A Scalable Services Architecture*

Tudor Marian Ken Birman Robbert van Renesse
Department of Computer Science
Cornell University, Ithaca, NY 14853
{tudorm,ken,rvr}@cs.cornell.edu

Abstract

Data centers constructed as clusters of inexpensive machines have compelling cost-performance benefits, but developing services to run on them can be challenging. This paper reports on a new framework, the Scalable Services Architecture *(SSA), which helps developers develop scalable clustered applications. The work is focused on non-transactional high-performance applications; these are poorly supported in existing platforms. A primary goal was to keep the SSA as small and simple as possible. Key elements include a TCP-based "chain replication" mechanism and a gossip-based subsystem for managing configuration data and repairing inconsistencies after faults. Our experimental results confirm the effectiveness of the approach.*

1 Introduction

Large computing systems are often structured as Service Oriented Architectures (SOAs), for example using Web Services platforms. In such systems, clients access services in a request-reply model. Each service is self-contained, offers its own API, and handles its own quality of service or availability guarantees, for example by arranging to be restarted after a failure.

While many services need to maintain availability in the face of challenging operating conditions (including load fluctuations, transient network disruptions, and node failures), building services with these properties is difficult. Existing Web Services platforms offer load-balancing and restart mechanisms for *transactional* services implemented using a three-tier database model, but not for services implemented using other technologies. Developers of non-transactional web services must implement their own mechanisms for replicating data, tracking membership and liveness, redirecting requests during failures to minimize client disruption, and detecting and repairing inconsistencies.

Our premise in this paper is that for many services, the transactional model is a poor fit and hence that tools aimed at non-transactional web services systems will be needed. We recognize that this is debatable.

Vendors have generally argued that only transactional systems offer the hooks needed to support automated scalability, self-repair and restart mechanisms. Key to this argument is the ease with which interrupted transactions can be rolled back (and restarted, if needed), and the relative simplicity of cleaning up a database after a crash. Yet the transactional programming model also brings constraints and overheads, were this not the case, the transactional model would long ago have become universal.

Some of these constraints relate to the challenges of maintaining a clean separation of code and data; not all applications can be structured in this manner. Transactional rollback and restart can be costly, and restarting a database after a crash incurs delays while cleanup code runs. High availability is difficult to acheive in the transactional model; the fastest database replication schemes (asynchronous log replay) suffer from failure scenarios that can require intervention by a human operator; yet the higher fidelity schemes require expensive multi-phase commit protocols and hence may not give adequate performance. Today, clustered three-tier database products are powerful solutions, but they negotiate these potential pitfalls in ways that preclude important classes of applications.

Our motivation is to show that a simple and remarkably inexpensive infrastructure can support clustered execution of a significant class of non-transactional services. The work reported here focuses on services that don't fit the transactional paradigm, typically for reasons of performance - ones that operate directly on in-memory data structures or simple non-transactional files. To simplify our task, we assume that these services are capable of handling out-of-order writes, and that processes implementing them experience only crash failures. As will be shown below, our assumptions hold for a very large group of applications.

*This work was supported by DARPA/IPTO under the SRS program and by the Rome Air Force Research Laboratory, AFRL/IF, under the Prometheus program. Additional support was provided by the NSF, AFOSR, and by Intel.

The SSA was built using epidemic (gossip) communication protocols in conjunction with a novel variant of the chain replication scheme which has evolved from the mechanism first proposed in [14]. Gossip based infrastructures are beneficial because they are:

- Simple to implement
- Rapidly self-stabilizing after disruptions
- Analytically appealing

This paper reports on the architecture and performance of the platform, and explores the limitations of its underlying techniques. More specifically, the experiments are designed to help us fully understand the fundamental properties of a single partitioned replicated service – and thus gain a firm grasp on the behavior of the SSA's building blocks. We defer for future work the full scale evaluation of multiple services deployed and running at the same time.

The SSA currently runs on a tightly coupled cluster of blade servers. We show that developers can tune parameters to trade overhead for speed of repair and we believe that our results validate the approach.

