Extensible Cluster-Based Scalable Network Services

Armando Fox Steven D. Gribble Yatin Chawathe Eric A. Brewer Paul Gauthier
University of California at Berkeley Inktomi Corporation
{fox, gribble, yatin, brewer}@cs.berkeley.edu {brewer, gauthier}@inktomi.com

We identify three fundamental requirements for scalable network services: incremental scalability and overflow growth provisioning, 24x7 availability through fault masking, and cost-effectiveness. We argue that clusters of commodity workstations interconnected by a high-speed SAN are exceptionally well-suited to meeting these challenges for Internet-server workloads, provided the software infrastructure for managing partial failures and administering a large cluster does not have to be reinvented for each new service. To this end, we propose a general, layered architecture for building cluster-based scalable network services that encapsulates the above requirements for reuse, and a service-programming model based on composable workers that perform transformation, aggregation, caching, and customization (TACC) of Internet content. For both performance and implementation simplicity, the architecture and TACC programming model exploit BASE, a weaker-than-ACID data semantics that results from trading consistency for availability and relying on soft state for robustness in failure management. Our architecture can be used as an "off the shelf" infrastructural platform for creating new network services, allowing authors to focus on the "content" of the service (by composing TACC building blocks) rather than its implementation. We discuss two real implementations of services based on this architecture: TranSend, a Web distillation proxy deployed to the UC Berkeley dialup population, and HotBot, the commercial implementation of the Inktomi search engine. We present detailed measurements of TranSend's performance based on substantial client traces, as well as anecdotal evidence from the TranSend and HotBot experience, to support the claims made for the architecture.

1 Introduction

"One of the overall design goals is to create a computing system which is capable of meeting almost all of the requirements of a large computer utility. Such systems must run continuously and reliably 7 days a week, 24 hours a day... and must be capable of meeting wide service demands."

"Because the system must ultimately be comprehensive and able to adapt to unknown future requirements, its framework must be general, and capable of evolving over time."

— Corbató and Vyssotsky on *Multics*, 1965 [15]

Although it is normally viewed as an operating system, Multics (Multiplexed Information and Computer Service) was originally conceived as an infrastructural computing service, so it is not surprising that its goals as stated above are similar to our own. The primary obstacle to deploying Multics was the absence of the network infrastructure, which is now in place. Network applications have exploded in popularity in part because they are easier to manage and evolve than their desktop application counterparts: they eliminate the need for software distribution, and simplify customer service and bug tracking by avoiding the difficulty of dealing with multiple platforms and versions. Also, basic queueing theory shows that a large central (virtual) server is more efficient in both cost and utilization than a collection of smaller servers; desktop systems represent the degenerate case of one "server" per user. All of these are key parts of the argument for Network Computers [27].

However, network services remain difficult to deploy because of three fundamental challenges: scalability, availability and cost effectiveness.

- By *scalability,* we mean that when the offered load to the service increases, an *incremental and linear* increase in hardware can maintain the same per-user level of service.
- By *availability,* we mean that the service as a whole must be available 24x7, despite transient partial hardware or software failures.
- By *cost effectiveness*, we mean that the service must be economical to administer and expand, even though it potentially comprises many workstation nodes.

We observe that clusters of workstations have some fundamental properties that can be exploited to meet these requirements: using commodity PCs as the unit of scaling allows the service to ride the leading edge of the cost/performance curve, the inherent redundancy of clusters can be used to mask transient failures, and "embarrassingly parallel" network service workloads map well onto networks of workstations. However, developing cluster software and administering a running cluster remain complex. The primary contributions of this work are the design and analysis of an implemented layered framework for building network services that addresses this complexity. New services can use this framework as an off-the-shelf solution to scalability, availability, and several other problems, and focus instead on the *content* of the service being developed. The lower layer handles scalability, availability, load balancing, support for bursty offered load, and system monitoring and visualization, while the middle layer provides extensible support for caching, transformation among MIME types, aggregation of information from multiple sources, and personalization of the service for each of a large number of users (*mass customization*). The top layer allows composition of transformation and aggregation into a specific service, such as accelerated Web browsing or a search engine.

Pervasive throughout our design and implementation strategies is the observation that much of the data manipulated by a network service can tolerate semantics weaker than ACID [25]. We combine ideas from prior work on availability vs. consistency and the use of soft state for robust fault-tolerance to characterize the data semantics of many network services, which we refer to as BASE semantics (basically available, soft state, eventual consistency). In addition to demonstrating how BASE simplifies the implementation of our architecture, we present a programming model for service authoring that is a good fit for BASE semantics and that maps well onto our cluster-based service framework.

1.1 Validation: Two Real Services

Our framework reflects the implementation of two real network services in use today: TranSend, a scalable transformation and caching proxy for the 25,000 Berkeley dialup IP users (connecting through a bank of 600 modems), and the Inktomi search engine (commercialized as HotBot), which performs millions of queries per day against a database of over 50 million web pages.

The Inktomi search engine is an aggregation server that was initially developed to explore the use of cluster technology to handle the scalability and availability requirements of network services. The commercial version, HotBot, handles several million queries per day against a full-text database of 54 million web pages. It has been incrementally scaled from 5 to 26 nodes (and soon to 66), provides high availability, and is extremely cost effective. Inktomi predates the framework we describe, and thus differs from it in some respects. However, it strongly influenced the framework's design, and we will use it to validate particular design decisions.

We focus our detailed discussion on TranSend, which provides Web caching and data transformation. In particular, real-time, datatype-specific distillation and refinement [21] of inline Web images results in an end-to-end latency reduction of 3-5x, giving the user a much more responsive Web surfing experience with only modest image quality degradation. TranSend was developed at UC Berkeley and has been deployed for the 25,000 home-IP dialup users there, and is in the process of being deployed to a similar community at UC Davis.

In the remainder of this section we argue that clusters are an excellent fit for Internet services, provided the challenges we describe for cluster software development can be surmounted. In Section 2 we describe the proposed layered architecture for building new services, and a programming model for creating services that maps well onto the architecture. We show how TranSend and HotBot map onto this architecture, using HotBot to justify specific design decisions within the architecture. Sections 3 and 4 describe the TranSend implementation and its measured performance, including experiments on its scalability and fault tolerance properties. Section 5 discusses related work and the continuing evolution of this work, and we summarize our observations and contributions in Section 6.

1.2 Advantages of Clusters

Particularly in the area of Internet service deployment, clusters provide four primary benefits over single larger machines, such as SMPs: incremental scalability, high availability, and the cost/performance and maintenance benefits of commodity PC's. We elaborate on each of these in turn.

Scalability: Clusters are well suited to Internet service workloads, which are highly parallel (many independent simultaneous users) and for which the grain size typically corresponds to at most a few CPU-seconds on a commodity PC. For these workloads, large clusters can dwarf the power of the largest machines. For example, Inktomi's HotBot cluster contains 60 nodes with 120 processors, 30 GB of physical memory, and hundreds of commodity disks. Wal-Mart uses a cluster from TeraData with 768 processors and 16 terabytes of online storage.

Furthermore, the ability to grow clusters incrementally over time is a tremendous advantage in areas such as Internet service deployment, where capacity planning depends on a large number of unknown variables. Incremental scalability replaces capacity planning with relatively fluid reactionary scaling. Clusters correspondingly eliminate the "forklift upgrade", in which you must throw out the current machine (and related investments) and replace it via forklift with an even larger one.

High Availability: Clusters have natural redundancy due to the independence of the nodes: Each node has its own busses, power supply, disks, etc., so it is "merely" a matter of software to mask (possibly multiple simultaneous) transient faults. A natural extension of this capability is to temporarily disable a subset of nodes and then upgrade them in place ("hot upgrade"). Such capabilities are essential for network services, whose users have come to expect 24-hour uptime despite the inevitable reality of hardware and software faults due to rapid system evolution.

Commodity Building Blocks: The final set of advantages of clustering follows from the use of commodity building blocks over high-end, low-volume machines. The obvious advantage is cost/performance, since memory, disks, and nodes can all track the leading edge; for example, we changed the building block every time we grew the HotBot cluster, each time picking the reliable high volume previous-generation commodity units, helping to ensure stability and robustness. Furthermore, since many commodity vendors compete on service (particularly for PC hardware), it is easy to get high-quality configured nodes in 48 hours or less. Large SMPs typically have a lead time of 45 days, are more cumbersome to purchase, install, and upgrade, and are supported by a single vendor, so it is much harder to get help when difficulties arise. Once again, it is a "simple matter of software" to tie a collection of possibly heterogeneous commodity building blocks together.

To summarize, clusters have significant advantages in scalability, growth, availability, and cost. Although fundamental, these advantages are not easy to realize.

1.3 Challenges of Cluster Computing

There are a number of areas in which clusters are at a disadvantage relative to SMP's. In this section we describe some of these challenges and how they influenced the architecture we will propose in Section 2.

Administration: Administration is a serious concern for systems of many nodes. We leverage ideas in prior work [1], which describes how a unified monitoring/reporting framework with data visualization support was an effective tool for simplifying cluster administration.

Component vs. system replication: Each commodity PC in a cluster is not usually powerful enough to support an entire service, but can probably support some *components* of the service. Component-level rather than whole-system replication therefore allows commodity PCs to serve as the unit of incremental scaling, provided the software can be naturally decomposed into loosely coupled modules. We address this challenge by proposing an architecture in which each component has well-circumscribed functional responsibilities and is largely "interchangeable" with other components of the same type. For example, a cache node can run anywhere that a disk is available, and a worker that performs a specific kind of data compression can run anywhere that significant CPU cycles are available.

Partial failures: Component-level replication leads directftly to the fundamental issue separating clusters from SMPs: the need to handle *partial failures*, i.e. the ability to survive and adapt to failures of subsets of the system. Traditional workstations and SMPs never face this issue, since the machine is either up or down.

Shared state: Unlike SMPs, clusters have no shared state. Although much work has been done to emulate global shared state through software distributed shared memory [31,32,34], we can improve performance and reduce complexity if we can avoid or minimize the need for shared state across the cluster.

