# 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 offered load to many Internet sites. To use multiple machines for scalable performance, issues involving load balancing, fault tolerance, and backward compatibility with URL naming must be addressed . A number of server-side approaches have been developed to provide this level of transparent access to Internet services, including HTTP redirect, DNS aliasing, Magic Routers, and Active Networks. In this paper, we will argue that in many instances, a client-side approach to providing transparent access to Internet services provides increased flexibility and performance. We describe the design and implementation of *Smart Clients* and show how our system can be used to provide a transparently scalable and highly available interface to three network services: 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 thousands to millions of requests per day 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 indexing services, to scale gracefully with offered load. Such scalable services must, at minimum, deal with the following problems:

- Incremental Scalability – If the offered load begins to exceed a service's hardware capacity, it should be a simple operation to add hardware to increase system capacity. This operation should be transparent to users (i.e., no service interruption).
- Load Balancing – Load should be spread dynamically among server resources so that clients receive the best available quality of service.

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. The authors can be contacted at {`chad, bnc, eastham, vahdat, tea, culler`}`@cs.berkeley.edu`.

- Dynamic Resource Recruitment – Related to the issue of scalability, a service should be able to transparently 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 Bay Area earthquake in May, 1996.

- Fault Tolerance – When possible, the service should remain available in the face of host, software, or network upgrades or failures.

Unfortunately, providing these properties for network services while remaining compatible with the de facto URL naming standard has proven difficult. URLs implicitly associate a single hostname, and hence a single point of failure and congestion, with each service. A number of efforts address this limitation by hiding the physical location of a particular service behind a logical hostname. Examples of such systems include HTTP redirect, DNS Aliasing, Failsafe TCP, Active Networks, and Magic Routers. We refer to these systems as "server-side" because the transparency mechanism is logically co-located with the servers.

We will 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 server-side solutions and demonstrate how these limitations can be avoided by moving portions of server functionality into the client. This approach offers the main 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. 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. Such flexibility would be much more difficult to implement if a server attempted to implement load balancing for geographically distributed clients.

Motivated by the above observations, we describe the design and implementation of *Smart Clients* to support our argument for client-side functionality. Smart Clients provide an interface to a generic networked service. These services are assumed to be implemented by a number of servers each capable of handling individual client requests[1]. We will demonstrate how the system can be specialized to provide transparently scalable, fault-tolerant access to a number of simple network services: `telnet`, `ftp`, and an Internet chat application[2]. Our measurements indicate no significant performance degradation from using Smart Clients for load balancing and fault tolerance.

By implementing the Smart Clients API in Java [Gosling & McGilton 1995], we allow network services to provide applets (client software) to mediate access to the service. Thus, client code running on an end-user's host can be viewed as an extension of the server. To extend the ftp example from above, the local applet (as provided by the server) might include references to mirror sites providing the same service. Further, the API allows clients to query servers about their current load and to gather estimates of current latency (or perhaps available bandwidth) to various servers.

The rest of this paper is organized as follows. In the next section, we will discuss in more detail the tradeoffs associated with client versus server-side implementation of scalable service. Section 3

[1]This assumption holds for many widely-used Internet services such as `http`, `ftp`, and Web indexing services.

[2]Our implementation of both Smart Clients and our sample applications can be retrieved from http://now.cs.berkeley.edu/SmartClients.

