System Support for Scalable and Fault Tolerant Internet Services

Yatin Chawathe and Eric A. Brewer
Computer Science Division, University of California at Berkeley
{yatin, brewer}@cs.berkeley.edu

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 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 paper, 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.

Keywords

Scalable Network Services, fault tolerance, scalability, availability, load balancing

1 INTRODUCTION

The explosive growth of the World-Wide Web has spawned a number of network-based services such as map servers (MapQuest 1996), search engines (Inktomi Corp. 1996, Digital Corporation 1996), anonymizing web-proxies (Anonymizer Inc. 1996), and directory services (Yahoo! Inc 1995, CitySearch Inc. 1997). Similarly, the expansion of the Internet multicast backbone (MBone) (Deering, Estrin, Farinacci, Jacobson, Liu & Wei 1996, Deering 1991) has created a demand for services such as video-on-demand servers (Bolosky, Fitzgerald & Douceur 1997, Almeroth & Ammar 1998) and real-time audio/video gateways (Amir, McCanne & Zhang 1995).

As these types of services get more popular, they will be required to provide certain 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 offered load. To ensure linear scalability, the service must distribute the load across all of its resources. Further, the service must be capable of absorbing any peaks in load.

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 faults in its internal operation do not take down the entire service.

In (Fox, Gribble, Chawathe, Brewer & Gauthier 1997), we argued that clusters of workstations (Anderson, Culler, Patterson & the NOW team 1994) are ideal for most Internet service 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. Not only do PCs have cost/performance benefits, but also the natural redundancy in a cluster of PCs is excellent for masking 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 above 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 kinds of services.

2 A REUSABLE "SCALABLE NETWORK SERVICES" ARCHITECTURE

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 ar-

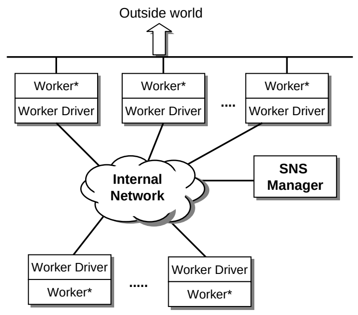

Figure 1 The SNS Architecture. (*****—Service developers customize these components; all other components are part of the SNS)

chitecture to separate the service support layer from the rest of the system, thereby isolating the rest of the system from having to deal with the SNS 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. They 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 with the outside world. Such a worker receives requests or commands from clients, and either handles them directly, or “shepherds” them to other workers.

For example, a web-based map service can implement a “front end” worker that actively listens for HTTP requests from end-clients, and acts as a control center for the service. The front end may rely on a back end map worker that generates appropriate map images based on the HTTP request and returns them to the front end for eventual delivery to the client. 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. Some workers, however, 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. 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. Within a given class, workers are identical and interchangeable, i.e. 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*) sends 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. Workers 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.

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 and scaling, 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 for each request.

2.2.2 Load Balancing and Scalability

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 used by workers to make local decisions regarding their choice of consumers for tasks.

As the load on the system grows, the SNS Manager pulls in idle nodes, and starts new workers. In addition, the system must be able to deal with sudden bursts in load (Leland, Taqqu, Willinger & Wilson 1994, Crovella & Bestavros 1995, Gribble, Manku, Roselli, Brewer, Gibson & Miller 1998). We deal with short traffic bursts by replicating workers and directing tasks across all replicated workers for greater throughput.

To handle more prolonged bursts, our design 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. For example, after the landing of the Pathfinder on Mars, NASA's web site experienced over 220 million hits in a four day period (National Aeronautics and Space Administration 1997) which excessively overloaded their web server. Overflow nodes can be machines that are usually used for other purposes (such as people's desktop machines), 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 launch workers on idle machines from the overflow pool for the duration of the burst.

2.2.3 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. 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. The system also uses *timeouts and retries*; if a task cannot be completed because of some failure in the system, the SNS subsystem will retry the task.

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; 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.

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 permits the actual 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. 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.

We now look at the three functions of the SNS Manager—resource location, load balancing, and fault tolerance—in detail.

