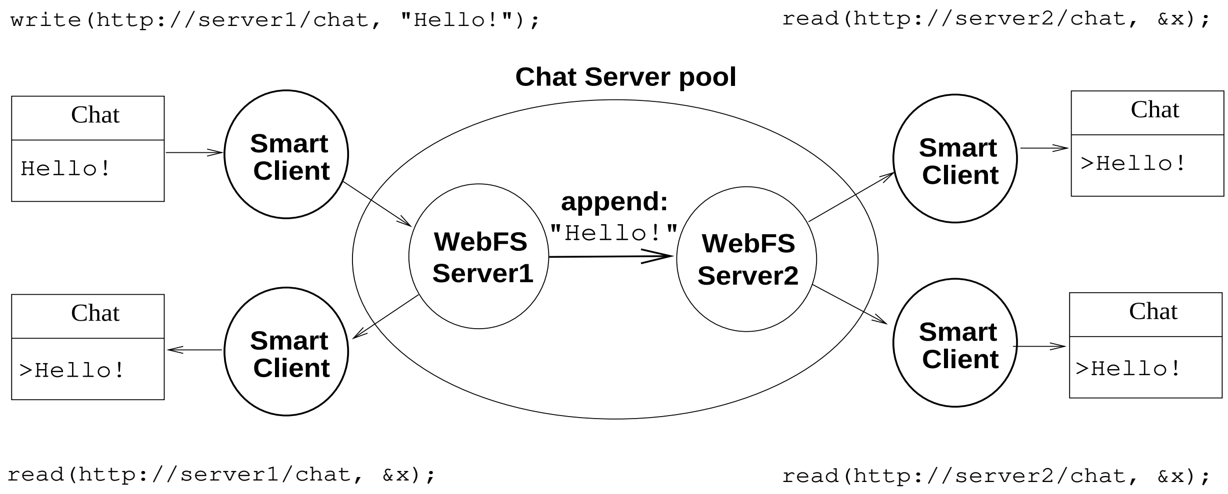

Figure 7: Implementation of the chat application. Chat rooms are modeled as files with reads corresponding to receiving conversation updates and writes to sending out a message. On a write, the WebFS updates all its clients; the updates are propagated to other servers in a lazy fashion.

tribution techniques [Ladin et al. 1992, Birman 1993] can be used to provide strong ordering of updates.

Since the read requests are idempotent, and the write requests are atomic with respect to WebFS, the chat application is completely tolerant to server crashes. This fault transparency provides the illusion of a single, highly-available chat server machine to the programmer of the Chat client interface applet. Figure 8 demonstrates the behavior of the chat application in the face of a failure to the client's primary server. The graph plots response time as a function of elapsed time. The graph shows that chat delivers less than 5 ms latency to the end user. On detecting a failure, the latency jumps to 1 second before switching to a secondary WebFS server, at which point the latency returns to normal.

5 Summary

We have described a solution to the problem of scalability and high-availability which logically migrates server functionality into the client. We will now revisit the goals set forth in Section 1 and examine how Smart Clients addresses each goal:

- Incremental Scalability - When a machine is added to or removed from a service group, the director applet supplied by the service updates its list of peer servers. The director applet discovers such modifications through a service-specific mechanism, e.g. keep-alive messages, connecting to a well-known port, or refetching the service certificate.

- Load Balancing - The director applet maintains a service-specific notion of load (such as number of processes, number of open connections, available bandwidth). Using this information, client requests are routed to the best candidate node.

- Fault Transparency - When a failure occurs, the director applet allows the client to clean up any stale state before resending the request to another server. Providing fault transparency requires service support when the request is non-idempotent. For example, in the chat application, the chat service provides atomic writes to the chat transcript.

- Wide Area Service Topology - Smart Clients does not place any restriction on topology of server machines. In fact, the director applet can choose an arbitrary site based on considerations such as proximity or predicted performance.

- Scalable Services To Legacy Servers - Existing servers that replicate a read-only database can be grouped together for access by Smart Clients. With knowledge of the group set, the director applet can load balance client requests among existing unmodified servers.

Finally, we believe that the architecture presented in this paper can simplify implementation of scalable services with respect to at least fault transparency and load balancing. The Smart Client director provides the illusion of a single, highly available server. This