These two concerns, partial failure and shared state, lead us to focus on the sharing semantics actually required by network services.

1.4 BASE Semantics

We believe that the design space for network services can be partitioned according to the data semantics that each service demands. At one extreme is the traditional transactional database model with the ACID properties (atomicity, consistency, isolation, durability) [25], providing the strongest semantics at the highest cost and complexity. ACID makes no guarantees regarding *availability*; indeed, it is preferable for an ACID service to be unavailable than to function in a way that relaxes the ACID constraints. ACID semantics are well suited for Internet commerce transactions, billing users, or maintaining user profile information for personalized services.

For other Internet services, however, the primary value to the user is not necessarily strong consistency or durability, but rather *high availability* of data:

- **Stale** data can be temporarily tolerated as long as all copies of data eventually reach consistency after a short time [19] (e.g. DNS servers do not reach consistency until entry timeouts expire [39]).
- **Soft state** [13], which can be regenerated at the expense of additional computation or file I/O, is exploited to improve performance; data is not durable.
- **Approximate** answers (based on stale data or incomplete soft state) delivered quickly may be more valuable than exact answers delivered slowly.

We refer to the data semantics resulting from the combination of these techniques as *BASE*—Basically Available, Soft State, Eventual Consistency. By definition, any data semantics that are not strictly ACID are BASE. BASE semantics allow us to handle partial failure in clusters with less complexity and cost. Like pioneering systems such as Grapevine [7] , BASE reduces the complexity of the service implementation, essentially trading consistency for simplicity; like later systems such as Bayou [19] that allow trading consistency for availability, BASE provides opportunities for better performance. For example, where ACID requires durable and consistent state across partial failures, BASE semantics often allows us to avoid communication and disk activity or to postpone it until a more convenient time.

In practice, it is simplistic to categorize every service as either ACID or BASE; instead, different *components* of services demand varying data semantics. Directories such as Yahoo! [64] maintain a database of soft state with BASE semantics, but keep user customization profiles in an ACID database. Transformation proxies [22,57] interposed between clients and servers transform Internet content on-the-fly; so the transformed content is BASE data that can be regenerated by computation, but if the service bills the user per session, the billing should certainly be delegated to an ACID database.

We focus on services that have an ACID component, but manipulate primarily BASE data. Web servers, search/aggregation servers [58], caching proxies [12,41], and transformation proxies are all examples of such services; our framework supports a superset of these services by providing integrated support for the requirements of all four. As we will show, BASE semantics greatly simplify the implementation of fault tolerance and availability and permit performance optimizations within our framework that would be precluded by ACID.

2 Cluster-Based Scalable Service Architecture

In this section we propose a system architecture and service-programming model for building scalable network services on clusters. The architecture attempts to address both the challenges of cluster computing and the challenges of deploying network services, while exploiting clusters' strengths. We view our contributions as follows:

- A proposed system architecture for scalable network services that exploits the strengths of cluster computing, as exemplified by cluster-based servers such as TranSend and HotBot.
- Separation of the *content* of network services—i.e., what the services do—from their implementation, by encapsulating the "scalable network service requirements" of high availability, scalability, and fault tolerance in a reusable layer with narrow interfaces.
- A programming model that maps well onto our system architecture, based on composition of stateless worker modules into new services, and demonstrate that numerous existing services map directly onto it.
- Detailed measurements of a production service that instantiates the architecture and validates our performance and reliability claims.

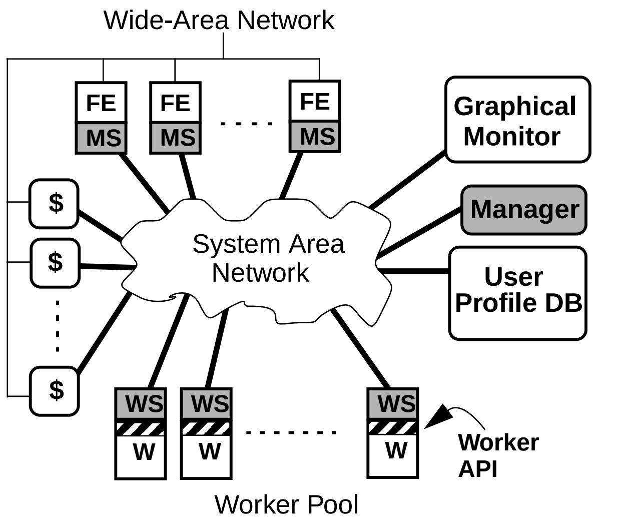

Figure 1: Architecture of a generic SNS. Components include front ends (FE), a pool of workers (W) some of which may be caches ($), a user profile database, a graphical monitor, and a fault-tolerant load manager, whose functionality logically extends into the manager stubs (MS) and worker stubs (WS).

The remainder of this section reviews the benefits and challenges of cluster computing, proposes a network service architecture that exploits these observations and allows encapsulation of essential implementation requirements for scalable services, and describes a programming model that minimizes service development effort by allowing implementation of new services entirely at the higher layers.

2.1 Proposed Functional Organization of an SNS

The above observations lead us to the software-component block diagram of a generic SNS shown Figure 1. Each physical workstation in a NOW supports one or more software components in the figure, but each component in the diagram is confined to one node. In general, the components whose tasks are naturally parallelizable are replicated for scalability, fault tolerance, or both. In our measurements (Section 4), we will argue that the performance demands on the non-replicated components are not significant for the implementation of a large class of services, and that the practical bottlenecks are bandwidth into and out of the system and bandwidth in the SAN.

Front Ends provide the interface to the SNS as seen by the outside world (e.g. HTTP server). They "shepherd" incoming requests by matching them up with the appropriate *user profile* from the customization database, and queueing them for service by one or more *workers*. Front ends maximize system throughput by maintaining state for many simultaneous outstanding requests, and can be replicated for both scalability and availability.[1]

The **Worker Pool** consists of caches and service-specific modules that implement the actual service (data transformation/filtering, content aggregation, etc.) Each type

1: Although availability and coarse-grained load balancing have typically been achieved using DNS round-robin [10] or IP packet redirection [14], today's Web clients support these mechanisms within the client itself, using JavaScript [43].

Service: Service-specific code

- Workers that present human interface to what TACC modules do, including device-specific presentation
- User interface to control the service
- complex services: implement directly on SNS layer.

TACC: Transformation,Aggregation, Caching, Customization

- API for composition of stateless data transformation and content aggregation modules
- Uniform caching of original, post-aggregation and post-transformation data
- Transparent access to Customization database

SNS: Scalable Network Service support

- Incremental and absolute scalability
- Worker load balancing and overflow management
- Front-end availability, fault tolerance mechanisms
- System monitoring and logging

Figure 2: Scalable Network Service Layered Model

of module may be instantiated zero or more times, depending on offered load.

The **Customization Database** stores user profiles that allow mass customization of request processing.

The **Manager** balances load across workers and spawns additional workers as offered load fluctuates or faults occur. When necessary, it may assign work to machines in the *Overflow Pool*, a set of backup machines (perhaps on desktops) that can be harnessed to handle load bursts and provide a smooth transition during incremental growth (e.g. as the load increases, the system notices increased usage of the overflow nodes and can notify the administrator to scale up the system.)

The **Graphical Monitor** for system management supports tracking and visualization of the system's behavior, asynchronous error notification via email or pager, and temporary disabling of system components for hot upgrades.

The **System-Area Network** provides a low-latency, high-bandwidth interconnect, such as switched 100-Mb/s Ethernet or Myrinet [40]. Its main goal is to prevent the interconnect from becoming the bottleneck as the system scales.

2.2 Separating Service Content From Implementation: A Reusable SNS Support Layer

Layered software models allow layers to be isolated from each other and allow existing software in one layer to be reused in different implementations. We observe that the components in the above architecture can be grouped naturally into three layers of functionality, as shown in Figure 2: SNS (scalable network service implementation), TACC (transformation, aggregation, caching, customization), and Service. The key contributions of our architecture are the reusability of the SNS layer, and the ability to add simple, stateless "building blocks" at the TACC layer and the ability to compose them in the Service layer. We discuss TACC in Section 2.3. The SNS layer, which we describe here, provides scalability, load balancing, fault tolerance, and high availability; it comprises the Front Ends, Manager, SAN, and Monitor in Figure 1.

2.2.1 Scalability

Our SNS architecture decomposes functionality into well-encapsulated components that may be replicated for the sake of fault tolerance or high availability. It is through this replication that we also achieve scalability: when the offered load to the system saturates the capacity of some component class, more instances of that component can be launched on additional nodes that are incrementally added to the system. The duties of our replicated components are largely independent of each other (because of the nature of the internet services' workload), which means the amount of additional resources required is a linear function of the increase in offered load. Although the components are mostly independent, they do have some dependence on the shared, non-replicated system components: the SAN, the resource manager, and possibly the user profile database. Our measurements in Section 4 confirm that even for very large systems, these shared components do not become a bottleneck.

The static partitioning of functionality between front ends and workers reflects the desire to keep workers as simple as possible, by localizing in the front ends the control decisions associated with satisfying user requests. In addition to managing the network state for outstanding requests, front ends encapsulate service-specific dispatch logic that selects which workers to invoke, access the profile database to pass the appropriate parameters to the workers, notify the end user in a service-specific way (e.g., constructing an HTML page describing the error) when one or more workers fails unrecoverably, provide the user interface to the profile database, and so forth. This division of responsibility allows workers to remain simple and stateless, and allows the behavior of the *service* as a whole to be defined almost entirely in the front end. If the workers are analogous to processes in a Unix pipeline, the front end is analogous to an interactive shell.

2.2.2 Centralized Load Balancing

Load balancing is controlled by a centralized policy implemented in the Manager. The Manager collects load information from the workers, synthesizes load balancing hints based on the policy, and transmits the hints to the front ends, which make local scheduling decisions based on the most recent hints. The load balancing and overflow *policies* are left to the system operator. We describe our experiments with load balancing and overflow in Section 4.5.