3.1 Resource location

The resource location problem for the SNS encompasses three issues: how clients locate the service, how workers locate the centralized SNS Manager, and how 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. Web-based services use Uniform Resource Locators (URLs) as the primary service location mechanism. Although this has fault-tolerance implications, techniques such as DNS round-robin (Brisco 1995) and specialized routers (Cisco Systems 1996, Anderson 1995) can be used to hide a number of replicated and/or distributed servers behind a single logical name. Smart clients (Yoshikawa, Chun, Eastham, Anderson & Culler 1997) can assist the service in transparent service location in the presence of faults by relying on browser support through Java and JavaScript. Similarly, services that rely on the IP multicast delivery model can use multicast as a level of indirection to locate the service (Amir, McCanne & Katz 1998). 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 that is loosely based on MIME types (Borenstein & Freed 1993). For example, all transformation workers that compress GIF images may be called *compress/image/gif*. Applications must pick names for their workers in a manner that makes 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 and the worker's location, 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. 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* to avoid contacting the SNS Manager on every request.

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 *launch table* of all workers that can be part of the service and the executables associated with each worker. The manager consults this table and attempts to start a new worker on a lightly loaded node in the cluster. 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. In addition to load, the spawning policy also takes into account any affinities that a worker might have for specific nodes in the cluster.

If the manager is unable to locate or spawn a new worker, this information is cached in a *negative cache* at the originator. This ensures that the originator does not bother the manager again for that class of workers. The negative cache is flushed whenever the launch table is updated by the manager, so no stale information 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 1 unit of load per task, thus making the load equivalent to the number of pending tasks at the worker.

The worker driver maintains a running statistic of the current load. Periodically, it reports this information to the SNS Manager. 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 locally. We use a scheme based on lottery scheduling (Waldspurger & Weihl 1994) for the load balancing algorithm. 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 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. The worker adds this correction factor to the cached load balancing hints when it tries to pick a consumer. Section 5.1 discusses the effectiveness of the policy.

3.3.3 Auto-launch for scalability

A good 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.

Even after a new worker has started, it may take time before the load stabilizes across the new and old workers. To avoid further false triggers of the auto-launch mechanism, we disable the 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 2.2.2, 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 from the overflow pool. When the burst dies down, the overflow workers may be reaped and the machine can be returned to its original task. We use a priority scheme to implement the overflow pool. Dedicated nodes are assigned the highest priority level, while overflow nodes have lower priorities.

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 is independent of the others. We rely on two main principles in our handling of failures: component restart and timeouts used by 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* (Anderson 1995, Bartlett 1981) 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.

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. 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 restart it upon failure. 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 (loss of) periodic load reports from the workers. 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 SNS Manager simply relies on the on-demand launch mechanism to 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 clients). When the SNS Manager detects the death of such a worker, instead of the usual on-demand recovery, it immediately launches a new one.

For workers that have affinity to specific nodes, the SNS Manager must restrict the respawning to 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. 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 (Brisco 1995) or specialized routers (Cisco Systems 1996, Anderson 1995). 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. We now 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. When beacons from the manager are lost, a distributed restart algorithm is triggered. Each worker starts a random timer; the worker whose timer expires first wins the arbitration process and restarts the manager. It multicasts an *AmRestarting* message to the rest of the system, allowing other components to back off their timers. In spite of this arbitration algorithm, multiple managers may get spawned. When the managers detect each other's beacons, all but one commit suicide.

Even while the manager is dead, the system can continue to function, albeit at some loss of functionality, using cached information on the workers. The use of soft state updates ensures that crash recovery itself is relatively simple. When the manager restarts and begins beaconing again, all workers detect the new beacons and re-register with the new manager. With time, all workers report their load and the manager can regain its lost load 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. Workers use timeouts to bound the time they must wait for a consumer to return the result of a task, before retrying the task on a different worker. Other uses of timeouts and retries include launching of new workers, handling software faults in a worker application that cause it to spin in an endless loop while handling a task, etc.

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 (Floyd & Jacobson 1994). 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 (Nonnenmacher & Biersack 1998). However, in practice, we have found uniform timers to be sufficient.

3.4.5 Soft State for Fault Tolerance

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 (Gray 1981) use write-ahead logging (Mohan et al. 1992) to ensure that no data is lost. This piece of code is extremely difficult to understand and debug: it is rarely invoked, yet 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 need only detect that its peer is alive, and be able to restart the peer.