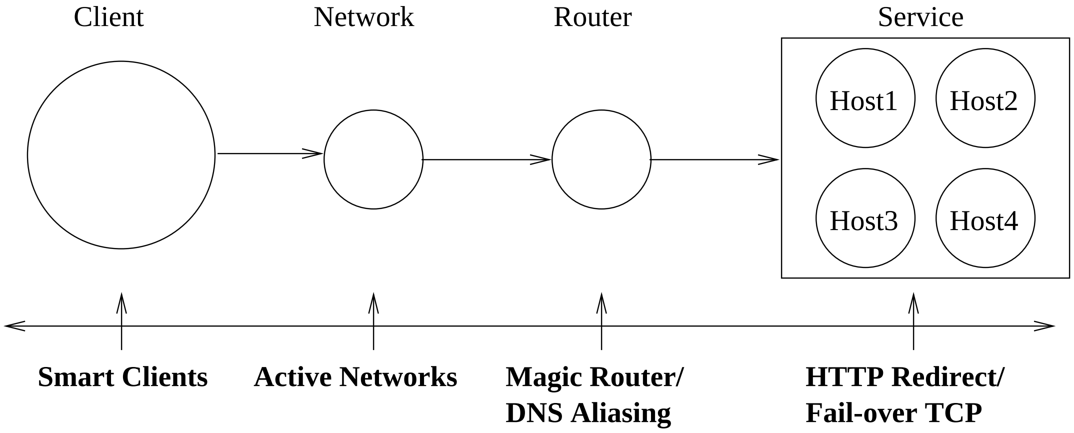


Figure 1: This figure describes the design space for providing transparent access to scalable network services. Transparency mechanisms can be implemented in a number of places, including the client, network, network routers, or at the service site.

describes the initial Smart Clients API. Section 4 demonstrates the utility of the API by describing the implementation and performance of interfaces for `telnet`, `ftp`, and a scalable chat service. In section 6, our work is compared to systems which address similar problems in other contexts. We present our conclusions in section 7.

## 2 Alternative Solutions

In this section, we will describe approaches for adding transparent fault tolerance and load balancing to Web services. Popular approaches currently include DNS Aliasing [Brisco 1995, Katz et al. 1994], HTTP redirect [Berners-Lee 1995], Magic Routers [Anderson et al. 1996], fail-safe 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 our description of the Smart Client architecture in Section 3.

NCSA's scalable Web server uses 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, the load of a particular machine is not considered in making a mapping. Second, hostname to IP mappings can be cached, possibly indefinitely, on the client; thus, poor load balancing can result when many active clients make requests of a single server, while others remain idle. Further, if a machine failure occurs after name resolution, the client does not have any knowledge of other available servers.

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 request, a single point of failure is still present (if the machine serving redirects is unavailable, the entire service appears unavailable), and servers can still be overloaded attempting to serve redirects. Further, like DNS aliasing, HTTP redirect cannot provide the illusion of a highly available service if a failure occurs after the redirect.

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 tolerance 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). Further, Magic Routers 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 transparent fault tolerance. 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.

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 tolerance and load balancing. However, providing knowledge of the load characteristics of servers within individual routers can be difficult, not to mention the difficulty associated with re-engineering the Internet architecture.

All of the above solutions provide a level of transparent access to network services with respect to load balancing and fault tolerance. 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.

Smart Clients implement a level of abstraction between a logical service and the physical servers comprising it in a service-specific manner. This client-side approach allows for greater system flexibility. With Smart Clients, once a user request is made, the client code can use any of a number of techniques to fulfill the request. The client may switch physical servers part way through a request or may make multiple redundant requests, all transparently to the user. Further, client-based approaches to transparency allow more flexibility for providing service across the wide area. As an example, transparency mechanisms co-located with server machines (such as the Magic Router) limit the server nodes to those in close proximity, e.g. on the same subnet. On the other hand, co-locating such mechanisms with the client places no restrictions on topology, allowing the service nodes to be transparently distributed across the wide area. This model of wide area distribution is common for the a number of popular Web services such as Netscape's home page [Net

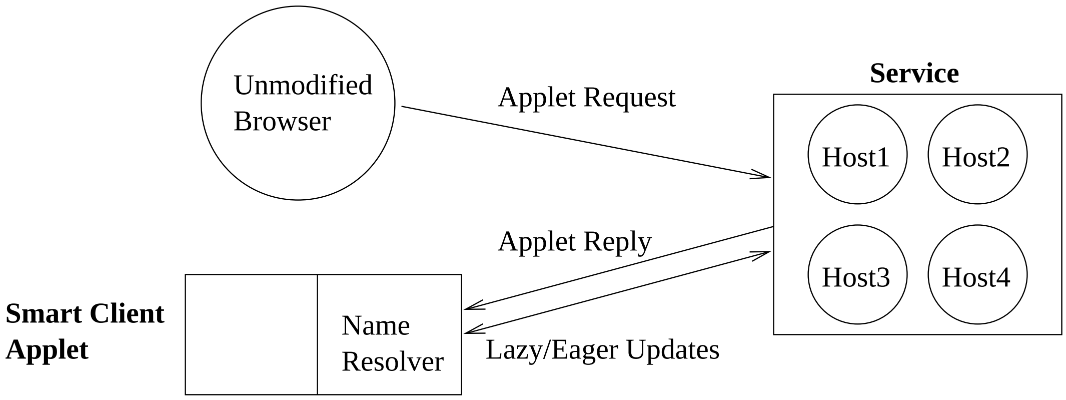


Figure 2: This Figure describes the Smart Client service access model. Rather than retrieving HTML for a service, a service-specific Java applet is supplied to mediate server access. The *name resolver* within the applet is responsible for making server requests and updating its notion of server state.