The decision to centralize rather than distribute load balancing is intentional: If the load balancer can be made fault tolerant, and if we can ensure it does not become a performance bottleneck, centralization makes it easier to implement and reason about the behavior of the load balancing policy. In Section 3.1.3 we discuss how this was achieved and discuss the evolution that led to this design decision and its implications for performance, fault tolerance, and scalability.

2.2.3 Prolonged Bursts and Incremental Growth

Although we would like to assume that there is a well-defined average load and that arriving traffic follows a Poisson distribution, burstiness has been demonstrated for Ethernet traffic [33], file system traffic [26], and Web requests [16], and is confirmed by our traces of web traffic (discussed later). In addition, Internet services can experience relatively rare but prolonged bursts of high load: after the recent landing of Pathfinder on Mars, its web site served over 220 million hits in a 4-day period [42]. Often, it is during such bursts that uninterrupted operation is most critical.

Our architecture includes the notion of an *overflow* pool for absorbing these bursts. The overflow machines are not dedicated to the service, and normally do not have workers running on them, but the Manager can spawn workers on the overflow machines on demand when unexpected load bursts arrive, and release the machines when the burst subsides. In an institutional or corporate setting, the overflow pool could consist of workstations on individuals' desktops. Because worker nodes are already interchangeable, workers do not need to know whether they are running on a dedicated or an overflow node, since load balancing is handled externally. In addition to absorbing sustained bursts, the ability to temporarily harness overflow machines eases incremental growth: when the overflow machines are being recruited unusually often, it is time to purchase more dedicated nodes for the service.

2.2.4 Soft State for Fault Tolerance and Availability

The technique of constructing robust entities by relying on cached soft state refreshed by periodic messages from peers has been enormously successful in wide-area TCP/IP networks [4,18,37], another arena in which transient component failure is a fact of life. Correspondingly, our SNS components operate in this manner, and monitor one another using *process peer fault tolerance*[2]: when a component fails, one of its peers restarts it (on a different node if necessary), while cached stale state carries the surviving components through the failure. After the component is restarted, it gradually rebuilds its soft state, typically by listening to multicasts from other components. We give specific examples of this mechanism in Section 3.1.3.

We use timeouts as an additional fault-tolerance mechanism, to infer certain failure modes that cannot be otherwise detected. If the condition that caused the timeout can be automatically resolved, e.g. workers lost because of a SAN partition can be restarted on still-visible nodes, the Manager performs the necessary actions. Otherwise, the SNS layer reports the suspected failure condition, and the service layer determines how to proceed (e.g. report the error or fall back to a simpler task that does not require the failed worker).

2.2.5 Narrow Interface to Service-Specific Workers

To allow new services to reuse all these facilities, the Manager and front ends provide a narrow API, shown as the Manager Stubs and Worker Stubs in Figure 1, for communicating with the workers, the Manager, and the graphical system monitor. The Worker Stub provides mechanisms for workers to implement some required behaviors for participating in the system, e.g. supplying load data to assist the Manager in load balancing decisions and reporting detectable failures in their own operation. The Worker Stub hides fault tolerance, load balancing, and multithreading considerations from the worker code, which may use all the facilities of the operating system, need not be thread-safe, and can, in fact, crash without taking the system down. The minimal restrictions on worker code allow worker authors to focus instead on the *content* of the service, even using off-the-shelf code (as we have) to implement the worker modules.

The Manager Stub linked to the Front Ends provides support for implementing the dispatch logic that selects which worker type(s) are needed to satisfy a request; since the dispatch logic is independent of the core load balancing and fault tolerance mechanisms, a variety of services can be built using the same set of workers.

2.3 TACC: A Programming Model for Internet Services

Having encapsulated the "SNS requirements" into a separate software layer, we now require a programming model for building the services themselves in higher layers. We focus on a particular subset of services, based on *transformation, aggregation, caching, and customization* of Internet content (TACC). Transformation is an operation on a single data object that changes its content; examples include filtering, transcoding, re-rendering, encryption, and compression. Aggregation involves collecting data from several objects and collating it in a prespecified way; for example, collecting all listings of cultural events from a prespecified set of Web pages, extracting the date and event information from each, and composing the result into a dynamically-generated "culture this week" page. Our initial implementation allows Unix-pipeline-like chaining of an arbitrary number of stateless transformations and aggregations; this results in a very general programming model that subsumes transformation proxies [21], proxy filters [67], customized information aggregators [59,11], and search engines. The selection of which workers to invoke for a particular request is service-specific and controlled outside the workers themselves; for example, given a collection of workers that convert images between pairs of encodings, a correctly chosen sequence of transformations can be used for general image conversion.

Customization represents a fundamental advantage of the Internet over traditional wide-area media such as television. Many online services, including the Wall Street Journal, the Los Angeles Times, and C/Net, have deployed "personalized" versions of their service as a way to increase loyalty and the quality of the service. Such *mass customization* requires the ability to track users and keep profile data for each user, although the content of the profiles differs across services. The *customization database*, in most respects a traditional ACID [25] database, maps a user identification token (such as an IP address or cookie) to a list of key-value pairs for each user of the service. A key strength of the TACC model is that the appropriate profile information is automatically delivered to workers along with the input data for a particular user request; this allows the same workers to be reused for different services. For example, an image-compression worker can be run with one set of parameters to reduce image resolution for faster Web browsing, and a different set of parameters to reduce image size and bit depth for handheld devices. We have found composable, customizable workers to be a powerful building block for developing new services, and we discuss our experience with TACC and its continuing evolution in Section 5.

Caching is important because recomputing or storing data has become cheaper than moving it across the Internet. For example, a study of the UK National web cache showed that even a small cache (400MB) can reduce the load on the network infrastructure by 40% [61], and SingNet, the largest ISP in Singapore, has saved 40% of its telecom charges using web caching [60]. In the TACC model, caches can store *post-transformation* (or post-aggregation) content and even intermediate-state content, in addition to caching original Internet content.

Many existing services are subsumed by the TACC model and fit well with it. (In Section 5.4 we describe some that do not.) For example, the HotBot search engine collects search results from a number of database partitions and collates the results. Transformation involves converting the input data from one form to another. In TranSend, graphic images can be scaled and filtered through a low-pass filter to tune them for a specific client or to reduce their size. A key strength of our architecture is the ease of *composition* of tasks; this affords considerable flexibility in the transformations and aggregations the service can perform, without requiring workers to understand task-chain formation, load balancing, etc., any more than programs in a Unix pipeline [51] need to understand the implementation of the pipe mechanism.

We claim that a large number of interesting services can be implemented entirely at the service and TACC layers, and that relatively few services will benefit from direct modification to the SNS layer unless they have very specific low-level performance needs.

2: Not to be confused with *process pairs*, a different fault-tolerance mechanism for hard-state processes, discussed in [5]. Process peers are similar to the fault tolerance mechanism explored in the early "Worm" programs [55] and to "Robin Hood/Friar Tuck" fault tolerance: "Each ghost-job would detect the fact that the other had been killed, and would start a new copy of the recently slain program within a few milliseconds. The only way to kill both ghosts was to kill them simultaneously (very difficult) or to deliberately crash the system." [49]

In Section 5.1 we describe our experience adding functionality at both the TACC and service layers.

3 Service Implementation

This section focuses on the implementation of TranSend, a scalable Web distillation proxy, and compares it with HotBot. The goals of this section are to demonstrate how each component shown in Figure 1 maps into the layered architecture, to discuss relevant implementation details and trade-offs, and to provide the necessary context for the measurements we report in the next section.

3.1 TranSend SNS Components

3.1.1 Front Ends

TranSend runs on a cluster of SPARCstation 10 and 20 machines, interconnected by switched 10baseT Ethernet and connected to the dialup pool by a single 10baseT segment. The TranSend front end presents an HTTP interface to the client population. A thread is assigned to each arriving TCP connection. Request processing involves fetching Web data from the caching subsystem (or from the Internet on a cache miss), pairing up the request with the user's customization preferences, sending the request and preferences to a pipeline of one or more *distillers* (the TranSend lossy-compression workers) to perform the appropriate transformation, and returning the result to the client. Alternatively, if an appropriate distilled representation is available in the cache, it can be sent directly to the client. A large thread pool allows the front end to sustain throughput and maximally exploit parallelism despite the large number of potentially long, blocking operations associated with each task, and provides a clean programming model. The production TranSend runs with a single front-end of about 400 threads.

3.1.2 Load Balancing Manager

Client-side JavaScript support [43] balances load across multiple front ends, although other mechanisms such as round-robin DNS [10] or commercial routers [14] could also be used. For internal load balancing, TranSend uses a centralized manager whose responsibilities include tracking the location of distillers, spawning new distillers on demand, balancing load across distillers of the same class, and providing the assurance of fault tolerance and system tuning. We argue for a centralized as opposed to distributed manager because it is easier to change the load balancing policy and reason about its behavior; the next section discusses the fault-tolerance implications of this decision. The front ends and distillers are linked with libraries (the *manager stub* and *distiller stub*, respectively) that encapsulate communication among the front end, distillers, and the manager.

The manager periodically beacons its existence on an IP multicast group to which the other components subscribe. The use of IP multicast provides a level of indirection and relieves components of having to explicitly locate each other. When the front end has a task for a worker, the manager stub code contacts the manager, which locates an appropriate worker, spawning a new one if necessary. The manager stub caches the new worker's location for future requests.

The distiller stub attached to each distiller accepts and queues requests on behalf of the distiller and periodically reports load[3] information to the manager. The manager aggregates load information from all distillers, computes weighted moving averages, and piggybacks the resulting information on its beacons to the manager stub. The manager stub (at the front end) caches the information in these beacons and uses lottery scheduling [63] to select a distiller for each request. The cached information provides a backup so that the system can continue to operate (using slightly stale load data) even if the manager crashes. Eventually, the fault tolerance mechanisms (discussed in Section 3.1.3) restart the manager and the system returns to normal.

