System Support for Scalable and Fault Tolerant Internet Services

Submission No. **12421**

Yatin Chawathe and Eric A. Brewer

Computer Science Division, University of California at Berkeley

{yatin, brewer}@cs.berkeley.edu

February 16, 1999

Abstract

Over the past few years, we have seen the proliferation of Internet-based services ranging from search engines and map services to video-on-demand servers. All of these kinds of services need to be able to provide certain guarantees of availability and scalability to their users. With millions of users on the Internet today, these services must have the capacity to handle a large number of clients and remain available even in the face of extremely high load.

In this work, we present a generic architecture for supporting such Internet applications. We provide a substrate for Scalable Network Services (SNS), on top of which application developers can design their services without worrying about the details of service management. We back our design with three real-world services: a web distillation proxy, a proxy-based web-browser for PDAs, and an MBone archive server.

1 Introduction

The explosive growth of the World-Wide Web has spawned a number of network-based services such as map services [38], search engines [35, 22], anonymizing web-proxies [8], and directory services [50, 17]. Similarly, the expansion of the Internet multicast backbone (MBone) [19, 20] has created a demand for services such as video-on-demand servers [11, 2] and real-time audio/video gateways [4].

As these types of services get more popular, they will be required to provide guarantees with respect to their level of service. With millions of users on the Internet, these services must have the capacity to handle a large number of clients and remain available even in the face of extremely high load. We identify two critical issues that any Internet service must address in its design:

System Scalability: The service must be able to scale incrementally with the offered load. As load begins to exceed the service's capacity, it should be able to adjust to the load simply by adding new hardware. In addition, to ensure linear scalability, the service must distribute its load across all of its resources. Further, the service must be capable of absorbing any peaks in the load. Any load balancing policies in the service must try to ensure that clients receive the best possible response latencies from the service.

Availability and Fault Tolerance: As much as possible, the service must remain available in the face of network failure, system crashes, and hardware upgrades. A key component of service availability is the ability to maintain a well-known and reliable name that clients can use to locate the service. In addition, a service must also ensure that, for example, faults in its internal operation do not take down the entire service. The service may try to completely mask out failures in the system from end-clients, or rely on *smart clients* [51] to participate in its fault tolerance strategies.

In [28], we argued that clusters of workstations [6] are ideal for most Internet ser-

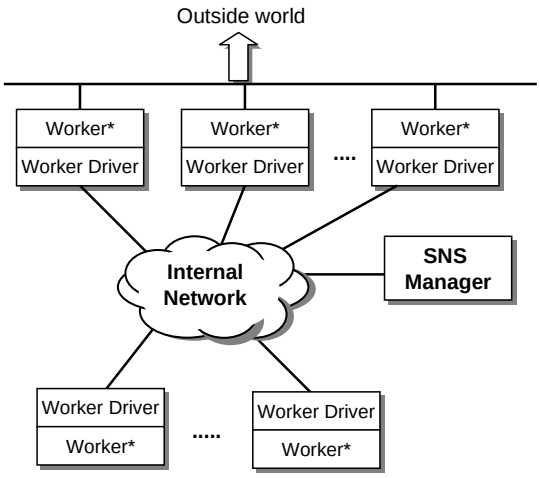
vice requirements. Internet services tend to be highly parallel with many simultaneous independent tasks that can easily be distributed across a cluster of commodity PCs. Clusters have the advantage of the ability to grow incrementally by simply adding more PCs as the workload grows. Moreover, the natural redundancy in a cluster of machines is excellent for masking out failures in the service and ensuring high availability.

However, designing and managing a system that is distributed across a cluster poses a number of challenges. In this paper, we describe an architecture substrate for building cluster-based systems that can satisfy the scalability and availability requirements. We isolate the key functions of scalability, load balancing, and fault tolerance into a reusable substrate, called the "Scalable Network Services" (SNS) architecture. Service developers build their systems on top of the SNS, and interact with the SNS through a narrow and well-defined interface. The SNS substrate allows application writers to concentrate on the semantics of their application rather than worrying about the details of service management. We demonstrate the effectiveness of our architecture through examples of three different services.

The rest of this paper is organized as follows. Section 2 explores a model for Internet services and the assumptions inherent in the model. We describe our design and implementation in Section 3. Section 4 presents some example applications based on this model, and Section 5 performs an evaluation of the system. Finally, we describe related and future work and present our conclusions.

2 A Reusable "Scalable Network Services" Architecture

Clusters are extremely amenable to building Internet services, primarily because of their advantages over single large machines such as SMPs in terms of providing the scalability and availability guarantees required by such services. Moreover, clusters can be built from commodity PCs that have far better cost/performance benefits than



***** – Service developers customize these components; all other components are part of the SNS

Figure 1: **The SNS Architecture**.

SMPs.

To make best use of a cluster of workstations, we have defined an architecture based on independent composable modules. We have carefully designed the service architecture to separate the service support layer from the rest of the system. This allows the rest of the system to be isolated from having to deal with the SNS (Scalable Network Services) issues described in Section 1.

The SNS subsystem assumes that the application is distributed across a number of service-specific modules, called *workers*. Workers implement the actual semantics of the service. Workers can be of two types:

- A back end engine that simply receives tasks from the rest of the system, operates on those tasks, and returns a result.
- A more active component that possibly interfaces to the outside world. Such

a worker receives requests or commands from clients, and either handles them directly, or "shepherds" them to other workers depending upon the requirements of the specific service and the specific request.

For example, a web-based service can implement a "front end" worker that actively listens for HTTP [36] requests from end-clients, and acts as a control center for the service by forwarding requests to other workers, waiting for responses, and shuttling the responses back to the client. The front end may rely on back end workers for acting on the request. For example, the front end of a map service receives requests that contain a postal address, and it is required to return a map that centers around the address. The front end dispatches this request to a specialized map worker that converts the postal address into map coordinates, retrieve the corresponding map, and return a map image to the front end.

Some other workers may communicate directly with their clients; for example, a worker in a video archive server can stream video data directly to its clients, instead of shuttling it through a front end.

The simplest form of worker has no dependencies, and can be executed on any node in the system. This, however, is not enough in many of scenarios. Some workers may rely on specific aspects of the system that are available only on one or a subset of the nodes in the cluster. For example, a front end worker for an HTTP service has an affinity for the IP address (or DNS name) associated with the service. Similarly, a video archive worker has affinity for the disk on which the archived session has been stored. Other workers may depend on specialized hardware that is available only on certain nodes. Worker affinity introduces additional factors in the SNS subsystem's decision to place workers on nodes.