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**

TranSend (Fox et al. 1997) is an HTTP proxy whose goal is to provide fast access to the web for clients connected via low-bandwidth links by performing lossy compression of images and other rich data on the fly (Fox, Gribble, Brewer & Amir 1996, Fox, Gribble, Chawathe & Brewer 1998). 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;
- image distillation engines for compressing GIF and JPEG images;

- an HTML engine 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 5-6 hours to implement, and although program bugs may sometimes cause the worker to crash, the fault tolerance mechanisms built into the SNS reduce the need to eliminate all such possible bugs from the worker code.

4.2 *Wingman*: A Web Browser for the PalmPilot

Wingman (Fox, Goldberg, Gribble, Polito, Lee & Brewer 1998) 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 program. Wingman is a proxy-based web browser for the 3COM PalmPilot (3COM Corporation 1996); 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.
- 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 PalmPilot's native 2 bits/pixel bitmap representation.

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

4.3 *MARS*: A Multimedia Archive Server

MARS is a server for distributing archived MBone content which was developed by researchers at Berkeley using the SNS system as the underlying platform (Schuett & Raman 1997). This system uses two types of workers: a *server manager* that clients contact to initiate new sessions, and an audio/video worker that performs the actual playback onto the MBone using the Real-time Transport (RTP) (Schulzrinne, Casner, Frederick & Jacobson 1996) 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 workers in the previous examples, because it 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.

The SNS architecture has been used for other services, including a proxy-based electronic shared whiteboard for PDAs (Chawathe, Fink, McCanne & Brewer 1998), and Cha-Cha, a system for contextualizing web-search results by augmenting them with semantics such as document location, content, etc. (Chen & Hong 1997).

5 SYSTEM EVALUATION

We used the TranSend proxy as a test-bed for measuring our implementation . 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 (Gribble & Brewer 1997).

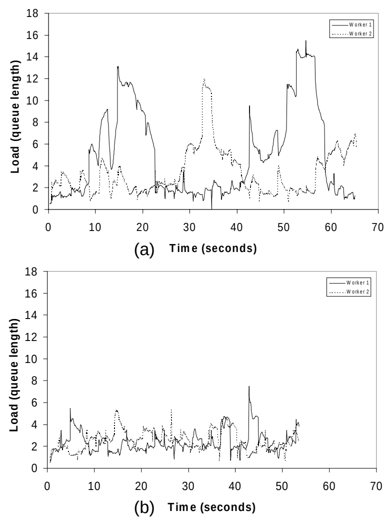

Figure 2 Load balancing algorithm: (a) Without feedback: load oscillates between two workers, (b) With feedback.

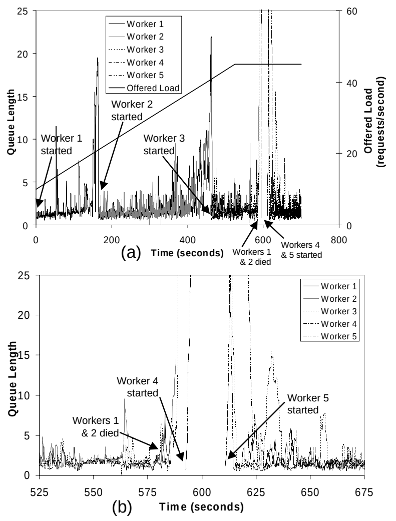

Figure 3 Worker queue length observed over time as the load presented to the system fluctuates and as workers are brought down manually. Figure (b) is an enlargement of a section of (a).

5.1 Load Balancing

The TranSend proxy uses queue lengths in the worker drivers as the default load metric. 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 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 a less loaded worker.

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 generally stay below five requests at any time.

5.2 Self Tuning

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. (To show greater detail in the graphs, this peak has been cut off in Figure 3.) The manager immediately reacted and started a new worker. The second worker was not sufficient to bring the system back to a stable state, and the manager had to 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

Table 1 Results of the scalability experiment (*****—Ethernet leading into the front end)

The data in Table 1 demonstrates the system's incremental scalability. We started with a minimal instance of the system (one front end and one worker), and gradually increased the request rate. As the load increased,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. 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.