To allow the system to scale as the load increases, the manager has a tuning mechanism. If it detects excessive load on distillers of a particular class, it can automatically spawn a new instance on an unused node. (The spawning and load balancing policies are described in detail in Section 4.5.) Another mechanism used for adjusting to bursts in load is *overflow*: if all the nodes in the system are used up, the manager can resort to starting up temporary distillers on a set of overflow nodes. Once the burst subsides, the distillers may be reaped.

3: In the current implementation, distiller load is characterized in terms of the queue length at the distiller, optionally weighted by the expected cost of distilling each item.

3.1.3 Fault Tolerance and Crash Recovery

In the original prototype for the manager, information about distillers was kept as hard state, using a log file and crash recovery protocols similar to those used by ACID databases. Resilience against crashes was via process-*pair* fault tolerance, as in [5]: the primary manager process was mirrored by a secondary whose role was to maintain a current copy of the primary's state, and take over the primary's tasks if it detects that the primary has failed. In this scenario, crash recovery is seamless, since all state in the secondary process is up-to-date.

However, by moving entirely to BASE semantics, we were able to simplify the manager greatly and increase our confidence in its correctness. In TranSend, all state maintained by the manager is now explicitly designed to be soft state. When a distiller starts up, it registers itself with the manager, whose existence it learns of by subscribing to a well-known multicast channel. If the worker crashes before de-registering itself, the manager detects the broken connection; if the manager crashes and restarts, the distillers detect beacons from the new manager and re-register themselves. Timeouts are used as a backup mechanism to infer failures. Since all state is soft and is periodically beaconed, no explicit crash recovery or state mirroring mechanisms are required to regenerate lost state. Similarly, the front end does not require any special crash recovery code, since it can reconstruct its state as it receives the next the next few beacons from the manager.

With this use of soft state, each "watcher" process only needs to detect that its peer is alive (rather than mirroring the peer's state) and, in some cases, be able to restart the peer (rather than take over its duties). Broken connections, timeouts, or loss of beacons are used to infer component failures and restart the failed process. The manager, distillers, and front ends are process peers:

- The manager reports distiller failures to the manager stubs, which update their caches of where distillers are running.
- The manager detects and restarts a crashed front end.
- The front end detects and restarts a crashed manager.

This process peer functionality is encapsulated within the manager stub code. Simply by linking against the stub, front ends are automatically recruited as process peers of the manager.

3.1.4 User Profile Database

The service interface to TranSend allows each user to register a series of customization settings, using either HTML forms or a Java/JavaScript combination applet. The actual database is implemented using *gdbm* because it is freely available and its performance is adequate for our needs: user preference reads are much more frequent than writes, and the reads are absorbed by a write-through cache in the front end.

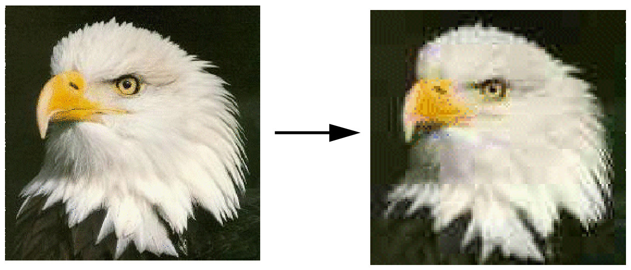
Figure 3: Scaling this JPEG image by a factor of 2 in each dimension and reducing JPEG quality to 25 results in a size reduction from 10KB to 1.5KB.

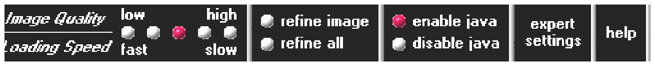

Figure 4: User Interface for manipulating preferences.

3.1.5 Cache Nodes

TranSend runs Harvest object cache workers [8] on four separate nodes. Harvest suffers from three functional/performance deficiencies, two of which we resolved.

First, although a collection of Harvest caches can be treated as "siblings", by default all siblings are queried on each request, so that the cache service time would increase as the load increases even if more cache nodes were added. Therefore, for both scalability and improved fault tolerance, the Manager Stub supports managing a number of separate cache nodes as a single virtual cache, hashing the key space across the separate caches and automatically re-hashing when cache nodes are added or removed. Second, we modified Harvest to allow data to be injected into it, allowing workers (via the Worker Stub) to store post-transformed or intermediate-state data into the large virtual cache. Finally, because the interface to each cache instance is HTTP, a separate TCP connection is required for each cache request. We did not repair this deficiency due to the complexity of the Harvest code, and as a result some of the measurements reported in Section 4.4 are slightly pessimistic.

Caching in TranSend is only an optimization. All cached data can be thrown away at the cost of performance—cache instances are workers whose only job is the management of BASE data.

3.1.6 Datatype-Specific Distillers

The second group of workers is the distillers, which perform transformation and aggregation. As a result, we were able to leverage a large amount of off-the-shelf code for our distillers. We have built three parameterizable distillers for TranSend: scaling and low-pass filtering of JPEG images using the off-the-shelf *jpeg-6a* library [28], GIF-to-JPEG conversion using *netpbm* [47] followed by JPEG degradation[4], and a Perl HTML "munger" that marks up inline image references with distillation preferences, adds extra links next to distilled images so that users can retrieve the original content, and adds a "toolbar" (Figure 4) to each page that allows users to control various aspects of TranSend's operation. The user interface for TranSend is thus controlled by the HTML distiller, under the direction of the user preferences from the front end.

Each of these distillers took approximately 5-6 hours to implement, debug, and optimize. Although pathological input data occasionally causes a distiller to crash, the process-peer fault tolerance guaranteed by the SNS layer means that we don't have to worry about eliminating all such possible bugs and corner cases from the system.

4: We chose this approach after discovering that the JPEG representation is smaller and faster to operate on for most images, and produces aesthetically superior results.

3.1.7 Graphical Monitor

Our extensible Tcl/Tk [46] graphical monitor presents a unified view of the system as a *single virtual entity.* Components of the system report state information to the monitor using a multicast group, allowing multiple monitors to run at geographically dispersed locations for remote management. The monitor can page or email the system operator if a serious error occurs, for example, if it stops receiving reports from some component.

The benefits of visualization are not just cosmetic: We can immediately detect by looking at the visualization panel what state the system as a whole is in, whether any component is currently causing a bottleneck (such as cache-miss time, distillation queueing delay, interconnect), what resources the system is using, and other such figures of interest.

3.1.8 How TranSend Exploits BASE

Distinguishing ACID vs. BASE semantics in the design of service components greatly simplifies TranSend's fault-tolerance and improves its availability. Only the user-profile database is ACID; everything else exploits some aspect of BASE semantics, both in manipulating application data (i.e. Web content) and in the implementation of the system components themselves.

Stale load balancing data: The load balancing data in the Manager Stub is slightly stale between updates from the manager, which arrive a few seconds apart. The use of stale data for the load balancing and overflow decisions improves performance and helps to hide faults, since using cached data avoids communicating with the source. Timeouts are used to recover from cases where stale data causes an incorrect load balancing choice, e.g., if a request is sent to a worker that no longer exists, the request will time out and another worker will be chosen. From the standpoint of performance, as we will show in our measurements, the use of slightly stale data is not problem in practice.

Soft state: The two advantages of soft state are improved performance from avoiding (blocking) commits and trivial recovery. Transformed content is cached and can be regenerated from the original (which may be also cached).

Approximate answers: Users of TranSend request objects that are named by the object URL and the user preferences, which will be used to derive distillation parameters. However, if the system is too loaded to perform distillation, it can return a somewhat different version from the cache; if the user clicks the "Reload" button later, they will get the distilled representation they asked for if the system now has sufficient resources to perform the distillation. If the required distiller has temporarily or permanently failed, the system can return the original content. In all cases, *an approximate answer delivered quickly is more useful than the exact answer delivered slowly.*

3.2 HotBot Implementation

In this section we highlight the principal differences between the implementations of TranSend and HotBot.The original Inktomi work, which is the basis of HotBot, predates the layered model and scalable server architecture presented here and as such uses *ad hoc* rather than generalized mechanisms in some places.

Front ends and service interface: HotBot runs on a mixture of 26 single- and multiple-CPU SPARCstation server nodes, interconnected by Myrinet [40]. The HTTP front ends in HotBot run 50-80 threads per node and handle the presentation and customization of results based on user preferences and browser type. The presentation is performed using a form of "dynamic HTML" based on Tcl macros [54].

Component	TranSend	HotBot
Load balancing	Dynamic, by queue lengths at worker nodes	Static partitioning of read-only data
Application layer	Composable TACC workers	Fixed search-service application
Service layer	Worker dispatch logic, HTML/JavaScript UI	Dynamic HTML generation, HTML UI
Failure management	Centralized but fault-tolerant using process-peers	Distributed to each node
Worker placement	FE's and caches bound to their nodes	All workers bound to their nodes
User profile (ACID) database	Berkeley DB+read caches	Parallel Informix server
Caching	Harvest caches store pre- and post-transformation Web data	integrated cache of recent searches, for incremental delivery

Table 1: Main differences between TranSend and HotBot.

Load balancing: HotBot workers statically partition the search-engine database for load balancing. Thus each worker handles a subset of the database proportional to its power (some workers run on multi-CPU nodes), and every query goes to all workers in parallel.

Failure management: Unlike the workers in TranSend, HotBot worker nodes are not interchangeable, since each worker uses a local disk to store its part of the database. The original Inktomi nodes cross-mounted databases, so that there were always multiple nodes that could reach any database partition. Thus, when a node when down, other nodes would automatically take over responsibility for that data, maintaining 100% data availability with graceful degradation in performance.

Since the database partitioning distributes documents randomly and it is acceptable to lose part of the database temporarily, HotBot moved to a model in which RAID storage handles disk failures, while fast restart minimized the impact of node failures. For example, with 26 nodes the loss of one machine results in the database dropping from 54M to about 51M documents, which is still significantly larger than other search engines (such as Alta Vista at 30M).