1994] or Alta Vista [Dig 1995].

Finally, Smart Clients allow for service specific load balancing techniques. An applet may use a default load balancing mechanism (random selection for example) or develop an algorithm using service-specific knowledge. For example, the applet for a compute service might select a lightly loaded server machine, while the applet for an ftp service might pick the server with highest available bandwidth.

## 3 Smart Client Architecture

In this Section, we describe our approach to providing transparent, fault tolerant, and load balanced access to network resources. Figure 2 presents an overview of the Smart Client service access model. When a user wishes to make use of a service, a bootstrapping mechanism is used to retrieve the applet used to access that service (i.e. rather than retrieving HTML, service-specific Java code is retrieved). This applet contains hints to physical machines providing the service. Each time the user makes a request, the Smart Client uses the *name resolver* to invoke a service-specific mechanism for retrieving the requested information. Depending on the application, either lazy or eager techniques can be used to keep the name resolver of changes in the service's state. 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 applet. 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 Tolerance

We begin our discussion of the Smart Client architecture by describing the methodology 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 applet makes a service-specific choice of a physical host to contact based on an internal list of potential 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 better technique involves maintaining service-specific profiles of servers. In the FTP example above, a description of hardware performance and network connectivity would be associated with each server. The Smart Client then uses 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 network connectivity, server performance, and current server load.

The mechanisms used for load balancing can be adapted to provide transparent fault tolerance to the end user. Techniques such as keep-alive messages or time outs can be utilized to determine server failure. On a failure, the client can reinvoke the load balancing mechanism to choose an alternate server. By storing all uncompleted server requests, the client can connect to an alternative site to retransmit all outstanding requests. In the future, we plan to provide a package implementing this model of transactional RPCs as a generic interface for fault tolerant service access.

### 3.1.2 Updating Applet State

The above discussion brings up a number of questions regarding the techniques used to keep an applet's state up to date. Clearly, a tradeoff is present between lazy and eager techniques 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 placing large load on servers. Server callbacks can be more scalable than client polling, however they require more server state, increasing implementation complexity. Neither technique is likely to scale to the level of millions of clients necessary in the Web environment. 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 out only occasionally, after a number of updates have been collected. One particularly attractive mode of lazy updates

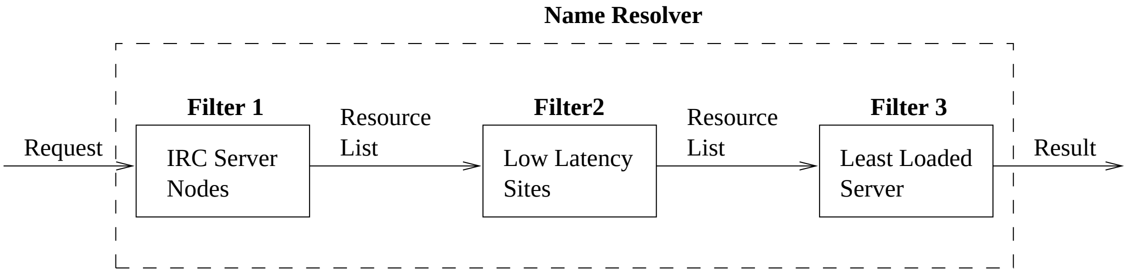


Figure 3: This Figure describes how service-specific name resolvers are used to implement load balancing. Stackable filters are used to select a lightly-loaded IRC chat server with low round-trip latency times to the client.

is piggy-backing update information with server replies to client requests. For example, a server can inform Smart Client applets of the addition of new server machines comprising the service. Alternately, a client using a random method for load balancing can bias future random choices based on how quickly responses are delivered.

#### 3.1.3 Extensible Naming

Smart Clients provide a very flexible mechanism for implementing service-specific naming mechanisms. We now describe how these mechanisms are used by service providers to implement Smart Client access to their servers. The model does not dictate the use of any specific policy for naming a server. Rather, each Smart Client applet attempts to name a physical server by making a query to the Smart Client *name resolver*. For example, the name resolver accepts logical queries of the form "choose randomly an ftp site from list of mirrors" or "choose least loaded IRC [Oikarinen & Reed 1993] server with round-trip packet delivery times under 500 ms". The name resolver implements this model of constraint-based naming as a chain of filters constructed by the application, similar to streams in UNIX SVR4.