In addition to worker modules, the SNS architecture relies on a central management entity, the *SNS Manager*, to coordinate all the workers in the system by balancing load across workers, spawning new workers on demand, and ensuring that the system can recover from faults. Figure 1 shows a block diagram of the SNS. The SNS Manager is

the core of the SNS substrate. *Worker Drivers* are libraries that are attached to workers, and implement the SNS functionality at the workers. Applications customize individual workers to implement their service.

2.1 Service Assumptions

Workers communicate with each other through *tasks*. A worker sends a task to another worker, which operates on the task, and returns the result back to the originator. The SNS subsystem makes certain assumptions about the semantics of workers and their tasks:

- Workers are grouped into classes. All workers that are capable of performing the same tasks are grouped into the same class. Within a given class, workers are identical and interchangeable, that is, a task destined for the class may be handled by any worker in that class.
- Although workers may be tied to specific nodes in the cluster, tasks themselves have no dependencies.
- Workers have a simple serial interface for accepting tasks. A worker (the *originator*) can send a task to another worker (the *consumer*) by specifying the class and inputs for the task, and expects a result back from the consumer.
- The time required to handle a task and return a response is small relative to the acceptable latency from the system.
- Tasks are assumed to be atomic and restartable. The easiest way of implementing this assumption is to make workers completely stateless. This ensures that no adverse effects occur even if a task is executed multiple times, or a worker fails halfway through its operation. A worker can build stronger semantics on top of these basic requirements.
- The SNS subsystem can detect task failure and take appropriate action to overcome the failure. Fault detection is either in the form of timeouts, or a preemptive

signal that notifies the SNS subsystem of failure.

These assumptions simplify the design of the SNS layer and enable us to effectively provide the availability and fault tolerance guarantees promised by the service. Any service that needs stricter semantics than these can customize its workers accordingly.

2.2 Centralized Resource Management

The SNS Manager has been intentionally designed to be a centralized entity rather than distributing its functionality across the entire cluster. If we can ensure the fault tolerance of the Manager, and make sure that it is not a performance bottleneck, a centralized design makes it easier to reason about and implement the various policies for load balancing and resource location/creation. We classify the functions of the SNS Manager into resource location, load balancing, handling bursts and scaling with load, and fault tolerance.

2.2.1 Resource Location

The main task of the SNS Manager is to act as a name service for the rest of the system. Workers contact the SNS Manager to locate other resources. The SNS Manager maintains a table of all available resources in the system. If it finds an appropriate resource, it returns the location of the resource to the worker. Workers cache this information, so they do not have to contact the manager each time.

2.2.2 Load Balancing

Load across the various workers is handled by a centralized policy in the SNS Manager. Although the final load balancing decision is distributed across all workers, the load gathering and reporting mechanisms are centralized in the manager. Based on load reports from each worker, the manager generates load balancing hints which are announced to the rest of the system. Workers use these hints to make local decisions regarding their choice of consumers for tasks.

2.2.3 Handling Bursts and Incremental Growth

As the load on the system grows, the SNS Manager must pull in idle nodes, and start new workers to deal with excess load. Moreover, the system must be able to deal with bursts in load. Network traffic has been shown to be bursty at varying time scales [37, 18, 33], and a network service must be able to handle such bursts. We deal with short traffic bursts by replicating workers and directing tasks across all replicated workers for greater throughput.

More prolonged bursts, however, can result in stressing the system's resources. For example, after the landing of the Pathfinder on Mars, NASA's web site experienced over 220 million hits in a four day period [43] which excessively overloaded their web server. A service may choose to dedicate enough nodes to the cluster to handle the worst case load, but in normal usage this will lead to a waste of resources.

Our design, instead, supports the notion of a pool of overflow nodes that are not dedicated to the system, and are recruited by the system only in the event of extreme bursts. These nodes can be machines that are usually used for other purposes (such as people's workstations), or machines that the service can "rent" on a pay-per-use basis from another service provider. When the SNS Manager runs out of dedicated machines to handle the system load, it may rein in idle machines from the overflow pool and launch workers on those machines. As soon as the load on the system subsides, the overflow machines are released.

2.2.4 Fault Tolerance and Availability

An SNS must guarantee its ability to withstand failures and faults in the system. We rely on two basic design principles for achieving reliance against system failures: soft state and timeouts. The concept of soft state has been used in a number of systems to achieve better reliability. For example, many networking systems for routing protocols [39, 19] and performance optimization [9] rely on soft state, periodic update mechanisms to keep their tables up-to-date. By explicitly designing our system to rely

entirely on soft state, we have managed to simplify our crash recovery process. No explicit crash recovery is required—recovery is built into the normal functioning of the protocol machinery.

We use a fault tolerance strategy that we call *starfish fault tolerance* to detect and recover from faults in the system. The components in the SNS system monitor each other, and when a component fails, one of its peers restarts it, possibly on a different node. Cached stale state carries the rest of the components through the fault while the failed component is being restarted; the new component gradually rebuilds its state and the system returns to a normal mode of operation. We describe our fault tolerance strategies in greater detail in Section 3.4.

In addition to reliance on soft state, the system also uses *timeouts and retries* to handle faults that cannot otherwise be detected. If a task cannot be completed because of some failure in the system, the SNS subsystem will retry the task, possibly on a different node. To avoid tying up resources due to a systematic fault, any task is retried only a limited number of times before the SNS subsystem gives up and relies on the application to explicitly deal with the error.

3 SNS Implementation

This section focuses on the details of the design of the SNS components and describes the implementation of the system. The SNS functionality is encompassed in the SNS Manager and in the Worker Driver which is a library attached to each worker in the system. This permits the worker implementations to be largely independent and ignorant of the SNS subsystem.

Task originators locate an appropriate consumer for their task with the help of the SNS Manager; they cache consumer locations to avoid having to contact the SNS Manager for each task. The worker driver at the consumer maintains a queue of pending tasks; tasks are taken off the queue serially and handed to the worker for processing. The queue at the worker driver may be a simple first-in-first-out queue, or may use a

more sophisticated queuing policy such as priority queues. Priority queues are useful for categorizing clients or tasks into levels of service and servicing tasks for preferred clients before handling requests from other clients [32].

We divide the functionality of the SNS subsystem into three parts: resource location, load balancing, and fault tolerance, and look at each of these in detail.

3.1 Resource location

The resource location problem for the SNS encompasses three issues:

- how do clients locate the service,
- how do workers locate the centralized SNS Manager, and
- how do workers locate each other.