A possible concern is that the centralized SNS Manager might prove to be a bottleneck in the system. Our experience, and a simple experiment shows that this has not been a problem—the SNS Manager was easily able to manage at least 900 workers in the system. A potential concern for scalability is the amount of network bandwidth consumed by control traffic (load reports, existence beacons, etc.). However, most of these packets are relatively small. We can impose a strict limit on the amount of control traffic by adjusting the rate, by either intelligently selecting dynamic data over data that changes less rapidly, or using a scheme similar to the MBone Session Announcement Protocol (Handley 1996) which 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 WORK

The SNS architecture has been used as the basis for a programming model, *TACC* (Fox 1997): 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 (Amir et al. 1998) 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 (Amir et al. 1995).

Our style of starfish fault tolerance is related to the peer-mode fault tolerance exhibited by early “Worm” programs (Shoch & Hupp 1982) and to “Robin Hood/Friar Tuck” fault tolerance (Raymond, E. S., ed. 1991). In (Mitzenmacher 1997), the authors analyze various load balancing policies for distributed systems that use periodic load reporting. As a rule of thumb, they found that selecting the less loaded of two randomly chosen workers worked well.

A number of other systems have used the notion of soft state to achieve better performance and robustness. Networking systems for routing protocols (McQuillan, Richer & Rosen 1980, Deering et al. 1996) and performance optimization (Balakrishnan, Seshan, Amir & Katz 1995) 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 (Demers, Peterson, Spreitzer, Terry, Theimer & Welch n.d.) have explicitly traded consistency for availability in application-specific ways.

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.

REFERENCES

3COM Corporation (1996), '3COM PalmPilot'. http://www.palmpilot.com/.

Almeroth, K. & Ammar, M. (1998), 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'.

Amir, E., McCanne, S. & Katz, R. (1998), An Active Service Framework and its Application to Real-time Multimedia Transcoding, *in* 'Proceedings of ACM SIGCOMM '98'.

Amir, E., McCanne, S. & Zhang, H. (1995), An Application-level Video Gateway, *in* 'Proceedings of ACM Multimedia '95', San Francisco, CA, pp. 255–265.

Anderson, E. (1995), The Magicrouter: An Application of Fast Packet Interposing. Class report, UC Berkeley.

Anderson, T. E., Culler, D. E., Patterson, D. A. & the NOW team (1994), A Case for Networks of Workstations: NOW, *in* 'Principles of Distributed Computing'.

Anonymizer Inc. (1996), 'The Web Anonymizer'. http://www.anonymizer.com/.

Balakrishnan, H., Seshan, S., Amir, E. & Katz, R. (1995), Improving TCP/IP Performance Over Wireless Networks, *in* 'Proceedings of the first ACM Conference on Mobile

Computing and Networking', Berkeley, CA.

Bartlett, J. F. (1981), A NonStop Kernel, *in* 'Proceedings of the 8th SOSP'.

Bolosky, W. J., Fitzgerald, R. P. & Douceur, J. R. (1997), Distributed schedule management in the Tiger video fileserver, *in* 'Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles'.

Borenstein, N. & Freed, N. (1993), *MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies*. RFC-1521.

Brisco, T. (1995), *DNS Support for Load Balancing*. RFC-1764.

Chawathe, Y., Fink, S., McCanne, S. & Brewer, E. (1998), 'A Proxy Architecture for Reliable Multicast in Heterogeneous Environments'.

Chen, M. & Hong, J. (1997), Cha-Cha: Contextualizing Hypertext Searches. Class report, UC Berkeley.

Cisco Systems (1996), 'Local Director'. http://www.cisco.com/warp/public/751/lodir/.

CitySearch Inc. (1997). http://www.citysearch.com/.

Crovella, M. E. & Bestavros, A. (1995), Explaining world wide web traffic self-similarity, Technical Report TR-95-015, Computer Science Department, Boston University.

Deering, S. E. (1991), Multicast Routing in a Datagram Internetwork, PhD thesis, Stanford University.

Deering, S., Estrin, D., Farinacci, D., Jacobson, V., Liu, C.-G. & Wei, L. (1996), 'An Architecture for Wide-area Multicast Routing', *IEEE/ACM Transactions on Networking* **4**(2).

Demers, A., Peterson, K., Spreitzer, M., Terry, D., Theimer, M. & Welch, B. (n.d.), 'The Bayou Architecture: Support for Data Sharing Among Mobile Users'.