Figure 3 graphically describes the filters needed to evaluate the IRC query from above. Three filters are constructed to recognize (i) IRC chat server nodes, (ii) sites with round-trip latencies to the client's host under 500 ms, and (iii) a notion of load on IRC nodes. Each filter takes as input a list of available resources and is responsible for retrieving information necessary to perform the filter. The result is resources with a list of unsatisfied constraints (empty if all constraints were met).

Since query results may change over time (i.e., clients may want updated load information to periodically reconnect to a lightly loaded server), the resolver provides two mechanisms for keeping the client aware of updated query results. The applet can explicitly request query re-evaluation or an optional thread can be associated with each query filter. This thread actively updates information associated with the filter. If the results of the filter changes, the next filter in the link is invoked, eventually resulting in a callback to the applet. For example, a filter which chooses the least loaded server can periodically poll for load or block waiting for server updates; the application is notified via a callback if the result of the query changes.

### 3.2 Bootstrapping Applet Retrieval

The goal of transparent access to network services would be compromised if the applet needed to access a service must be downloaded from a single hostname before each access. One possible bootstrapping mechanism to circumvent this problem works as follows. To retrieve the applet for accessing a service, users would ordinarily use a Java compliant Web browser to access a well known host, such as `now.cs.berkeley.edu`. To remove the single point of failure associated with a single hostname, our bootstrapping model allows the retrieval to time out. In the case of a time out, the client contacts a number of well-known search engines, such as Inktomi [Brewer & Gauthier 1995] or Alta Vista [Dig 1995], to find alternative sites for the applet (these sites contain keywords indicating replication of the required applet).

In the future, caching techniques can be used to avoid bootstrapping altogether in most cases. Web broswers can be extended to cache applets associated with individual services. Cache consistency is not critical in this situation since individual applets can update outdated load information. In the case where the applet contains code which can no longer be used to access a service, the applet can initiate the bootstrapping mechanism described above to reload itself.

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.

### 3.3 Smart Clients API

In this subsection, we will describe how the design considerations described above are embodied in 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. 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 Sections 4 and 5).

Figure 4 presents a high-level overview of the Java methods which make up the Smart Clients API. The class `NameResolver` contains an application specific chain of filters used for lookups on a vector (list) of constraints. For example, a `Constraint` might specify "hostname in Berkeley domain". Associated with each constraint is a `Filter` which understands the semantics of the constraint. The API does not assign any meaning to `Constraint` or `Name` objects. However, the toolkit includes a number of filters for performing common operations such as finding lightly loaded Internet hosts.

The Smart Clients API does not impose any constraints on how stale name lookups are invalidated or re-resolved. If a `Filter` is constructed as active, then a thread can update its results periodically in a service-specific manner. For example, for a query such as "least loaded ftp server", the result of the query will vary with time. To obtain current values for least loaded server, an

```
class NameResolver {
    /* Perform name resolution and return satisfying resources */
    Resource Lookup(Vector constraints);
    Resource Reevaluate(); // Force reevaluation of the filter chain
    void AddLink(Filter filter); // add new filter to head of chain of filters
}

/* Resource is defined to be a vector of names (e.g. hostnames implementing
   a service) and a list of constraints unsatisfied by the names */
class Resource {
    Vector  constraints;
    Vector  names;
}

class Filter {
    /* Constructor specifying whether filter has active thread associated
       with it and whether it should update its children if value changes */
    Filter(boolean active, boolean update);
    /* Perform a specific filter operation and pass results to next filter */
    void ComputeOutputResource(Resource inputResource);
}
```

Figure 4: This Figure describes the Smart Clients API. Java classes are provided for name resolution and for construction of filters to allow applications to perform attribute name lookups.

active thread can be associated with the name resolver's filter which periodically polls available ftp sites for the number of users connected to the server (i.e., server load). If the query results change, an application callback is made with the update.

# 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 fairly straightforward. The telnet web page contains a Smart Clients applet, and can be viewed by Java-compliant browsers such as Netscape Navigator [Net 1994] or Hotjava [Sun 1995]. The name resolver is instantiated as an active thread which 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, the applet simply starts a telnet window on the least loaded machine in the cluster. Empirical observations indicate that the system performs well in load balancing access to NOW machines.