3.1.1 Locating the Service

A service must present to the world a well-known and reliable name that clients can use to contact it. The mechanism used to locate the service depends largely on the service itself. An application must customize the service location component according to its interface and the protocols used to communicate with the rest of the world. In this section, we present a number of different alternatives that a service may employ.

Web-based services must use Uniform Resource Locators (URLs) as the primary service location mechanism. This has an inherent fault tolerance flaw in that the location of the service is hard-coded into its name, thereby making it difficult to move the service to a new location when a fault occurs. A number of techniques have been developed to address this problem. DNS round-robin [13] and specialized routers [16, 5] can be used to hide a number of replicated and/or distributed servers behind a single logical name. Smart clients [51] can assist the service in transparent service location in the presence of faults by relying on browser support through Java and JavaScript.

Services that rely on the IP multicast delivery model can use multicast as a level of indirection to locate the service [3]. Service front ends actively listen on a well-known

multicast channel; all clients that need to access the service broadcast their requests on the multicast channel.

3.1.2 Locating the SNS Manager

The SNS Manager is the centralized resource manager of the system. To avoid expensive fault recovery when the SNS Manager crashes, we use multicast as a level of indirection for locating the manager. The SNS Manager periodically beacons its existence on a well-known multicast address. All other components in the system subscribe to this multicast group and listen for beacons from the manager. This mechanism relieves the components from having to perform explicit fault recovery when the SNS Manager crashes and comes back up on a different location. The workers simply hear a new beacon from the manager, and redirect their communication to the new location.

3.1.3 Locating workers

As described in Section 2.1, all workers are grouped into classes based on their semantics and the operations that they perform. Each class of workers is assigned a unique name. We use a naming scheme that is loosely based on MIME types [12]. The name consists of a sequence of words separated by slashes. For example, all transformation workers that compress GIF images may be called *compress/image/gif*. Component names may reflect specific instances, or sessions that the component belongs to. For instance, a specific instance of a front end worker is identified as *frontend/<ip-address>/<port>*. Similarly, a transformation worker for converting MBone video on the fly is tied to a specific multicast session, and its name must reflect that (*transform/video/h261/<multicast-session-address>*). Applications must pick names for their workers in a manner that make sense for the service being implemented.

When a worker locates the SNS Manager, it registers itself with the manager. This registration message contains the name of the worker's class, which the SNS Manager uses to build a table of all active components in the system. When an originator of a

task needs to locate a consumer, it sends a *Find* message to the manager that contains a name that identifies the consumer. The manager performs a lookup in its *active components table* and returns a list of all workers that match the specification. The originator caches this list in a *local active components cache*, so as to avoid contacting the SNS Manager every time.

3.2 On-demand Launching of Workers

The SNS Manager may not be able to locate an appropriate worker in response to a *Find* message. The manager must then either start a new worker, or respond with a *NotFound* message.

The manager maintains a table of all possible workers in the system. This "launch table" contains a list of all workers that can be part of the service and the executables associated with each worker. The manager builds the table from a file on startup (or whenever the file changes). To start a new worker, the manager consults the launch table and attempts to start the worker on a lightly loaded node in the cluster. When this worker registers with the manager, the manager returns the worker's location in response to the original *Find* message.

To spawn workers, we use a simple scheme where the manager relies on the load balancing information that it gathers from other workers (see Section 3.3) and selects the least loaded node. We believe that this simple policy is sufficient; more fine grained load balancing is performed at the task level by the SNS Manager. The spawning policy must take into account any affinities that a worker might have for specific nodes in the cluster. The SNS Manager's launch table keeps track of any restrictions on worker launch. The launch strategy notes these restrictions before spawning a new worker.

If the manager is unable to locate or spawn a new worker, it returns a *NotFound* message. To ensure that a worker does not keep asking the manager for the same component over and over, we cache the *NotFound* message in a separate negative cache at the originator. If a component name exists in the negative cache, the originator

desists from contacting the manager, and reports an error to the application. If the launch table on the SNS Manager is updated, the manager requests all workers to flush their negative caches, so no stale information in the cache causes denial of service.

3.3 Load Balancing and Scalability

A good service implementation must smooth out the load on the service across all its components. We divide the load balancing policy into three tasks: load measurement, communication, and resource allocation. The SNS Manager gathers load samples from all workers, and uses this information to make intelligent load balancing decisions.

3.3.1 Load measurement

CPU usage is the most common and often the most dynamic load metric. However, each application may have its own notion of load. Hence, we avoid building load semantics into the SNS layer, and instead provide hooks for service developers to tailor the load metrics to each worker's needs. All tasks destined for a worker arrive at the worker driver, which invokes a method in the worker to perform a cursory analysis of the task and estimate the approximate "load" that would be generated by the task. For a number of Internet service applications, tasks are very short and the fine granularity of estimating the actual load per task is unimportant; for such tasks we use a default estimate of one unit of load per task. This makes the load equivalent to the number of pending tasks at the worker.

The worker driver maintains a running statistic of the current load; when a new task arrives, or when a pending task is finished, the driver updates its load statistic. Periodically, it reports this information to the SNS Manager. Each load report contains an average of the load statistics gathered over the past reporting interval. The reporting interval itself is configurable on a per-worker basis and represents a tradeoff between system response time to load fluctuations versus the amount of network bandwidth consumed by the load reports.

3.3.2 Balancing the load

The SNS Manager centralizes all load information. It maintains a history of load reports from workers thereby allowing it to smooth out any small transient bursts of load. As mentioned in Section 3.1.2, the SNS Manager periodically beacons its existence to the rest of the system on a well-known multicast channel. The manager distributes the aggregate load information to all other components by piggy-backing it on these existence beacons. Workers cache this load information in their *local active components cache*, thus allowing them to perform the final load balancing decision entirely locally.

In [40], the authors analyze various load balancing policies for distributed systems that use periodic load reporting. They use theoretical models to compare the effects of three policies: choosing a worker at random, choosing d workers at random and selecting the least loaded of those d workers, and selecting the worker with the absolute least load. As a rule of thumb, they found that selecting the less loaded of two randomly chosen workers worked well. Our implementation takes into account the load on each worker, while performing the load balancing. We use a scheme based on lottery scheduling [49]. Each worker is assigned tickets that are inversely proportional to the load on the worker.