The success of the fault management of HotBot is exemplified by the fact that during February 1997, HotBot was physically moved (from Berkeley to San Jose) without ever being down, by moving half of the cluster at a time and changing DNS resolution in the middle. Although various parts of the database were unavailable at different times during the move, the overall service was still up and useful—user feedback indicated that few people were affected by the transient changes.

User profile database: We expect commercial systems to use a real database for ACID components. HotBot uses Informix with primary/backup failover for the user profile and ad revenue tracking database, with each front end linking in an Informix SQL client. However, all other HotBot data is BASE, and as in TranSend, timeouts are used to recover from stale cluster-state data.

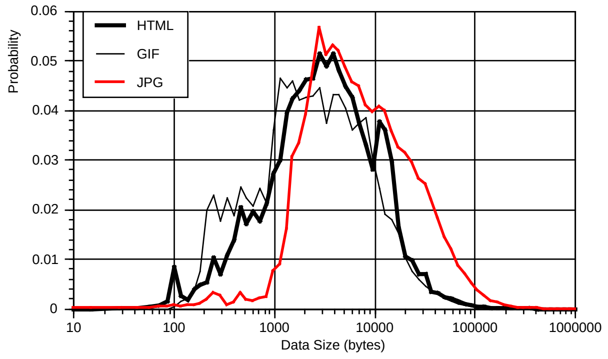

Figure 5: Distribution of content lengths for HTML, GIF, and JPEG files. The spikes to the left of the main GIF and JPEG distributions are error messages mistaken for image data, based on file name extension. Average content lengths: HTML - 5131 bytes, GIF - 3428 bytes, JPEG - 12070 bytes.

3.3 Summary

The TranSend implementation quite closely maps into the general layered architecture presented in Section 2, while the HotBot implementation differs in the use of a distributed manager, static load balancing by data partitioning, and workers that are tied to particular machines. The careful separation of responsibility into different components of the system, and the layering of components according to the architecture made the implementation complexity manageable. In the next section, we present detailed performance measurements of TranSend.

4 Measurements of the TranSend Implementation

We took measurements of TranSend using a cluster of 15 Sun SPARC Ultra-1 workstations connected by 100-Mbit switched Ethernet and isolated from external load or network traffic. For measurements requiring Internet access, the access was via a 10-Mbit switched Ethernet network connecting our workstation to the outside world. In the following subsections we analyze the size distribution and burstiness characteristics of TranSend's expected workload, describe the performance of two throughput-critical components (the cache nodes and data-transformation workers) in isolation, and report on experiments that stress TranSend's fault tolerance, responsiveness to bursts, and scalability.

4.1 HTTP Traces and the Playback Engine

Many of the performance tests are based upon HTTP trace data that we gathered from our intended user population, namely the 25,000 UC Berkeley Dialup home-IP users, up to 600 of whom may be connected via a bank of 14.4K or 28.8K modems. The modems' connection to the Internet passes through a single 10 Mbit Ethernet segment; we placed a tracing machine running an IP packet filter on this segment for a month and a half period, and unobtrusively gathered a trace of approximately 20 million (anonymized) HTTP requests. GIF, HTML, and JPEG were by far the three most common MIME types observed in our traces (50%, 22%, and 18%, respectively), and hence our three implemented distillers cover these common cases. Data for which no distiller exists is passed unmodified to the user.

Figure 5 illustrates the distribution of sizes occurring for these three MIME types. Most content access on the web is small (considerably under 1 KB) however the average byte transferred is part of large content (3-12 KB). This means that the users' modems spend most of their time transferring a few, large files. It is the goal of TranSend to eliminate this bottleneck by distilling this large content into smaller, but still useful representation; data under 1 KB is

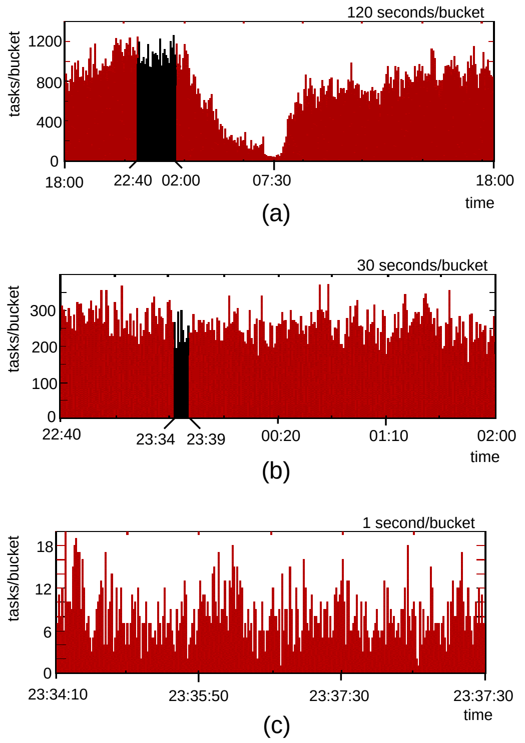

Figure 6: The number of requests per second for traced home-IP users, showing burstiness across different time scales. (a) 24 hours with 2 minute buckets, 5.8 req/s avg., 12.6 req/s max. (b) 3 hr 20 min with 30 second buckets, 5.6 req/s avg., 10.3 req/s peak. (c) 3 min 20 sec, 8.1 req/s avg., 20 req/s peak.

transferred to the client unmodified, since distillation of such small content rarely results in a size reduction.

Figure 5 also reveals a number of interesting properties of the individual data types. The GIF distribution has two plateaus - one for data sizes under 1KB (which correspond to icons, bullets, etc.) and one for data sizes over 1KB (which correspond to photos or cartoons). Our 1KB distillation threshold therefore exactly separates these two classes of data, and deals with each correctly. JPEGs do not show this same distinction - the distribution falls of rapidly under the 1KB mark.

In order to realistically stress test TranSend, we created a high performance playback engine that can replay traces at our proxy. The playback engine can generate requests at a constant (and dynamically tunable) rate, or it can faithfully play back a trace according to the timestamps in the trace file. We thus had fine-grained control over both the amount and nature of the load offered to our implementation during our experimentation.

4.2 Burstiness

Burstiness is a fundamental property of a great variety of computing systems, and can be observed across all time scales [16,26,33]. Our HTTP traces show that the offered load to our implementation will contain bursts—Figure 6 shows the request rate observed from the user base across a 24 hour, 3.5 hour, and 3.5 minute time interval. The 24 hour interval exhibits a strong 24 hour cycle that is overlaid with shorter time-scale bursts. The 3.5 hour and 3.5 minute intervals reveal finer grained bursts.

We described in Section 2.2.3 how our architecture allows an arbitrary subset of machines to be managed as an "overflow pool" during temporary but prolonged periods of high load. The overflow pool can also be used to absorb bursts on shorter time scales. We argue that there are two possible administrative avenues for managing the overflow pool:

1. Select an average desired utilization level for the dedicated worker pool. Since we can observe a daily cycle, this amounts to drawing a line across Figure 6a (i.e. picking a number of tasks/sec) such that the fraction of black under the line is the desired utilization level.
2. Select an acceptable percentage of time that the system will resort to the overflow pool. This amounts to drawing a line across Figure 6a such that the fraction of columns that cross the line is this percentage.[5]

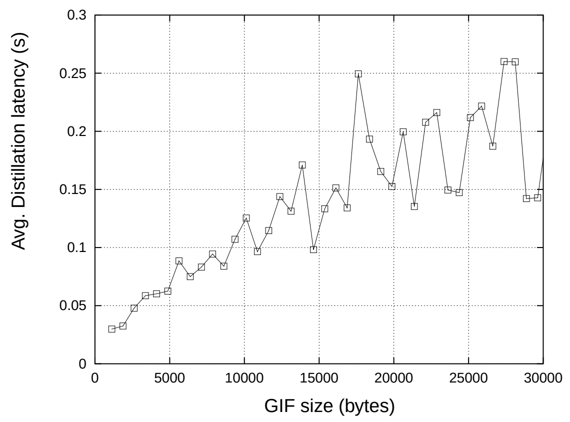

Figure 7: Average distillation latency vs. GIF size, based on GIF data gathered from the home-IP trace.

Since we have measured the average number of requests/s that a distiller of a given class can handle, the number of tasks /s that we picked (from step 1 or 2 above) dictates how many distillers will need to be in the dedicated (non-overflow) pool.

4.3 Distiller Performance

If the system is behaving well, the distillation of images is the most expensive task performed by TranSend, both in terms of the required computational cycles and end-to-end latency. We measured the performance of our distillers by timing distillation latency as a function of input data size, calculated across approximately 100,000 items from the home-IP trace file. Figure 7 shows that for the GIF distiller, there is an approximately linear relationship between distillation time and input size, although a large variation in distillation time is observed for any particular data size. The slope of this relationship is approximately 8 milliseconds per kilobyte of input. Similar results were observed for the JPEG and HTML distillers, although the HTML distiller is far more efficient.

4.4 Cache Partition Performance

In [8], a detailed performance analysis of the Harvest caching system is presented. We summarize the results here:

- The average cache hit takes 27 ms to service, including network and OS overhead, implying a maximum average service rate from each partitioned cache instance of 37 requests per second. TCP connection and tear-down overhead is attributed to 15 ms of this service time.
- 95% of all cache hits take less than 100 ms to service, implying cache hit rate has low variation.
- The miss penalty (i.e. the time to fetch data from the Internet) varies widely, from 100 ms through 100 seconds. This implies that should a cache miss occur, it will likely dominate the end-to-end latency through the system, and therefore great effort should be expended to minimize cache miss rate.

As a supplement to these results, we ran a number of cache simulations to explore the relationship between user population size, cache size, and cache hit rate, using LRU replacement. We observed that the size of the user population greatly affects the attainable hit rate. Cache hit rate increases monotonically as a function of cache size, but plateaus out at a level that is a function of the user population size. For the user population observed across the traces (approximately 8000 people over

5: Note that the utilization level cannot necessarily be predicted given a certain acceptable percentage, and vice-versa.