2 Application model

Our work focuses on datacenters supporting one or more services deployed within a cluster of compute nodes. For example, an *e-tailer* might implement a front-end service that builds web pages, parallelizing the task by dispatching sub-tasks to services to rank product popularity, maintain personal blogs, compute recommendations, track inventory, schedule shipping, and so forth. The front-end service would probably just be cloned, with identical replicas that build pages, while the back-end services might be partitioned into subservices for scalability using some key (for example product-id), and subservices cloned for fault-tolerance and load-balancing. This is a common model; Jim Gray and others have suggested that such a system be termed a "farm" consisting of "RAPS" (reliable array of partitioned services) and "RACS" (reliable array of cloned server processes; see Figure 1) [5].

Up to the present, this structure has arisen mostly in very large datacenters and is supported primarily in the context of three-tier database architectures. However, we believe that similar architectures will be needed more widely, because the need to tolerate heavy loads is increasingly ubiquitous, and economic considerations favor clustered solutions. For example, game servers require scalability for situations in which there are many users; military systems require scalability to support new generations of integrated applications; hospital automation is putting new demands on medical information subsystems. In a wide range of everyday settings, the rollout of SOAs and the ease of application integration they support will place services under growing load.

Our goal is to make it easy to build RAPS and RACS from traditional, non-replicated, web service applications designed for quick responsiveness. However, we also want to build the simplest platform capable of accomplishing this task.

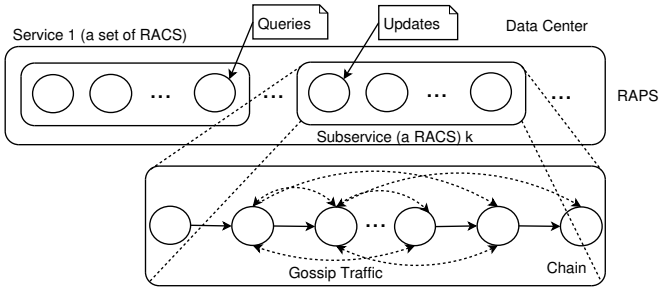

Figure 1: Datacenter abstractions.

2.1 Elements of the model

A *service* is simply an application that provides interfaces that manipulate objects of unspecified nature:

- A *query* operation reads some object and returns a computed value.
- An *update* operation modifies one or more objects.

One unusual assumption made in our work is that many services can process updates out of order. More precisely, we focus on services that

- Can respond "correctly" to queries even if some updates are temporarily missing.
- Converge into a state determined entirely by the set of updates, so that if two members of some subservice receive the same updates they will be in equivalent states, even if those updates were delivered in different orders.
- Guarantee idempotence: a reissued query or update returns an equivalent result.

What this amounts to is that the SSA should deliver updates as soon as it can even if they are not in order. Of course, one way that an application might process out of order updates is simply to delay processing them until it can sort them into order, but we believe that for many uses, it will be possible to act on an update or query immediately upon receiving it.

The SSA can support RAPS, RACS and combinations: a RAPS of RACS. A service that can be structured as a RAPS must have a *partitioning function* that can be used to map

each operation to the subservice that should execute it. Existing systems typically implement partitioning functions in one of two ways:

- *Client side.* The service exports its partitioning function, so that clients are able to locally implement the logic mapping requests to subservices. For example, the cluster might control the DNS, or could influence the creation of web pages by modifying URLs, so that clients will be directed to an appropriate subservice. In the limit, the servers might export actual code that the client runs.

- *Server side.* The partitioning logic is situated on a load balancing component resident in the server cluster. The load balancer sprays requests over the subservices in accordance with server logic.

The SSA supports the latter approach, offering a mechanism that assists the load-balancing component in tracking membership so that it can appropriately route queries and updates.

Finally, we assume that processes are fail-stop: should a failure occur, the process halts, and will eventually be detected as faulty. However, a failure may be transient: a process can become temporarily unavailable, but then restart and recover any missing updates.

2.2 Discussion

Our model is not completely general, and for this reason some discussion is needed. Consider the following example: we wish to support a scalable inventory service that receives updates corresponding to inventory consumption and re-stocking. Queries against such a service would compute and return an inventory count as of the time the query was processed. But inventory can change in real-time. The identical query, reissued a moment later, might yield a different result and yet both would be correct. Thus, responses reflecting *a reasonably current server state* are acceptable. On the other hand, a response reflecting a very stale state would be incorrect: A client should not be offered a promotional price on a plasma TV if the last unit was actually sold hours ago. In short, the inventory service should reflect as many updates as possible in the replies it gives to requests, but any reply is correct provided that it was based on a recent state. Below, we shall see that the SSA allows brief inconsistencies but that they can be limited to a few seconds.