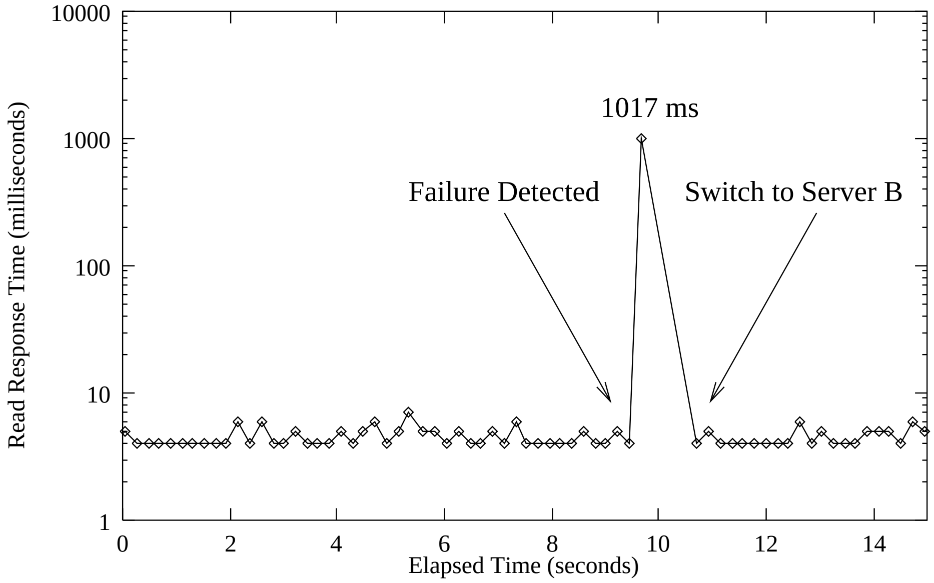

Figure 8: Chat response times in the face of server load. The chat application delivers latencies of approximately 10 ms under normal circumstances. On server failure, the applications takes one second to switch to a peer server.

model substantially decreases the complexity of the client interface applet since this applet need not be concerned with maintaining the set of server nodes. In addition, because of the public interface between the client interface and director applets, each can be written once and interchanged for a number of different services.

6 Related Work

The problem of transparently providing fault transparency and load balancing to network services has been addressed previously in a number of contexts. File systems have used server-side replication of volumes and servers to provide fault transparency in systems such as Deceit [Marzullo et al. 1990], AFS [Howard et al. 1988], and HA-NFS [Bhide et al. 1991]. More recently, systems such as xFS [Anderson et al. 1995b] and Petal [Lee & Thekkath 1996] use client-side techniques to improve overall file system performance. Many distributed clusters perform load balancing on the level of jobs (interactive or otherwise) submitted to the system [Nichols 1987, Bricker et al. 1991, Douglis & Ousterhout 1991, Zhou et al. 1992]. Once again, all these systems implement server-side solutions for load balancing and require client intervention to spread jobs among cluster machines.

Perhaps most closely related to Smart Clients are Transaction Processing monitors [Gray & Reuter 1993] (TP monitors). TP monitors provide functionality similar to Smart Clients for access to databases. The TP monitor functions as the director for transactions to *resource managers*, accounting for load on machines, the RPC program number, and any affinity between client and server. Resource managers are usually SQL databases, but can be any server that supports transactions. TP monitors differ from Smart Clients in that they deal exclusively with transactional RPCs as the communication mechanism to the servers. TP monitors are also more closely coupled with the server nodes since they are responsible for starting new server processes.

The Smart Client director can be tailored to each service, while the TP monitor is more of a general purpose director. Smart Clients also provide a bootstrapping mechanism to remove the single point of failure associated with downloading the necessary routing software. In addition, the Smart Client code is significantly more lightweight than the TP monitor which often includes many of the features of traditional operating systems: process management/creation, authentication, and linking resource manager object code with the Transaction Processing operating system (TPOS). This lightweight nature enables Smart Clients to be downloaded into existing Web browsers to customize existing Internet services.

Also related to our systems are ISIS [Birman 1993] and gossip architectures [Ladin et al. 1992] which provide a substrate for developing distributed applications. ISIS provides reliable group communication to support many of the applications we envision. Gossip architectures use front-ends analogous to Smart Clients to access replicated servers which are kept consistent through lazy updates. Both systems are orthogonal to our work in many respects and still use server-side techniques for much of their functionality.

7 Conclusions

In this paper, we have shown that existing solutions to providing transparent access to network services suffer from a lack of knowledge about the semantics of individual services. The recent advent of Java allowing distribution of portable client code presents an opportunity to migrate certain service functionality onto the client machine. We show that such migration can simplify service implementation and improve the quality of service to users. To this end, we describe our implementation of Smart Clients to show the greater flexibility available from a client-side approach to building scalable services. The Smart Clients API provides a generic interface for accessing network services. Further, the decomposition of the API into individual client interface and director applets allows interchanging of these applets for a variety of services. The Smart Clients API is specialized to provide scalable access to three sample services: `telnet`, `FTP`, and Internet chat.

In the future, we will further explore service-specific load balancing techniques for achieving scalability. We also plan to demonstrate how Smart Clients can be used to provide load balancing and fault transparency for services replicated across the wide area. We also plan to implement an interface for transparent access to `HTTP` servers and a fault-tolerant `telnet` client. Migration of other code, besides the director, from the server to the client will also be explored.