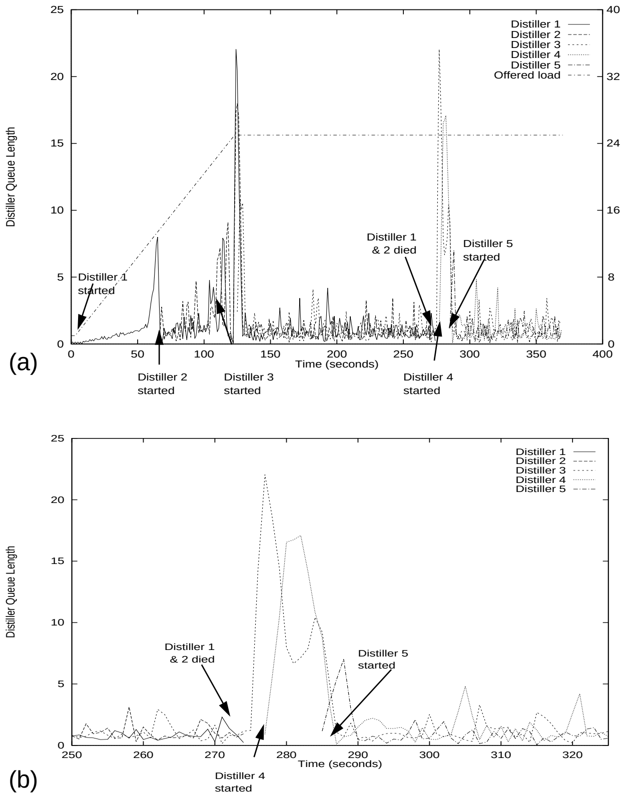

Figure 8: Distiller queue lengths observed over time as the load presented to the system fluctuates, and as distillers are manually brought down. (b) is an enlargement of (a).

the 1.5 month period), six gigabytes of cache space (in total, partitioned over all instances) gave us a hit rate of 56%. Similarly, we observed that for a given cache size, increasing the size of the user population increases the hit rate in the cache (due to an increase in locality across the users), until the point at which the sum of the users' working sets exceeds the cache size, causing the cache hit rate to fall.

From these results, we can deduce that the capacity of a single front end will be limited by the high cache miss penalties. The number of simultaneous, outstanding requests at a front end is equal to $N \times T$, where N is the number of requests arriving per second, and T is the average service time of a request. A high cache miss penalty implies that T will be large. Because two TCP connections (one between the client and front end, the other between the front end and a cache partition) and one thread context is maintained in the front end for each outstanding request, there is potentially a very large amount of state kept by the front end, implying state management and context switching overhead. As an example, for offered loads of 15 requests per second to a front end, we have observed from 150-350 outstanding requests and therefore up to 700 open TCP connections and 300 "active" thread contexts at any given time. The front end spends more than 70% of its time in the kernel (as reported by the *top* utility) under this load.

4.5 Self Tuning and Load Balancing

TranSend uses queue lengths at the distillers as a metric for load balancing. As queue lengths grow (due to increased load), the moving average of the queue length maintained by the manager starts increasing; when the average crosses a configurable threshold H, the manager spawns a new distiller to absorb the load. The threshold H maps to the greatest delay the user is willing to tolerate when the system is under high load. To allow the new distiller to stabilize the system, the spawning mechanism is disabled for D seconds; the parameter D represents a tradeoff between stability (rate of spawning and reaping distillers) and user-preceptible delay.

Figure 8(a) shows the variation in distiller queue lengths over time. The system was bootstrapped with one front end and the manager. On-demand spawning of the first distiller was observed as soon as load was offered. With increasing load, the distiller queue gradually increased until the manager decided to spawn a second distiller, which reduced the queue length of the first distiller and balanced the load across both distillers within five seconds. Continued increase in load caused a third distiller to start up, which again reduced and balanced the queue lengths within five seconds.

Figure 8(b) shows an enlarged view of the graph in Figure 8(a). Some time during the experiment, we manually killed the first two distillers, thereby causing the load on the remaining distiller to rapidly increase. The manager immediately reacted and started up a new distiller. Even after D seconds, the manager discovered that the system was overloaded and started up one more distiller, causing the load to stabilize.

As we first ran this experiment, we noticed rapid oscillations in queue lengths. Inspection revealed that since the manager stubs only periodically received distiller queue length reports, they were making load balancing decisions based on stale data. To repair this, we maintained a running estimate of the change in queue lengths at the distillers in between successive reports; these estimates were sufficient to eliminate the oscillations. The data in Figure 8 reflects the modified load balancing functionality.

4.6 Scalability

To demonstrate the scalability of the system, we needed to eliminate two bottlenecks that limit the load we could offer: the Solaris limit on the number of open file descriptors, and the bottleneck 10Mb/s Ethernet connecting our cluster to the Internet. To do this, we prepared a trace file that repeatedly requested a fixed number of JPEG images, all approximately 10KB in size, based on the distributions we observed (Section 4.1). These images would then remain resident in the cache partitions, eliminating cache miss penalty and the resulting buildup of file descriptors in the front end. We recognize that although a non-zero cache miss penalty does not introduce any additional network, stable storage, or computational burden on the system, it does result in an increase in the amount of state in the front end, which as we mentioned in Section 4.4 limits the performance of a single front end. On the other hand, by turning off caching of distilled images, we force our system to re-distill the image every time it was requested, and in that respect our measurements are pessimistic relative to the system's normal mode of operation.

Our strategy for the experiment was as follows:

1. Begin with a minimal instance of the system: one front end, one distiller, the manager, and some fixed number of cache partitions. (Since for these experiments we repeatedly requested the same subset of images, the cache was effectively not tested.)
2. Increased the offered load until some system component saturates (e.g. distiller queues grow too long, front ends cannot accept additional connections, etc.)
3. Add more resources to the system to eliminate this saturation (in many cases the system does this automatically, as when it recruits overflow nodes to run more workers), and record the amount of resources added as a function of the increase in offered load, measured in requests per second.
4. Continue until the saturated resource cannot be replenished (i.e. we run out of hardware), or until adding more of the saturated resource no longer results in a linear or close-to-linear improvement in performance.

Requests/ Second	# Front Ends	# Distillers	Element that saturated
0-24	1	1	distillers
25-47	1	2	distillers
48-72	1	3	distillers
73-87	1	4	FE Ethernet
88-91	2	4	distillers
92-112	2	5	distillers
113-135	2	6	distillers & FE Ethernet
136-159	3	7	distillers

Table 2: Results of the scalability experiment

For example, at 24 requests per second, as the offered load exceeded the capacity of the single available distiller, the manager automatically spawned one additional, and then subsequent distillers as necessary. Also, at 87 requests per second, the Ethernet segment leading into the front end saturated, requiring a new front end to be spawned.

Table 2 presents the results of this experiment. We were unable to test the system at rates higher than 159 requests per second, as all of our cluster's machines were hosting distillers, front ends, or playback engines. We did observe nearly perfectly linear growth of the system over the scaled range: a distiller can handle approximately 23 requests per second, and a 100 Mb/s Ethernet segment into a front-end can handle approximately 70 requests per second.[6] We were unable to saturate the front end, the cache partitions, or fully saturate the interior SAN during this experiment. We draw two conclusions from this result:

- Even with commodity a 100 Mb/s SAN, linear scaling is limited primarily by bandwidth into the system rather than bandwidth inside the system.
- Although we run TranSend on four SPARC 10's, a single Ultra-1 class machine would suffice to serve the entire home-IP population of UC Berkeley (25,000 users officially, over 8000 surfed during the trace).

Ultimately, the scalability of our system is limited by the shared or centralized components of the system, namely the user profile database, the manager, and the SAN. In our experience, neither the database nor the manager have ever been close to saturation. The main task of the manager (in steady state) is to accumulate load announcements from all distillers and multicast this information to the front ends. We conducted an experiment to test the capability of the manager to handle these load announcements. Nine hundred distillers were created on four machines. Each of these distillers generated a load announcement packet for the manager every half a second. The manager was easily able to handle this aggregate load of 1800 announcements per second. With each distiller capable of processing over 20 front end requests per second, the manager is computationally capable of sustaining a total number of distillers equivalent to 18000 requests per second. This number is nearly three orders of magnitude greater than the peak load ever seen on UC Berkeley's modem pool which is comparable to a modest-sized ISP. Similarly, HotBot's ACID database (parallel Informix server), used for ad revenue tracking and user profiles, can serve about 400 requests per second, significantly greater than HotBot's load.

6: We believe that TCP connection setup and processing overhead is the dominating factor. Using a more efficient TCP implementation such as Fast Sockets [52] may alleviate this limitation, although more investigation is needed.

On the other hand, SAN saturation is a potential concern for communication-intensive workloads such as TranSend's. The problem of optimizing component placement given a specific network topology, technology, and workload is an important topic for future research. As a preliminary exploration of how TranSend behaves as the SAN saturates, we repeated the scalability experiments using a 10 Mb/s switched Ethernet. As the network was driven closer to saturation, we noticed that most of our (unreliable) multicast traffic was being dropped, crippling the ability of the manager to balance load and the ability of the monitor to report system conditions.

One possible solution to this problem is the addition of a low-speed utility network to isolate control traffic from data traffic, allowing the system to more gracefully handle (and perhaps avoid) SAN saturation. Another possibility is to use a higher-performance SAN interconnect: a Myrinet [40] microbenchmark run on the HotBot implementation measured 32 MBytes/s all-pairs traffic between 40 nodes, far greater than the traffic experienced during the normal use of the system, suggesting that Myrinet will support systems of at least several tens of nodes.

5 Discussion

In previous sections we presented detailed measurements of a scalable network service implementation that confirmed the effectiveness of our layered architecture. In this section of the paper, we discuss some of the more interesting and novel aspects of our architecture, reflect on further potential applications of this research, and compare our work with others' efforts.