### 4.2 Scalable FTP Interface

After validating the Smart Client implementation with the telnet interface describe above, we now demonstrate the use of Smart Clients to build a scalable frontend for ftp sites. As a motivating example, the Netscape Navigator FTP download page[3] contains twelve hyperlinks for netscape ftp hosts. Users choose among netscape sites or mirrors to perform manual load balancing. To improve on this interface, we provide a Smart Client applet which presents a single download button to the user. By pressing 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 code 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.

Figure 5 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 servers, 8 clients are able to saturate the single available Ethernet link at 1.2 MB/s[4]. 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, we will avoid this problem by modifying the ftp servers to provide information about the number of clients currently connected to the server. This information can be used to incrementally scale connections to available ftp servers.

## 5 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

[3] `http://www.netscape.com/comprod/mirror/client_download.html`

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

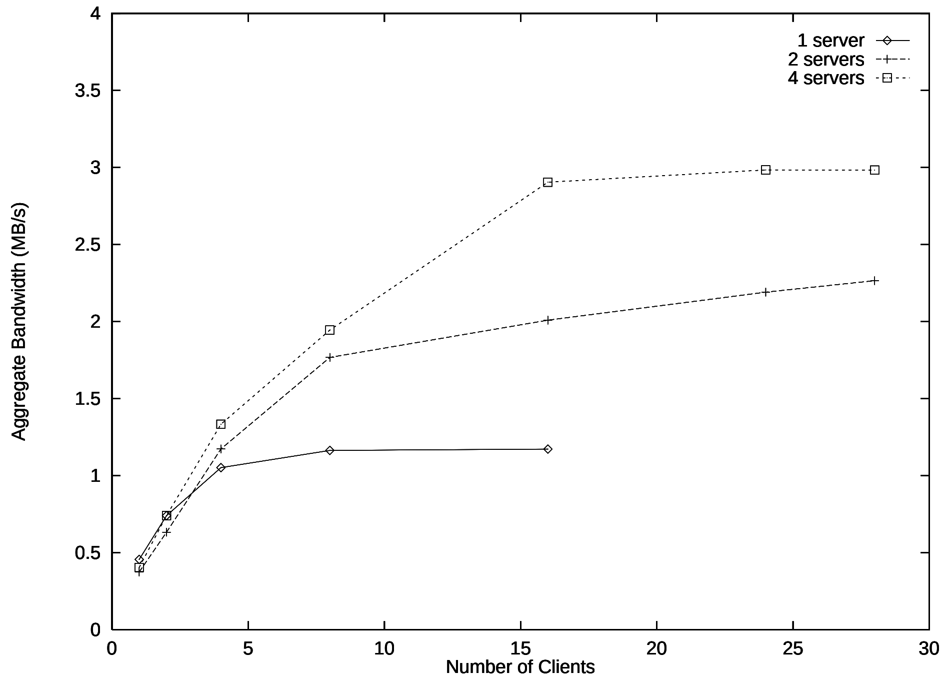


Figure 5: This figure demonstrates how a Smart Client interface to ftp delivers scalable performance. The graph shows delivered aggregate bandwidth as a function of number of clients making simultaneous requests.

chat application is implemented as Java applets which can be run through Sun's applet viewer application (Navigator's restrictions on file access prevent the execution of the applet). Figure 6 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).

From the above discussion, it is clear that a single WebFS server can quickly become a performance bottleneck if multiple users are contributing to a conversation or if a single WebFS server is responsible for multiple rooms. To provide for 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