If tasks are generated at a much faster rate than the beaconing interval for load reports, stale load balancing information at the workers can result in incorrect decisions. We instead add feedback to the system by having each worker maintain a correction factor that corresponds to the tasks that it has already dispatched to a consumer. Every time an up-to-date load report is received, the correction factor is reset. The worker adds this correction factor to the cached load balancing hints when it tries to pick a consumer. In practice, we have found that this policy works quite well in our example applications. Section 5.1 shows the effectiveness of the policy.

3.3.3 Auto-launch for scalability

A crucial aspect of a load balancing protocol must include the ability to gradually scale the system as the load increases. The SNS Manager periodically checks the aggregate load statistics for workers in a particular class, and launches a new worker if the load gets too high. The same mechanisms are used to auto-launch a new worker, as to launch the first on-demand worker.

Even after a new worker has started, it might be a while before the load stabilizes across the new and old workers. To avoid further false triggers of the auto-launch mechanism, we disable this feature for workers of that class for a short duration after the new worker is created. The threshold for launching workers maps to the greatest delay the client is willing to tolerate when the system is under heavy load, while the time for which the spawning mechanism is disabled once a new worker is launched represents a tradeoff between stability (rate of spawning and reaping workers) and client-perceptible delay.

3.3.4 The overflow pool

As described in Section 2.2.3, the SNS subsystem supports the notion of an overflow pool to handle prolonged bursts in traffic that cannot be dealt with by the dedicated nodes. When the SNS Manager runs out of dedicated nodes to handle a burst, it tries to harness an unloaded node in the overflow pool. The overflow nodes behave no differently than any other nodes in the system; workers do not notice any difference between being run on a dedicated node versus being started on an overflow node. When the burst dies down, the overflow workers may be reaped and the machine can be returned to its original task.

The SNS Manager uses a priority scheme to implement the overflow pool. Dedicated nodes are assigned the highest priority level, while overflow nodes have lower priorities. When the auto-launch mechanism kicks in to handle a burst of traffic, it tries to locate nodes according to their priorities. Higher priority nodes (dedicated nodes)

are selected over lower priority ones. Thus overflow nodes are used only when all dedicated nodes have been used up. Similarly, when the burst subsides, the SNS Manager first reaps workers from lower priority nodes, thereby releasing nodes from the overflow cluster before releasing any dedicated node.

3.4 Availability and Fault Tolerance

We leverage the natural redundancy of clusters to achieve availability and mask transient failures in the system: each node in the cluster in independent of the others with its own busses, power supply, disks, etc. Fault tolerance involves three tasks: detecting that a fault has occurred, bypassing the fault, and recovering any loss that might have occurred as a result of the fault. We rely on two main principles in our handling of failures: component restart, and timeouts. We present a mode of fault tolerance that we call *starfish fault tolerance*.

3.4.1 Starfish Fault Tolerance

Most traditional systems have relied on *primary/secondary fault tolerance* [5, 10] to ensure availability of the system. This involves replicating each component: one of the clones acts as the primary process and participates in the system, while the other is the secondary and simply monitors the primary to mirror its state. If the primary crashes or fails in any other way, the secondary takes over the job of the primary and creates its own secondary. Thus each component needs to be replicated in this mode to achieve fault tolerance.

Instead of a primary/secondary configuration, all components in the SNS architecture are fault tolerance "peers" and each monitors the others to ensure that all necessary components are always available. There is no unnecessary replication. The SNS Manager is the centralized fault tolerance agent; it acts as a peer for all other components, monitoring and restarting them in the event of a crash. Similarly, workers monitor the SNS Manager and try to restart it upon failure. Just as a starfish can regenerate all its

arms even if it is cut into pieces, as long as there is at least one component active in the system, it can eventually regenerate all other components.

3.4.2 Worker Failure

To detect worker failure, the SNS Manager relies on the periodic load reports it receives from the workers. When the reports cease (or if the connection between the worker and the manager is severed), it assumes a fault in the worker, and notifies all other components. A crash may result in loss of tasks pending in the worker driver queue. Since tasks are restartable, the system can simply retry the task upon failure.

For passive workers that only respond to requests from other workers, the system does not need to perform any recovery to restart the worker. The on-demand launch mechanism will restart the worker if and when it is needed. Workers that actively operate even in the absence of tasks may need to be restarted immediately after failure (for example, an ongoing video session, or an HTTP front end whose job is to wait for connections from end-client). Such workers rely on a special recovery mode in the SNS Manager: when the worker registers with the SNS Manager, it includes in the registration message a special "restart-on-death" flag. When the SNS Manager detects the death of such a worker, instead of the usual on-demand recovery, it immediately launches a new one.

The fault recovery problem is further complicated by the fact that some workers might have affinity for specific nodes. The SNS Manager must restrict the respawning to only those nodes. In the extreme case of affinity for a single node, the manager may simply be unable to restart the worker if the node itself has crashed. This scenario violates the redundancy principle of clusters, and there is nothing the manager can do until the node comes back up.

In many situations, we may be able to hide the affinity through other means. Affinity to IP addresses can be hidden via DNS round-robin [13] or specialized routers [16, 5]. Disk affinity may be reduced by using a shared disk across multiple nodes.

3.4.3 SNS Manager Failure

In Section 2.2, we emphasized the use of a centralized SNS Manager to simplify the design of the system. In this section, we show how our strategies ensure that the centralized manager does not end up becoming a fault tolerance nightmare.

All workers in the system collaborate to monitor the SNS Manager. The existence beacons from the manager are proof of the manager's existence. When the beacons are lost, a distributed restart algorithm kicks in. To ensure that multiple workers do not simultaneously start a new manager, we use a random backoff timer scheme. Each worker starts a random timer; the worker whose timer expires first wins the arbitration process and restarts the manager. Before doing so, it multicasts an *AmRestarting* message to the rest of the system. This allows the other components to back off their timers.

In spite of this arbitration algorithm, it is possible that two managers get spawned. The managers simply detect this and one of them commits suicide. Each manager includes a random number in its beacon packets. When a manager detects beacons from another manager, it compares the random number in the beacon; the manager with the larger random number wins. In the rare event of a tie, we use the IP address and port number associated with the managers as a tie breaker.

Associated with SNS fault recovery, we must address two issues: the ability of the system to continue to function in the event of a crash, and the ability of the manager to recreate its centralized state after it recovers from the crash. The first problem is alleviated by the local active components cache maintained by each worker. While the manager is being restarted, the rest of the components can continue to communicate with each other using this cached information. The one ability that is lost while the manager is dead is the worker-launch mechanism. But, we expect this to be rare, and usually the manager will be restarted within a few seconds and the ability is regained.