5.1 Extensibility: New Workers and Composition

One of our goals was to make the system easily extensible at the TACC and Service layers by making it easy to create workers and chain them together. Our HTML and JPEG distillers consist almost entirely of off-the-shelf code, and each took an afternoon to write. Debugging the pathological cases for the HTML distiller was spread out over a period of days - since the system masked transient faults by bypassing original content "around" the faulting distiller, we could only deduce the existence of bugs by noticing (using the Monitor display) that the HTML distiller had been restarted several times over a period of hours.

The other aspect of extensibility is the ease with which new services can be added by composing workers and modifying the service presentation interface. We now discuss several examples of new services in various stages of construction, indicating what must be changed in the TACC and Service layers for each. The services share the following common features, which make them amenable to implementation using our framework:

- Compute-intensive transformation or aggregation
- Computation is parallelizable with granularity of a few CPU seconds
- Substantial value added by mass customization
- Data manipulated has BASE semantics

Without loss of generality, we restrict our discussion here to services that can be implemented using the HTTP proxy model, i.e., transparent interposition of computation between Web clients and Web servers.

Keyword Filtering: the keyword filter aggregator is very simple (about 10 lines of Perl). It allows users to specify a Perl regular expression as customization preference. This regular expression is then applied to all HTML before delivery. A simple example filter is one that marks up all occurences of some keyword with large, bold, red typeface.

Bay Area Culture Page: this service retrieves scheduling information from a number of cultural pages on the web, and collates the results into a single, comprehensive calendar of upcoming events, bounded by dates stored as part of each user's profile. The service is implemented as a single aggregator in the TACC layer, and is composed with the unmodified TranSend service layer, delivering the benefits of distillation automatically. This service exploits BASE "approximate answers" semantics at the application layer: extremely general, layout-independent heuristics are used to extract scheduling information from the cultural pages. These heuristics tend to work 80-90% of the time ; the resulting 10-20% of the time, spurious results are retured to the user, which are simply ignored.

TranSend Metasearch: the metasearch service is similar to the Bay Area Culture Page in that it collates content from other sources in the Internet. This content, however, is dynamically produced - an aggregator accepts a search string from a user, queries a number of popular search engines, and collates the top results from each into a single result page. This application is not novel: commercial metasearch engines have been deployed in the past (such as the Metacrawler service [58]). However, the TranSend metasearch engine was implemented using 3 pages of Perl code in roughly 2.5 hours, and inherits scalability, fault tolerance, and high availability from the SNS layer.

Anonymous Rewebber: just as anonymous remailer chains [23] allow email authors to anonymously disseminate their content, an *anonymous rewebber network* allows web authors to anonymously publish their content. The rewebber described in [24] was implemented in one person-week using our TACC architecture. The rewebber's workers perform encryption and decryption, the user profile database maintains public key information for anonymous servers, and the cache stores decrypted versions of frequently accessed pages. Since encryption and decryption of distinct pages requested by independent users is both computationally intensive and highly parallelizable, this service is a natural fit for our architecture.

Real Web Access for PDAs and Smart Phones: We have already extended TranSend to support graphical Web browsing on the USR PalmPilot [62], a typical "thin client" device. Previous attempts to provide Web browsing on such devices have foundered on the severe limitations imposed by small screens, limited computing capability, and austere programming environments, and virtually all have fallen back to very simple text-only browsing. But the ability of our architecture to move complexity into the service workers rather than the client allows us to approach this problem from a different perspective. We have built TranSend workers that output simplified markup and scaled-down images ready to be "spoon fed" to an extremely simple browser client, given knowledge of the client's screen dimensions and font metrics. This greatly simplifies client-side code since no HTML parsing, layout, or image processing is necessary, and as a side benefit, the smaller and more efficient data representation reduces transmission time to the client.

5.2 Economic Feasibility

Given the improved quality of service provided by TranSend, an interesting question is the additional cost required to achieve this service. Given our performance data, a US$5000 Pentium Pro server should be able to support about 750 modems, or about 15,000 subscribers (assuming a 20:1 subscriber to modem ratio). Amortized over 1 year, the marginal cost per user is an amazing 25 cents/month.

If we include the savings to the ISP due to a cache hit rate of 50% or more, as we observed in our cache experiments, then we can eliminate the equivalent of 1-2 T1 lines per TranSend installation, which reduces operating costs by about US$3000 per month. Thus, we expect that the server would pay for itself in only two months. In this argument we have ignored the cost of administration, which is nontrivial, but we believe administration costs for TranSend would be minimal— we run TranSend at Berkeley with essentially no administration except for feature upgrades and bug fixes, both of which are performed without bringing the service down.

5.3 Related Work

Content transformation by proxy: Filtering and on-the-fly compression have become particularly popular for HTTP [30], whose proxy mechanism was originally intended for users behind security firewalls. The mechanism has been used to shield clients from the effects of poor (especially wireless) networks [21,35], perform filtering [67] and anonymization, and perform value-added transformations on content, including Kanji transcoding [56], Kanji-to-GIF conversion [65], and application-level stream transducing [11,59].

Fault tolerance and high availability: The Worm programs [55] are an early example of process-peer fault tolerance. Tandem Computer and others explored a related mechanism, *process-pair* fault tolerance, [5] in which a secondary (backup) process ran in parallel with the primary and maintained a mirror of the primary's internal state by processing the same message traffic as the primary, allowing it to immediately replace the primary in the event of failure. Tandem also advocated the use of simple "building blocks" to ensure high availability [5]. The Open Group SHAWS project [48] plans to build scalable highly available web servers using a fault tolerance toolkit called CORDS, but that project appears to be in its early stages.

BASE: Grapevine [7] was an important early example of trading consistency for simplicity; Bayou [19] later explored trading consistency for availability in application-specific ways, providing an operational spectrum between ACID and BASE for a distributed database. The use of soft state to provide improved performance and increase fault tolerance robustness has been well explored in the wide-area Internet, in the context of IP packet routing [37], multicast routing [18], and wireless TCP optimizations such as TCP Snoop [4]; the lessons learned in those areas strongly influenced our design philosophy for the TACC server architecture.

Load balancing and scaling: WebOS [66] and SWEB++ [2] have exploited the extensibility of client browsers via Java and JavaScript to enhance scalability of network-based services by dividing labor between the client and server. We note that our system does not preclude, and in fact benefits from, exploiting intelligence and computational resources at the client, as we do for the TranSend user interface and coarse-grained load balancing. However, as discussed in the Introduction, we expect the utility of centralized, highly-available services to continue to increase, and this cannot occur without the growth path provided by linear incremental scalability in the SNS sense.

5.4 Future Work

Our past work on adaptation via distillation [22,21] described how distillation could be dynamically tuned to match the behavior of the user's network connection, and we have successfully demonstrated adaptation to network changes by combining our original WWW proxy prototype with the Event Notification mechanisms developed by Welling and Badrinath [3], and plan to leverage these mechanisms to provide an adaptive solution for Web access from wireless clients.

We have not investigated how well our proposed architecture works for write-intensive services where the writes carry hard state, such as for commerce servers or online voting systems. We suspect, however, that if the ratio of writes to reads is small, our architecture will be adequately flexible.

The programming model for TACC services is still embryonic. We plan to develop it into a well-defined programming environment with an SDK, and encourage our colleagues to author services of their own using our system.

6 Conclusions

We proposed a layered architecture for cluster-based scalable network services. We identified challenges of cluster-based computing, and showed how our architecture addresses these challenges. The architecture is reusable: authors of new network services write and compose stateless workers that transform, aggregate, cache, and customize (TACC) Internet content, but are shielded from the software complexity of automatic scaling, high availability, and failure management. We argued that a large class of network services can get by with BASE, a weaker-than-ACID data semantics that results from the combination of trading consistency for availability and exploiting soft state for performance and failure management.

We discussed in depth the design and implementation of two cluster-based scalable network services: the TranSend distillation Web proxy and the HotBot search engine. Using extensive client traces, we conducted detailed performance measurements of TranSend. While gathering these measurements, we scaled TranSend up to 10 Ultra-1 workstations serving 159 web requests per second, and demonstrated that a single such workstation is sufficient to serve the needs of the entire 600 modem UC Berkeley home-IP dialup bank.

Since the class of cluster-based scalable network services we have identified can substantially increase the value of Internet access to end users while remaining cost-efficient to deploy and administer, we believe that cluster-based value-added network services it will become an important Internet-service paradigm.

7 Acknowledgments

This paper has benefited from the detailed and perceptive comments of our reviewers, especially our shepherd Hank Levy. We also thank Randy Katz, Eric Anderson, David Culler provided valuable feedback on the TACC model and its potential as a model for cluster programming. Ken Lutz and Eric Fraser configured and administered the test network on which the TranSend scaling experiments were performed. Cliff Frost of the UC Berkeley Data Communications and Networks Services group allowed us to collect traces on the Berkeley dialup network and has worked with us to deploy and promote TranSend within Berkeley. Undergraduate researchers Anthony Polito, Benjamin Ling, and Andrew Huang implemented various parts of TranSend's user profile database and user interface. Ian Goldberg and David Wagner helped us debug TranSend, especially through their implementation of the rewebber.

8 References

[1] E. Anderson and D. A. Patterson. *System Monitoring and Diagnostic Console for a Network of Workstations.* Second NOW Workshop at ASPLOS-VII, Boston, October 1996. http://now.cs.berkeley.edu/Sysadmin/slides/ASPLOS7.ps.gz

[2] D. Andresen, T. Yang, O. Egecioglu, O. H. Ibarra, and T. R. Smith. *Scalability Issues for High Performance Digital Libraries on the World Wide Web.*Proceedings of ADL '96, Forum on Research and Technology Advances in Digital Libraries, IEEE, Washington D.C., May 1996

[3] B. R. Badrinath and G. Welling. *A Framework for Environment Aware Mobile Applications*. International Conference on Distributed Computing Systems, May 1997 (to appear)

[4] H. Balakrishnan, S. Seshan, E. Amir, R. Katz. *Improving TCP/IP Performance over Wireless Networks.* Proeedings. of the 1st ACM Conference on Mobile Computing and Networking, Berkeley, CA, November 1995.