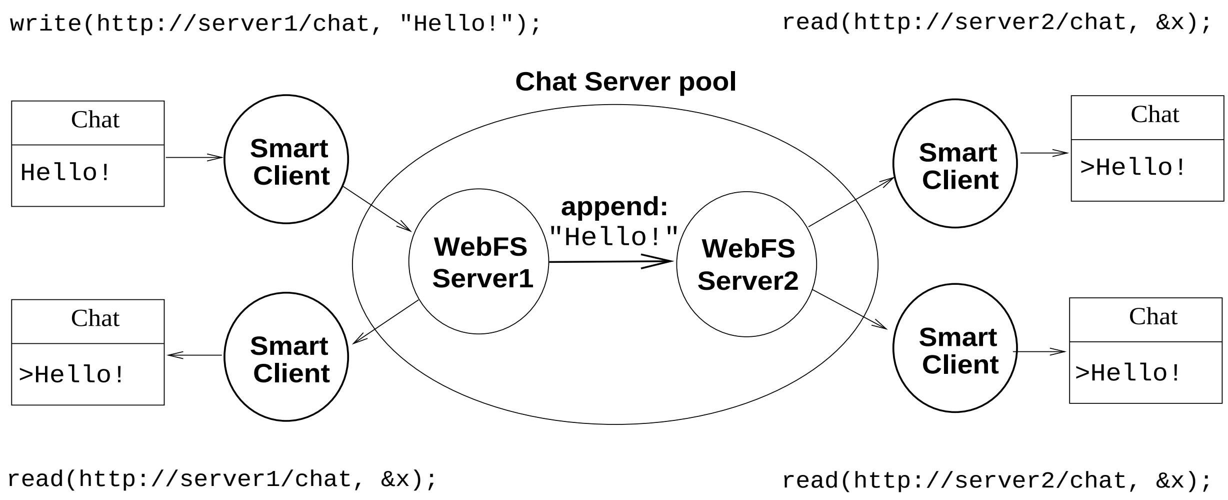


Figure 6: This Figure describes our implementation of the chat application. Chat rooms are modeled as files with reads corresponding to receiving conversation updates and writes to sending out a message. On a write, the WebFS updates all its clients; the updates are propagated to other servers in a lazy fashion.

every 300 ms all accumulated updates are propagated to the 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 distributed techniques [Ladin et al. 1992, Birman 1993] can be used to provide strong ordering of updates.

A preliminary prototype of the chat application is up and running. Initial measurements of the system indicate that the system is indeed scalable. One WebFS server propagates approximately 30 updates per second to four clients attempting to send out chat messages as quickly as possible (i.e., in a tight loop). Two WebFS servers, on the other hand, are able to propagate approximately 60 updates per second to each of the 4 clients (two per server with lazy updates between the servers). Our results, while very preliminary for the chat application, indicate that our implementation is likely to yield the desired scalability. For the final version of the paper, we will have measurements of up to 4 chat servers serving up to 30 clients.

## 6 Related Work

The problem of transparently providing fault tolerance 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 tolerance 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 our systems are ISIS [Birman 1993] and so-called gossip architectures [Ladin et al. 1992] which provide a substrate for developing distributed applications. ISIS reliable group communication to support the type of applications we envision. Gossip architectures use front-ends analogous to Smart Clients to access replicated servers which 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. In the past, server-side approaches to providing fault tolerance and load balancing were more convenient because they provided greater transparency to the clients. With the recent advent of Java however, it is possible for clients to download server code (applets) to run locally. We argue that in many instances, client-side approaches for transparency offer increased flexibility by using service-specific knowledge to perform load balancing and fault tolerance. In this context, 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. We show how this API can be specialized to provide scalable access to three such sample services: `telnet`, `ftp`, and Internet chat.

In the future, we will further explore service-specific load balancing techniques for achieving scalability. Further, we will demonstrate how Smart Clients can be used to provide load balancing and fault tolerance for services replicated across the wide area. We also plan to implement an interface for transparent access to HTTP servers. Finally, we hope to show that bootstrapping retrieval of Smart Client applets can be simplified through simple additions to Web browsers.

## Acknowledgments

We would like to thank Bruce Mah and Rich Martin for their comments on drafts of this paper.

## 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.

[Brewer & Gauthier 1995] E. Brewer and P. Gauthier. "The Inktomi Search Engine". http://-www.inktomi.com, 1995.

[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.

[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.

[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.

[Lee & Thekkath 1996] E. K. Lee and C. A. Thekkath. “Petal: Distributed Virtual Disks”. In *Proceedings of the 6th 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.

[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.

[Oikarinen & Reed 1993] J. Oikarinen and D. Reed. “Internet Relay Chat Protocol”, May 1993. Network Working Group RFC 1459.

[Sun 1995] Sun Microsystems. *The HotJava(tm) Browser*, 1995. http://www.javasoft.com/-HotJava/index.html.

[Vahdat et al. 1996] A. Vahdat, M. Dahlin, and T. Anderson. “Turning the Web into a Computer”. May 1996. Submitted for publication. Also see http://now.cs.berkeley.edu/WebOS.

[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.