The use of soft state updates ensures that crash recovery is relatively simple. When the manager restarts and begins beaconing again, all components detect the new bea-

cons and re-register with the new manager. With time, all components report their load information to the new manager and the manager can regain its lost load balancing information.

If a network partition splits the system into two, each half of the system will grow into a full-fledged SNS with its own manager and workers. Once the partition is removed, the two systems coalesce together, and one of the managers kills itself, returning the system to a stable state.

3.4.4 Timeouts and Retries

The final mechanism to deal with faults is the use of timeouts throughout the system. Timeouts are a last resort in some situations where no other fault tolerance mechanism is useful. Workers use timeouts to bound the amount of time they must wait for a consumer to return the result of a task, before retrying the task on a different worker. Similarly, if the SNS Manager does not hear back from a newly launched worker, it assumes the launch failed and retries. Software faults in a worker's application code may cause it to spin in an endless loop while handling a task. To avoid eating up system resources while the worker is effectively useless, the worker driver uses a timer to wait for the task to complete and kills the worker if the timer expires.

To ensure that a systematic failure does not cause repeated retries, we limit the number of retries to a small finite number. In the event that a systematic failure does occur (possibly due to bad input), the SNS subsystem returns an error that may be trapped by the service in an application-specific manner.

The heavy use of timers all over the SNS subsystem for fault recovery, beaconing, and load reporting can result in synchronization effects, as have been observed in a number of systems, both naturally occurring phenomena, and computer-based systems [24]. To counter these effects, we introduce a randomized element to the timeout values. We use a uniformly distributed random value centered around the required absolute value for all timers in the system. For randomized timers that are used to suppress duplicate events (e.g. SNS Manager restart), exponential timers have been

shown to provide better results [44]. But in practice, we have found uniform timers to be sufficient.

3.5 Soft State for Fault Tolerance

As described above, our use of soft state all over the system in preference to a hard state design has greatly simplified our implementation. Traditional crash recovery systems such as ACID databases [30] use write-ahead logging [42] to ensure that no data is lost. This piece of code is extremely difficult to understand and debug: it is invoked on (hopefully) very rare occasions, but when it does get invoked, it has to work without any errors. By ensuring that the SNS subsystem has no hard state, we have precluded the need for any explicit data recovery protocols upon a crash.

Moreover, with this use of soft-state, no mirror processes are required for fault tolerance. 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 the peer's duties).

4 Example Applications

The SNS architecture provides an abstract layer for building various kinds of network services. We describe a few of the applications that we and other researchers have built using this architecture.

4.1 *TranSend*: A Web Distillation Proxy

This was our first prototype built using the SNS platform. TranSend [28] is an HTTP proxy that performs on-the-fly content transformation for web documents. The goal of this service is to provide fast access to the web for clients connected via low-bandwidth links, by performing lossy compression [27, 29] of images and other rich data on the fly. TranSend uses four types of workers:

- Front end workers that listen for HTTP requests, forward them through the system, and return the result to the client.
- Web cache workers that fetch the actual content from the web, and cache it for optimization (cache workers have affinity to the disk on which they are caching their content, and must be started on a node that can access the cache disk)
- Image distillation engines for compressing GIF and JPEG images.
- An HTML "munger" that marks up inline image references with distillation preferences and adds links next to distilled images, so users can retrieve the originals.

Most of these workers took about 5-6 hours to implement, and although program bugs may sometimes cause the worker to crash, the fault tolerance mechanisms built into the SNS ensure that we do not have to worry about eliminating all such possible bugs from the worker code.

4.2 *Wingman*: A Web Browser for the PalmPilot

Wingman [26] represents a class of applications that use an infrastructural proxy to build a split application for "thin clients" such as PDAs. Most of the application functionality is moved to a powerful proxy, while the client application is a stripped-down bare-bones program. Wingman is a proxy-based web browser for the 3COM PalmPilot [1]; it uses the SNS architecture to convert web content into much simpler representations.

- Front end workers listen for requests from the Wingman client, much like the front end in TranSend, while web cache workers fetch content from the web.
- Rather than implementing a full HTML parser on the client, a specialized HTML layout engine converts HTML documents into simple draw operations that are shipped to the client.
- Specialized image converters transform web images into the Pilot's native 2 bits/pixel bitmap representation.

Both TranSend and Wingman have been released to the general public for use, and we currently enjoy a user population of approximately 12,000.

4.3 *MARS*: A Multimedia Archive Server

This is a server for distributing archived MBone content which was developed by researchers at Berkeley using the SNS system as the underlying platform [45]. This system uses two main types of workers: a *server manager* that clients communicate with to initiate new sessions, and an audio/video worker that performs the actual playback onto the MBone using the Real-time Transport (RTP) [46] protocol. Once a session has begun, clients can directly communicate with the a/v worker to control the session parameters. This worker is different from most other workers in the previous examples, because it inherently has state: the session it is playing, the current playback point in the stream, etc. The application relies on smart clients to regenerate this state for the worker, in case it dies and must be restarted. A client can detect failure of the worker via the loss of RTP control packets; it immediately contacts the server manager to restart the worker; once the worker has come up, the client can communicate the lost state back to the worker.

In addition to these applications, the SNS architecture has been used for a number of other services. Some of them include a proxy-based electronic shared whiteboard for PDAs [14], and Cha-Cha, a system for contextualizing web-search results to provide more structure by organizing the results according to some semantics such as document location, content, etc. [15].

5 System Evaluation

We took measurements of our implementation using the TranSend proxy as the testbed. We present some of our results for the load balancing strategies employed, the ability of the system to tune itself to the offered load, and its scalability. Load was generated using a simple tool to play out HTTP requests based on trace data that was

gathered from the UC Berkeley dialup IP users [31]. We instrumented the worker drivers to collect statistics about the load on each worker as we varied the rate of requests generated to the system.

5.1 Load Balancing

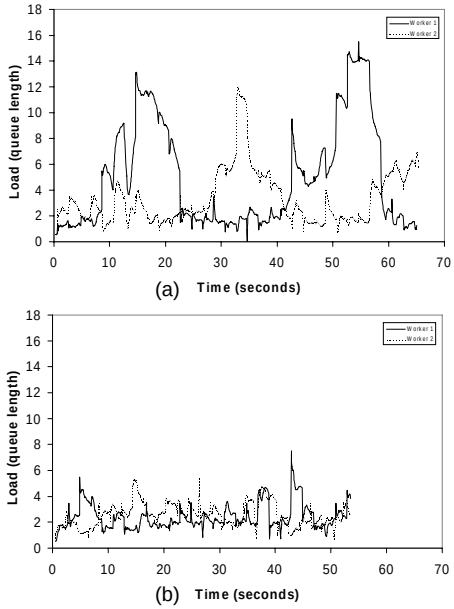


Figure 2: **Load balancing algorithm: (a) Without feedback: load oscillates between two workers, (b) With feedback**.