Operations against the inventory service happen to be commutative, hence the service can process updates out of order. But many kinds of services can handle out of order updates, if for no other reason than that in many settings, each update is uniquely sequenced by its source, permitting the service to sort updates and to process queries against the sorted database.

Our group has held discussions with operators of several large datacenters, and concluded that many services have the kinds of properties just cited: ability to respond based on a reasonable current state, and to handle out-of-order updates. The SSA is a good match for "personalization" services ([11] [13] [15] [2]); these deal primarily with weakly consistent data, caching proxies, transformation proxies, and all sorts of services in which replies are intrinsically "noisy", such as services that report data gathered from remote sensors. Of course, a datacenter would also host some kinds of services ill-matched to our model, but because we are working with Web Services, services running on the SSA can easily interact with services that employ other solutions.

2.3 Consistency semantics

The SSA implements stochastic consistency semantics: an application will only observe an inconsistency if a fault occurs, and even then only for a period of time associated with our repair protocol, and only if it has the bad luck to query a node impacted by the failure. This window can be made small, so that applications are unlikely to observe a problem, or permitted to grow somewhat larger, depending upon the cost of inconsistency and the relative value, to the end-user, of faster response time versus lower risk of an observed fault. In the experimental work that follows, we measure these windows for scenarios representative of conditions that arise in realistic settings.

3 The SSA Framework

The basic operation of the SSA is as follows. As queries or updates are received in the cluster, they are passed through a partition mapping component, which directs the request to an appropriate RACS. We will use the term subservice rather than RACS in the remainder of the paper.

To create a subservice the developer must first implement a non-replicated service program. This is then cloned using the SSA platform. Each replica is placed on a separate node, and the replicas are then linked using TCP to create a chain (see Figure 1). We therefore have a 1:1 mapping between a subservice and a chain.

3.1 Gossip based chain replication

The replication scheme has evolved out of the chain replication mechanism first introduced in [14]. The original scheme was developed as a means of obtaining high throughput and availability for query and update requests without sacrificing strong consistency guarantees. As it

stands, the gossip based chain replication behaves in the following manner during normal operation – when nodes aren't failing or restarting:

- Update operations are forwarded to the head of the chain, where the request is processed using the local replica. The state changes are passed along down the chain to the next element, which in turn updates it's state and performs the same operation until the tail is reached.

- Queries can either be directed towards a randomly selected process in the group or to a specific one. The strongest consistency guarantee is acheived if all query operations are targeted at the tail of the chain node, which is the case for the vanilla chain replication scheme; however this eliminates the opportunity to load-ballance.

Faults and node restarts can disrupt the "primary" communication pattern of the SSA. If the head of a chain fails, update sources will need to discover a new head; if an inner node crashes the chain may break, and if the tail crashes, acks might not be sent back. During such periods, a subservice (or some of its members) can become inconsistent: processes will miss updates and hence queries will return outdated results. To repair these inconsistencies, the SSA implements a "secondary" update propagation mechanism: it uses *gossip* protocols to rapidly detect and repair inconsistencies, while simultaneously orchestrating repair of the chain. The gossip rate can be tuned: with a higher rate overheads rise but repair occurs more rapidly; with lower rates, repair is slower but overheads drop. The subsections that follow discuss the two core mechanisms in greater detail.

A second class of faults are transient and relate to the behavior of TCP when a node is subjected to stress, such as a burst of traffic. In these cases, the OS tends to lose packets and the effect is that TCP will impose congestion control mechanisms and choke back. Updates will cease to propagate down the chain, even though most of the nodes involved could still have ample capacity. Below, we will show that when such a problem arises, gossip will route data around the congested nodes, and will also deliver missed updates to the overloaded nodes when the problem ends.

In the original chain replication scheme the queries are directed to the tail of the chain. Since there is no additional epidemic communication, any update known to the tail is stable because it must first have been seen by all the members of the chain. To maintain such an invariant, the original paper includes mechanisms to ensure that a request really reaches the head of the chain, that updates are passed down the chain and applied in a strictly FIFO manner even when nodes fail and the chain is restructured, and that queries are sent to the tail of the chain. Strong consistency follows easily because query requests and update requests are processed *serially* at the tail element.