Acknowledgments

We would like to thank our shepherd Rob Gingell, Bruce Mah, Rich Martin, and Neal Cardwell for their help in substantially improving the presentation of this paper. We would also like to thank Thomas Wendt for providing the source to jfox, the Java Web browser we modified to support Smart Clients.

References

[Anderson et al. 1995a] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. "A Case for NOW (Networks of Workstations)". *IEEE Micro*, February 1995.

[Anderson et al. 1995b] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. "Serverless Network File Systems". In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 109–126, December 1995.

[Anderson et al. 1996] E. Anderson, D. Patterson, and E. Brewer. "The Magicrouter, an Application of Fast Packet Interposing". May 1996. Submitted For Publication. Also see http://HTTP.CS.Berkeley.EDU/~eanders-/magicrouter/.

[Berners-Lee 1995] T. Berners-Lee. "Hypertext Transfer Protocol HTTP/1.0", October 1995. HTTP Working Group Internet Draft.

[Bhide et al. 1991] A. Bhide, E. N. Elnozahy, and S. P. Morgan. "A Highly Available Network File Server". In *Proceedings of the 1991 USENIX Winter Conference*, pp. 199–205, 1991.

[Birman 1993] K. P. Birman. "The Proecss Group Appraoch to Reliable Distributed Computing". *Communications of the ACM*, 36(12):36–53, 1993.

[Bricker et al. 1991] A. Bricker, M. Litzkow, and M. Livny. "Condor Technical Summary". Technical Report 1069, University of Wisconsin—Madison, Computer Science Department, October 1991.

[Brisco 1995] T. Brisco. "DNS Support for Load Balancing", April 1995. Network Working Group RFC 1794.

[Dig 1995] Digital Equipment Corporation. *Alta Vista*, 1995. http://www.altavista.digital.com/.

[Douglis & Ousterhout 1991] F. Douglis and J. Ousterhout. "Transparent Process Migration: Design Alternatives and the Sprite Implementation". *Software - Practice and Experience*, 21(8):757–85, August 1991.

[Ghormley et al. 1995] D. Ghormley, A. Vahdat, and T. Anderson. "GLUnix: A Global Layer UNIX for NOW". See http://now.cs.berkeley.edu-/Glunix/glunix.html, 1995.

[Goldstein & Dale 1995] I. Goldstein and P. Dale. "A Sacalable, Fault Resilient Server for the WWW". OSF ARPA Project Proposal, 1995.

[Gosling & McGilton 1995] J. Gosling and H. McGilton. "The Java(tm) Language Environment: A White Paper". http://java.dimensionx.com-/whitePaper/java-whitepaper-1.html, 1995.

[Gray & Reuter 1993] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Howard et al. 1988] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. "Scale and Performance in a Distributed File System". *ACM Transactions on Computer Systems*, 6(1):51–82, February 1988.

[Jav 1996] JavaSoft. *Java RMI Specification, Revision 1.1*, 1996. See http://chatsubo.javasoft.com/-current/doc/rmi-spec/rmiTOC.doc.html.

[Katz et al. 1994] E. D. Katz, M. Butler, and R. McGrath. "A Scalable HTTP Server: The NCSA Prototype". In *First International Conference on the World-Wide Web*, April 1994.

[Ladin et al. 1992] R. Ladin, B. Liskov, L. Shirira, and S. Ghemawat. "Providing Availability Using Lazy Replication". *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.

[Leach 1996] P. Leach. Personal Communication, November 1996.

[Lee & Thekkath 1996] E. K. Lee and C. A. Thekkath. "Petal: Distributed Virtual Disks". In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[Marzullo et al. 1990] K. Marzullo, K. Birman, and A. Siegel. "Deceit: A Flexible Distributed File System". In *Proceedings of the 1990 USENIX Summer Conference*, pp. 51–61, 1990.

[Mat 1996] Matrix Information and Directory Services, Inc. *MIDS Internet Weather Report*, 1996. See http://www2.mids.org/weather/index.html.

[Net 1994] Netscape Communications Corporation. *Netscape Navigator*, 1994. http://www.-netscape.com.

[Nichols 1987] D. Nichols. "Using Idle Workstations in a Shared Computing Environment". In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pp. 5–12, November 1987.

[Vahdat et al. 1996] A. Vahdat, M. Dahlin, and T. Anderson. "Turning the Web into a Computer". May 1996. See http://now.cs.berkeley.edu/WebOS.

[Wendt 1996] T. Wendt. *Jfox*, 1996. http://www.uni-kassel.de/fb16/ipm/mt/java/jfox.html.

[Wetherall & Tennenhouse 1995] D. Wetherall and D. L. Tennenhouse. "Active Networks: A New Substrate for Global Applications". 1995. Submitted for Publication.

[Zhou et al. 1992] S. Zhou, J. Wang, X. Zheng, and P. Delisle. "Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computing Systems". Technical Report CSRI-257, University of Toronto, 1992.