The TranSend proxy uses the default load metric: queue lengths in the worker drivers. Figure 2 shows the effects of the load balancing strategy (a) without, and (b) with the correction feedback included in the selection algorithm. The absence of any feedback manifests itself in oscillations in the load experienced by workers. During a single beaconing interval, one worker is lightly loaded and most tasks are directed to it; when then load information is updated in the front end's cache, the front end

immediately stops directing further tasks to this worker and redirects them to another less loaded worker, which in turn starts to get loaded instead.

Figure 2 (b) shows the results of running the same experiment with the improved feedback-based algorithm. As expected, the oscillations are reduced and queue levels pretty much stay below five items at any time.

5.2 Self Tuning

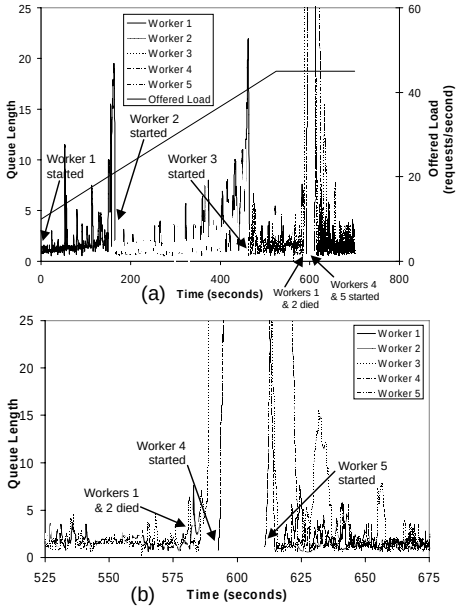


Figure 3: **Worker queue length observed over time as the load presented to the system fluctuates and as workers are brought down manually. Figure (b) is an enlargement of a section of (a).**

Figure 3 (a) shows the variations in worker queue lengths over time. The system was bootstrapped with one front end and the SNS Manager. We observe on-demand spawning of the first worker as soon as load is offered to the system. With increasing load, the worker queue gradually increases until the SNS Manager decides to spawn

a second worker, which reduces the queue length of the first worker and balances the load across the two workers within five seconds. Continued increase in load causes a third worker to start up.

Figure 3 (b) shows an enlarged view of a section of the graph in Figure 3 (a). During the experiment, we manually killed the first two workers, causing the load on the remaining worker to rapidly increase. At that point, the request rate was 45 per second. This high rate was too much for the single worker that was left, and its queue length immediately shot up to almost a hundred. (To show greater detail in the graphs, this peak has been cut off in Figure 3). However, we notice that the manager immediately reacted and started a new worker. The second worker is not sufficient to bring the system back to a stable state, and the manager must start yet another worker.

5.3 Scalability

Requests/ second	# of front ends	# of workers	Element that saturated
0-24	1	1	workers
25-47	1	2	workers
48-72	1	3	workers
73-87	1	4	FE Ethernet*
88-91	2	4	workers
92-112	2	5	workers
113-135	2	6	workers and FE Ethernet
136-159	3	7	workers

* – Ethernet leading into the front end

Table 1: **Results of the scalability experiment**

To demonstrate the incremental scalability of the system, we conducted an experiment as follows:

- Start with a minimal instance of the system: one front end, one worker, and the SNS Manager.
- Gradually increase the offered load by increasing the request rate to the front end until some system component saturates.
- Observe the system pull in more resources as the load on the system increases.

Table 1 summarizes the results of this experiment. As we ramped up the request rate to 24 requests per second, the offered load exceeded the capacity of the single worker, and the manager spawned a new worker to handle the load. With further increases in load, the manager spawned more workers as needed. At 87 requests per second, the Ethernet segment leading into the front end saturated, requiring a new front end. We were unable to test the system at rates higher than 159 requests per second, since all the nodes in our cluster were used up at that point.

We noticed that our prototype worker could handle approximately 23 requests per second, and a 100 Mb/s Ethernet segment into a front end can handle approximately 70 requests per second. Since our test bed was an HTTP proxy, we believe a large part of the overhead at the front end was due to the TCP connection setup and processing overhead. We were unable to saturate either the front end itself or fully saturate the network within the proxy. Thus, even with a commodity 100 Mb/s network, linear scaling is limited primarily by bandwidth into the system rather than bandwidth inside the system.

A possible concern is that the centralized SNS Manager might prove to be a bottleneck in the system. Our experience shows that this has never been a problem. The main task of the manager (in steady state) is to accumulate load announcements from all workers and multicast this information to the rest of the system. We conducted an experiment to test the capability of the manager to handle these load announcements. 900 stub workers were created on four machines. Each of these workers generated a

load announcement packet approximately every half a second. The manager was easily able to handle this aggregate load of 1800 announcements per second. 900 workers represents quite a sizeable load on the system, so we do not expect the SNS Manager to be a performance bottleneck.

A potential concern for scalability the the amount of network bandwidth eaten up by the control traffic (load reports, existence beacons, etc.). Most of these control packets are relatively small – payload sizes on the order of 10-30 bytes. Even with a large number of workers, the control traffic should not exceed a few kilobytes. We can impose a strict limit on the amount of control traffic by adjusting the rate at which this traffic is generated with the size of the system. Since this is easily detectable, each component can scale back its load reporting rate as the system grows beyond a certain point. This scheme is used extensively in MBone protocols: for example, the MBone Session Announcement Protocol [34] limits its traffic to a maximum bandwidth by requiring clients to adjust their rate of generating messages as the size of the multicast group increases.

6 Related and Future Work

The SNS architecture has been used as the basis for a programming model, *TACC* [25]: Transformation (filtering, compressing, or format conversion of data), Aggregation (combining data from multiple sources), Caching (both of original and transformed data), and Customization (a persistent data store of per-user preferences to control the behavior of the service). In fact, both the TranSend and Wingman examples build upon this model.

The active service framework [3] is another approach at a substrate for network service applications that is designed for the MBone. This framework uses many concepts similar to ours: the use of clusters for availability and scalability, and soft state. It has been used as a building block for deploying media gateways for real-time audio/video transformation [4].