[5] J. F. Bartlett. *A NonStop Kernel.* Proc. 8th SOSP and Operating Systems Review 15(5), December 1981

[6] Berkeley Home IP Service FAQ. http://ack.berkeley.edu/dcns/modems/hip/hip_faq.html.

[7] A.D. Birrell et al. *Grapevine: An Exercise in Distributed Computing.* Communications of the ACM 25(4), Feb. 1984.

[8] C.M. Bowman et al. Harvest: *A Scalable, Customizable Discovery and Access System*. Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, Boulder, August 1994

[9] Tim Bray. *Measuring the Web.* Proc. WWW-5, Paris, May 1996.

[10] T. Brisco. *RFC 1764: DNS Support for Load Balancing*, April 1995.

[11] C. Brooks, M.S. Mazer, S. Meeks and J. Miller. *Application-Specific Proxy Servers as HTTP Stream Transducers.* Proc. WWW-4, Boston, May 1996. http://www.w3.org/pub/Conferences/WWW4/Papers/56.

[12] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz and K. J. Worrell. *A Hierarchical Internet Object Cache*. Proceedings of the 1996 Usenix Annual Technical Conference, 153-163, January 1996.

[13] D.Clark. *Policy Routing in Internet Protocols.* Internet Request for Comments 1102, May 1989,

[14] Cisco Systems. *Local Director*. http://www.cisco.com/warp/public/751/lodir/index.html.

[15] F. J. Corbató and V. A. Vyssotsky. Introduction and Overview of the Multics System. *AFIPS Conference Proceedings*, 27, 185-196, (1965 Fall Joint Computer Conference), 1965. http://www.lilli.com/fjcc1.html

[16] M.E. Crovella and A. Bestavros. *Explaining World Wide Web Traffic Self-Similarity*. Tech Rep. TR-95-015, Computer Science Department, Boston University, October 1995.

[17] P. B. Danzig, R. S. Hall and M. F. Schwartz. *A Case for Caching File Objects Inside Internetworks*. Proceedings of SIGCOMM '93. 239-248, September 1993.

[18] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei. *An Architecture for Wide-Area Multicast Routing.* Proceedings of SIGCOMM '94, University College London, London, U.K., September 1994.

[19] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, B. Welch. *The Bayou Architecture: Support for Data Sharing Among Mobile Users.*

[20] Firefly Network, Inc. http://www.ffly.com

[21] A. Fox and E. A. Brewer. *Reducing WWW Latency and Bandwidth Requirements via Real-Time Distillation.* Proc. WWW-5, Paris, May 1996.

[22] A. Fox, S. D. Gribble, E. Brewer and E. Amir. *Adapting to Network and Client Variation Via On-Demand Dynamic Distillation.* Proceedings of ASPLOS-VII, Boston, October 1996.

[23] Ian Goldberg, David Wagner, and Eric Brewer. *Privacy-enhancing Technologies for the Internet*. Proc. of IEEE Spring COMPCON, 1997

[24] Ian Goldberg and David Wagner. *TAZ Servers and the Rewebber Network: Enabling Anonymous Publishing on the World Wide Web*. Un-

published manuscript, May 1997, available at http://www.cs.berkeley.edu/~daw/cs268/.

[25] J. Gray. *The Transaction Concept: Virtues and Limitations*. Proceedings of VLDB. Cannes, France, September 1981, 144-154.

[26] S.D. Gribble, G.S. Manku, and E. Brewer. *Self-Similarity in File-Systems: Measurement and Applications*. Unpublished, available at http://www.cs.berkeley.edu/~gribble/papers/papers.html

[27] T.R. Halfhill. *Inside the Web PC*. Byte Magazine, March 1996.

[28] Independent JPEG Group. *jpeg6a* library.

[29] Inktomi Corporation: *The Inktomi Technology Behind HotBot*. May 1996. http://www.inktomi.com/whitepap.html.

[30] Internet Engineering Task Force. *Hypertext Transfer Protocol—HTTP 1.1*. RFC 2068, March 1, 1997.

[31] P. Keleher, A. Cox, and W. Zwaenepoel. *Lazy Release Consistency for Software Distributed Shared Memory*. Proceedings of the 19th Annual Symposium on Computer Architecture. May, 1992.

[32] P. Keleher, A. Cox, S. Swarkadas, and W. Zwaenepoel. *TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems*. Proceedings of the 1994 Winter USENIX Conference, January, 1994.

[33] W. Leland, M.S. Taqqu, W. Willinger, and D.V. Wilson. *On the Self-Similar Nature of Ethernet Traffic (extended version).* IEEE/ACM Transactions on Network v2, February 1994.

[34] K. Li. *Shared Virtual Memory on Loosely Coupled Microprocessors*. PhD Thesis, Yale University, September 1986.

[35] M. Liljeberg et al. *Enhanced Services for World Wide Web in Mobile WAN Environment*. University of Helsinki CS Technical Report No. C-1996-28, April 1996.

[36] Bruce A. Mah. *An Empirical Model of HTTP Network Traffic.* Proc. INFOCOM 97, Kobe, Japan, April 1997.

[37] J. McQuillan, I Richer, E. Rosen. *The New Routing Algorithm for the ARPANET*. IEEE Transactions on Communications COM-28, No. 5, pp. 711-719 , May 1980.

[38] M. Meeker and C. DePuy. *The Internet Report*. Morgan Stanley Equity Research, April 1996. http://www.mas.com/misc/inet/morganh.html

[39] P.V. Mockapetris and K.J. Dunlap. *Development of the Domain Name System*. ACM SIGCOMM Computer Communication Review, 1988.

[40] Myricom Inc. *Myrinet: A Gigabit Per Second Local Area Network*. IEEE-Micro, Vol.15, No.1, February 1995, pp. 29-36.

[41] National Laboratory for Applied Network Research. *The Squid Internet Object Cache*. http://squid.nlanr.net.

[42] National Aeronautics and Space Administration. *The Mars Pathfinder Mission Home Page*. http://mpfwww.jpl.nasa.gov/default1.html.

[43] Netscape Corporation. *JavaScript Authoring Guide*. http://home.netscape.com/eng/mozilla/Gold/handbook/javascript.

[44] Netscape Communications Corp. Persistent Client-State Http Cookies: Preliminary Specification. http://home.netscape.com/newsref/std/cookie_spec.html

[45] Nokia Communicator 9000 Press Release. Available at http://www.club.nokia.com/support/9000/press.html.

[46] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[47] J. Poskanzer. *Netpbm release 7.* ftp://wuarchive.wustl.edu/graphics/graphics/packages/NetPBM, 1993.

[48] Open Group Research Institute. *Scalable Highly Available Web Server Project (SHAWS).* http://www.osf.org/RI/PubProjPgs/SFTWWW.htm

[49] Eric S. Raymond, ed. *The New Hackers' Dictionary*. Cambridge, MA: MIT Press, 1991. Also http://www.ccil.org/jargon/jargon.html.

[50] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, J. Ried. *GroupLens: An Open Architecture for Collaborative Filtering of Netnews*. Proceedings of 1994 Conference on Computer Supported Cooperative Work, Chapel Hill, NC.

[51] D. M. Ritchie and K. Thompson. *The UNIX Time-Sharing System.* CACM 17(7), July 1974.

[52] S. H. Rodrigues and T. E. Anderson. *High-Performance Local-Area Communication Using Fast Sockets.* Proc. 1997 Winter USENIX, Anaheim, CA.

[53] Jacques A.J. Roufs. *Perceptual Image Quality: Concept and Measurement.* Philips Journal of Research, 47:3-14, 1992.

[54] A. Sah, K. E. Brown and E. Brewer. *Programming the Internet from the Server-Side with Tcl and Audience1*. Proceedings of Tcl96, July 1996.

[55] J. F. Shoch and J. A. Hupp. *The "Worm" Programs—Early Experience with a Distributed System*. CACM 25(3):172-180, March 1982.

[56] Y. Sato. DeleGate Server. Documentation available at http://www.aubg.edu:8080/cii/src/delegate3.0.17/doc/Manual.txt.

[57] B. Schilit and T. Bickmore. *Digestor: Device-Independent Access to the World Wide Web.* Proc. WWW-6, Santa Clara, CA, April 1997.

[58] E. Selberg, O. Etzioni and G. Lauckhart. *Metacrawler: About Our Service*. http://www.metacrawler.com/about.html.

[59] M.A. Schickler, M.S. Mazer, and C. Brooks. *Pan-Browser Support for Annotations and Other Meta-Information on the World Wide Web.* Proc. WWW-5, Paris, May 1996. http://www5conf.inria.fr/fich_html/papers/P15/Overview.html

[60] SingNet (Singapore ISP). *Heretical Caching Effort for SingNet Customers*. http://www.singnet.com.sg/cache/proxy

[61] N. Smith. *The UK National Web Cache - The State of the Art. Proc*. WWW-5, paris, May 1996. http://www5conf.inria.fr/fich_html/papers/P45/Overview.html

[62] US Robotics Palm Pilot home page - http://www.usr.com/palm/.

[63] C. Waldspurger and W. Weihl. *Lottery Scheduling: Flexible Proportional Share Resource Management*. Proceedings of the First OSDI, November 1994.

[64] Yahoo!, Inc. http://www.yahoo.com

[65] Kao-Ping Yee. Shoduoka Mediator Service. http://www.lfw.org/shodouka.

[66] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. *Using Smart Clients to Build Scalable Services.*

[67] B. Zenel. *A Proxy Based Filtering Mechanism for the Mobile Environment*. Ph.D. Thesis Proposal, Department of Computer Science, Columbia University, March 1996.