Digital Corporation (1996), 'The AltaVista Search Engine'. http://altavista.digital.com/.

Floyd, S. & Jacobson, V. (1994), 'The Synchronization of Periodic Routing Messages', *IEEE/ACM Transactions on Networking* **2**(2), 122–136.

Fox, A. (1997), The Case For TACC: Scalable Servers for Transformation, Aggregation, Caching and Customization. *Qualifying Exam Proposal, UC Berkeley Computer Science Division*.

Fox, A., Goldberg, I., Gribble, S. D., Polito, A., Lee, D. C. & Brewer, E. A. (1998), Experience with Top Gun Wingman, A Proxy-Based Graphical Web Browser for the 3COM PalmPilot, *in* 'Proceedings of Middleware '98'.

Fox, A., Gribble, S., Brewer, E. & Amir, E. (1996), Adapting to Network and Client Variability via On-demand Dynamic Distillation, *in* 'Proceedings of ASPLOS-VII', Cambridge, MA.

Fox, A., Gribble, S., Chawathe, Y., Brewer, E. & Gauthier, P. (1997), Cluster-based Scalable Network Services, *in* 'Proceedings of SOSP '97', St. Malo, France, pp. 78–91.

Fox, A., Gribble, S. D., Chawathe, Y. & Brewer, E. A. (1998), Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives, *in* 'A special issue of IEEE Personal Communications on Adapation'.

Gray, J. (1981), The Transaction Concept: Virtues and Limitations, *in* 'Proceedings of VLDB '81', Cannes, France, pp. 144–154.

Gribble, S. D. & Brewer, E. A. (1997), 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.

Gribble, S. D., Manku, G. S., Roselli, D., Brewer, E. A., Gibson, T. J. & Miller, E. L. (1998), Self-Similarity in File Systems, *in* 'Proceedings of ACM SIGMETRICS '98'.

Handley, M. (1996), *SAP: Session Announcement Protocol*. Internet Draft.

Inktomi Corp. (1996), 'The HotBot Search Engine'. http://www.hotbot.com/.

Leland, W. E., Taqqu, M. S., Willinger, W. & Wilson, D. V. (1994), 'On the Self-Similar Nature of Ethernet Traffic (extended version)', *IEEE/ACM Transactions on Networking* **2**.

MapQuest (1996). http://www.mapquest.com/.

McQuillan, J., Richer, I. & Rosen, E. (1980), 'The New Routing Algorithm for the ARPANET', *IEEE Transactions on Communications* **28**(5), 711–719.

Mitzenmacher, M. (1997), How useful is old information? Extended abstract: Using stale information for load balancing in distributed systems.

Mohan et al. (1992), 'ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging', *TODS* **17**(1), 94–162.

National Aeronautics and Space Administration (1997), 'The Mars Pathfinder Mission Home Page'. http://mpfwww.jpl.nasa.gov/default1.html.

Nonnenmacher, J. & Biersack, E. W. (1998), Optimal Multicast Feedback, *in* 'Proceedings of IEEE INFOCOM '98', San Francisco, CA.

Raymond, E. S., ed. (1991), 'The new hackers dictionary', MIT Press. Also http://www.ccil.org/jargon/jargon.html.

Schuett, A. & Raman, S. (1997), MARS: A Media Archive Server for On-demand Remote Playback. Class report, UC Berkeley.

Schulzrinne, H., Casner, S., Frederick, R. & Jacobson, V. (1996), 'RTP: A Transport Protocol for Real-Time Applications', Internet Engineering Task Force, Audio-Video Transport Working Group. RFC-1889.

Shoch, J. F. & Hupp, J. A. (1982), 'The "Worm" programs: Early Experiences with Distributed Systems', *Communications of the Association for Computing Machinery* **25**(3), 172–180.

Waldspurger, C. & Weihl, W. (1994), Lottery Scheduling: Flexible Proportional Share Resource Management, *in* 'Proceedings of the First OSDI'.

Yahoo! Inc (1995). http://www.yahoo.com/.

Yoshikawa, C., Chun, B., Eastham, P., Anderson, T. & Culler, D. (1997), Using Smart Clients to Build Scalable Services, *in* 'Proceedings of USENIX '97'.

8 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.