WebOS [51] and SWEB++ [7] have exploited the extensibility of client browsers via Java and JavaScript to enhance scalability of network-based services. Client cooperation is also used in SPAND (Shared Passive Network Performance Discovery) [47], a network performance prediction system that allows a group of similarly connected clients to pool information about the network characteristics of the path to a distant network server. [41] explores theoretical models for analyzing various randomized load balancing algorithms.

Primary/secondary fault tolerance was explored in early systems such as [10]. Our style of "starfish" fault tolerance is more related to the peer-mode fault tolerance exhibited by early "Worm" programs [48] and to "Robin Hood/Friar Tuck" fault tolerance: "Each ghost job would detect 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 or to deliberately crash the system." [23]

A number of other systems have used the notion of soft state to achieve better performance and robustness. Networking systems for routing protocols [39, 19] and performance optimization [9] rely on soft state to update their tables; this allows them to continue operating in the event of partial failures and ensures that no special recovery protocols are needed to regenerate state that might have been lost in a crash. Various soft state systems such as Bayou [21] have explicitly traded consistency for availability in application-specific ways.

In the future, we plan on developing new applications using the SNS platform in order to explore the limitations of the system. We plan to continue investigating more stateful or transaction-oriented workers, and analyze the effectiveness of SNS to support these kinds of services. Another aspect that we want to consider is migration of tasks from one worker to another if the system detects that the worker is taking too long.

7 Conclusions

We have presented an architecture substrate for building Internet service applications. We rely on clusters of workstations to provide guarantees of scalability and availability to the service application. Application developers program their services to a well-defined narrow interface that is used to communicate with the rest of the system. The SNS takes care of resource location, spawning, and fault detection and tolerance for the application.

Our architecture reflects principles that we believe are fundamental to the design of a scalable, available Internet service. Trading off consistency for availability via the use of soft state and periodic beaconing greatly simplifies the design and makes fault tolerance and recovery much simpler. We rely on a mode of fault tolerance that we call "starfish fault tolerance". This has many advantages over traditional primary/secondary fault tolerance in terms of resource usage and recovery algorithms.

We have used this architecture for building a number of applications: a web proxy (TranSend), a proxy-based graphical web browser for PDAs (Wingman), and an MBone archive server (MARS). These applications demonstrate the versatility of the architecture, and the ability of individual applications to customize it to their requirements. Our evaluation of the system demonstrates that the load balancing strategies work in practice, and that the system is able to scale incrementally with increasing load. Two of the above services have been deployed on a permanent basis on the UC Berkeley campus, and have approximately 12,000 regular users.

As Internet services become ubiquitous, we expect to see many more systems based on our (or a similar) architecture. The exponential growth of the web will only increase the demand for application platforms for building services quickly and easily. The SNS architecture is a step in that direction.

8 Acknowledgements

We would like to thank our collegues, Armando Fox, and Steve Gribble, for all their input and support during the course of this work. We'd like to thank Angela Schuett, Drew Roselli, and Eric Anderson for their comments on early drafts of this paper. We are also grateful to Professor Randy Katz for his suggestions and vision through the course of this research.

References

[1] 3COM CORPORATION. 3COM PalmPilot. http://www.3com.com/palm/index.html.

[2] ALMEROTH, K., AND AMMAR, M. The Interactive Multimedia Jukebox (IMJ): A new paradigm for the on-demand delivery of audio/video. In *Proceedings of the Seventh International World Wide Web Conference* (April 1998).

[3] AMIR, E., MCCANNE, S., AND KATZ, R. An Active Service Framework and its Application to Real-time Multimedia Transcoding, Jan. 1998. *Submitted for publication*.

[4] AMIR, E., MCCANNE, S., AND ZHANG, H. An Application-level Video Gateway. In *Proceedings of ACM Multimedia '95* (San Francisco, CA, Nov. 1995), pp. 255–265.

[5] ANDERSON, E. The Magicrouter: An Application of Fast Packet Interposing. Class report, UC Berkeley, Dec. 1995.

[6] ANDERSON, T. E., CULLER, D. E., PATTERSON, D. A., AND THE NOW TEAM. A Case for Networks of Workstations: NOW. In *Principles of Distributed Computing* (Aug. 1994).

[7] ANDRESEN, D., YANG, T., EGECIOGLU, O., IBARRA, O. H., AND SMITH, T. R. Scalability Issues for High Performance Digital Libraries on the World Wide Web. In *Proceedings of ADL '96, Forum on Research and Technology Advances in Digital Libraries, IEEE* (Washington, D.C., May 1996).

[8] ANONYMIZER INC. The Web Anonymizer. http://altavista.digital.com/.

[9] BALAKRISHNAN, H., SESHAN, S., AMIR, E., AND KATZ, R. Improving TCP/IP Performance Over Wireless Networks. In *Proceedings of the first ACM Conference on Mobile Computing and Networking* (Berkeley, CA, Nov. 1995).

[10] BARTLETT, J. F. A NonStop Kernel. In *Proceedings of the 8th SOSP* (Dec. 1981).

[11] BOLOSKY, W. J., FITZGERALD, R. P., AND DOUCEUR, J. R. Distributed schedule management in the Tiger video fileserver. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (October 1997).

[12] BORENSTEIN, N., AND FREED, N. *MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies*, Sept. 1993. RFC-1521.

[13] BRISCO, T. *DNS Support for Load Balancing*, Apr. 1995. RFC-1764.

[14] CHAWATHE, Y., FINK, S., MCCANNE, S., AND BREWER, E. A Proxy Architecture for Reliable Multicast in Heterogeneous Environments, Feb. 1998. *Submitted for publication to ACM Multimedia '98*.

[15] CHEN, M., AND HONG, J. Cha-Cha: Contextualizing Hypertext Searches. Class report, UC Berkeley, Dec. 1997.

[16] CISCO SYSTEMS. Local Director. http://www.cisco.com/warp/public/751/lodir/.

[17] CITYSEARCH INC. http://www.citysearch.com/.

[18] CROVELLA, M. E., AND BESTAVROS, A. Explaining world wide web traffic self-similarity. Tech. Rep. TR-95-015, Computer Science Department, Boston University, Oct 1995.

[19] DEERING, S., ESTRIN, D., FARINACCI, D., JACOBSON, V., LIU, C.-G., AND WEI, L. An Architecture for Wide-area Multicast Routing. *IEEE/ACM Transactions on Networking 4*, 2 (Apr. 1996).

[20] DEERING, S. E. *Multicast Routing in a Datagram Internetwork*. PhD thesis, Stanford University, Dec. 1991.

[21] DEMERS, A., PETERSON, K., SPREITZER, M., TERRY, D., THEIMER, M., AND WELCH, B. The Bayou Architecture: Support for Data Sharing Among Mobile Users.

[22] DIGITAL CORPORATION. The AltaVista Search Engine. http://altavista.digital.com/.

[23] E. S. RAYMOND, ED. The New Hackers Dictionary. MIT Press, 1991. Also http://www.ccil.org/jargon/jargon.html.

[24] FLOYD, S., AND JACOBSON, V. The Synchronization of Periodic Routing Messages. *IEEE/ACM Transactions on Networking 2*, 2 (Apr. 1994), 122–136.

[25] FOX, A. The Case For TACC: Scalable Servers for Transformation, Aggregation, Caching and Customization. *Qualifying Exam Proposal, UC Berkeley Computer Science Division*, April 1997.

[26] FOX, A., GOLDBERG, I., GRIBBLE, S. D., POLITO, A., LEE, D. C., AND BREWER, E. A. Experience with Top Gun Wingman, A Proxy-Based Graphical Web Browser for the 3COM PalmPilot, Mar. 1998. *Submitted to Middleware '98*.

[27] FOX, A., GRIBBLE, S., BREWER, E., AND AMIR, E. Adapting to Network and Client Variability via On-demand Dynamic Distillation. In *Proceedings of ASPLOS-VII* (Cambridge, MA, Oct. 1996).

[28] FOX, A., GRIBBLE, S., CHAWATHE, Y., BREWER, E., AND GAUTHIER, P. Cluster-based Scalable Network Services. In *Proceedings of SOSP '97* (St. Malo, France, Oct. 1997), pp. 78–91.

[29] FOX, A., GRIBBLE, S. D., CHAWATHE, Y., AND BREWER, E. A. Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives, Feb. 1998. *Submitted to a special issue of IEEE Personal Communications on Adapation*.

[30] GRAY, J. The Transaction Concept: Virtues and Limitations. In *Proceedings of VLDB '81* (Cannes, France, Sept. 1981), pp. 144–154.

[31] GRIBBLE, S. D., AND BREWER, E. A. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems* (Monterey, CA, Dec. 1997).

[32] GRIBBLE, S. D., AND FOX, A. Load Characterization and the Implementation of Service Levels in the Scalable Proxy Architecture. Class report, UC Berkeley, May 1997.

[33] GRIBBLE, S. D., MANKU, G. S., ROSELLI, D., BREWER, E. A., GIBSON, T. J., AND MILLER, E. L. Self-Similarity in File Systems. In *Proceedings of ACM SIGMETRICS '98* (June 1998).

[34] HANDLEY, M. *SAP: Session Announcement Protocol*. Internet Draft, Nov 19, 1996.

[35] INKTOMI CORP. The HotBot Search Engine. http://www.hotbot.com/.

[36] INTERNET ENGINEERING TASK FORCE. *HyperText Transfer Protocol – HTTP 1.1*, Mar. 1997. RFC-2068.

[37] LELAND, W. E., TAQQU, M. S., WILLINGER, W., AND WILSON, D. V. On the Self-Similar Nature of Ethernet Traffic (extended version). *IEEE/ACM Transactions on Networking 2* (Feb. 1994).

[38] MAPQUEST. http://www.mapquest.com/.

[39] MCQUILLAN, J., RICHER, I., AND ROSEN, E. The New Routing Algorithm for the ARPANET. *IEEE Transactions on Communications 28*, 5 (May 1980), 711–719.

[40] MITZENMACHER, M. How useful is old information? Extended abstract: Using stale information for load balancing in distributed systems.

[41] MITZENMACHER, M. Load Balancing and Density-dependent Jump Markov Processes. In *Proceedings of the FOCS '96* (1996).

[42] MOHAN, ET AL. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *TODS 17*, 1 (1992), 94–162.

[43] NATIONAL AERONAUTICS AND SPACE ADMINISTRATION. The Mars Pathfinder Mission Home Page. http://mpfwww.jpl.nasa.gov/default1.html.

[44] NONNENMACHER, J., AND BIERSACK, E. W. Optimal Multicast Feedback. In *Proceedings of IEEE INFOCOM '98* (San Francisco, CA, Mar. 1998).

[45] SCHUETT, A., AND RAMAN, S. MARS: A Media Archive Server for On-demand Remote Playback. Class report, UC Berkeley, Dec. 1997.

[46] SCHULZRINNE, H., CASNER, S., FREDERICK, R., AND JACOBSON, V. RTP: A Transport Protocol for Real-Time Applications. Internet Engineering Task Force, Audio-Video Transport Working Group, January 1996. RFC-1889.

[47] SESHAN, S., STEMM, M., AND KATZ, R. H. SPAND: Shared Passive Network Performance Discovery. In *Proceedings of the First Usenix Symposium on Internet Technologies and Systems (USITS) '97* (Monterey, CA, Dec. 1997).

[48] SHOCH, J. F., AND HUPP, J. A. The "Worm" programs: Early Experiences with Distributed Systems. *Communications of the Association for Computing Machinery 25*, 3 (Mar. 1982), 172–180.

[49] WALDSPURGER, C., AND WEIHL, W. Lottery Scheduling: Flexible Proportional Share Resource Management. In *Proceedings of the First OSDI* (Nov. 1994).

[50] YAHOO! INC. http://www.yahoo.com/.

[51] YOSHIKAWA, C., CHUN, B., EASTHAM, P., ANDERSON, T., AND CULLER, D. Using Smart Clients to Build Scalable Services. In *Proceedings of USENIX '97* (Jan. 1997).

9 Biography

Yatin Chawathe is a doctoral student at the University of California, Berkeley. He received a Bachelor of Engineering (Computer Engineering) degree from the University of Bombay, India, and a Master of Science (Computer Science) from the University of California, Berkeley. His primary interests are Internet services and application support for reliable multicast in heterogeneous environments.

Eric A. Brewer is an Assistant Professor of Computer Science at the University of California, Berkeley, and received his PhD in Computer Science from MIT in 1994. Interests include mobile and wireless computing (the InfoPad and Daedalus projects); scalable servers (the NOW, Inktomi, and TranSend projects); and application- and system-level security (the ISAAC project). Previous work includes multiprocessor-network software and topologies, and high-performance multiprocessor simulation.