The gossip based chain replication weakens the model in two key respects. First, our solution might sometimes use the "wrong" head of the chain, for example if an update source is operating with inaccurate membership information. Second, updates might sometimes arrive out of order, for example if the chain is disrupted by a failure and some updates arrive via the gossip protocol. These changes substantially simplify the algorithm but they also weaken the properties of the solution. A less significant change is that we load-balance queries over the members of the chain; this weakens consistency, but in ways that seem to match the class of applications of interest, and has the potential to greatly improve query performance.

3.2 Epidemic dissemination

As noted earlier, SSA uses gossip to detect and repair the inconsistencies that can arise after a failure or when a node joins. The basic idea is simple: each process in the system runs a periodic local timer, without synchronization across processes. When a timer expires, a process computes a summary (also called a "digest") of its state[1] - a list of things that it "knows". This summary is sent to a randomly selected peer (or subset of peers). Quick delivery is more important than reliability for gossip messages, hence we favor UDP datagrams over TCP for this kind of communication. The recipient compares the gossiped information with its own state, identifying information known to the sender but unknown to itself, or known to it but apparently unknown to the sender. It then sends back a gossip reply (again, using an unreliable datagram protocol) containing information the sender might find useful and requesting information it lacks. Receiving this, the originator of the exchange will send a final message containing any data that was solicited by the receiver. Gossip messages are bounded in size.

Thus during a "round" each process will send a message, perhaps eliciting a reply, and perhaps will respond to that reply. In the worst case, a round results in 3 messages per process. The load imposed on the network will thus be linear in the number of processes, but any individual process will see a constant load (independent of system size).

The SSA gossips about membership, failures, recoveries and application state, using this information to initiate repairs. One form of repair involves disruption to a chain: if a fault breaks a chain or disables the head of a chain, gossip is used to detect the problem and repair involves designating a new head for the chain or establishing a new TCP connection bridging the gap. A second form of repair involves lost updates: if subservice A has a member m that knows

[1] We assume that all forms of information are uniquely named and that updates are ordered separately by each update source.

of update X and a member m' that lacks X, gossip can be used to detect this and m can then send X to m' directly (without waiting for the chain to be repaired).

Gossip is not a particularly fast protocol: in general, information requires $log(N)$ rounds of the protocol to reach N processes. On the other hand, if rounds occur frequently, the delay before information spreads to all members of a system may still be small, even in a large system. Moreover, gossip is astonishingly robust; there are exponentially many paths by which information can pass from point A to point B, hence almost any imaginable disruption short of a lasting partitioning failure can be overcome.

The gossip protocols implemented in the SSA have been designed specifically for use in our modified version of chain replication, and with the goal of running in large clusters or datacenters. Let σ be a group of processes, and let p be a process in that group $p, \in \sigma$. Each process has its own view of the group, denoted $view(p, \sigma)$. These views can lag reality (for example if a process joins or leaves), and different members might not have consistent views. Our work assumes that the network within a cluster does not partition, although there are low-probability failure patterns that could temporarily partition some subservice in a logical sense. Every round (time quanta, or *step interval*) process p chooses a random subset of a particular size $\xi \in view(p, \sigma)$ and commences a dialog with each process in the set ξ. The initial message is a compact state digest summarizing the state of the sender. The follow up dialog consists of an explicit request of missing *update operations*.

Several details of the epidemic protocols employed in the framework turned out to be important determinants of system performance and behavior:

- Suppose that a process disseminates information via epidemics about a subject s. Process p *gossips about* subject s a finite number of times, as long as subject s is *hot*, after which subject s is no longer gossiped about.

- Explicit requests for copies of missed messages are limited in size, to prevent a process that lagged behind or just joined from trying to catch up all at once, which would result in enormous messages and serious fluctuations in system load. Instead, such a process may need to catch up over many seconds.

- Explicit message requests are honored if the requested messages are still in the bounded buffers. Once a message has been delivered to the upper levels, and it has been expunged from the buffers located at the *gossiper* level, requests are simply ignored (the requesting process would have to try to find the missing data elsewhere). If data cannot be recovered, we signal this to the application by delivering an exception upcall (MISSING_DATA) and leave it to the application to decide how to handle the problem. The size of the buffers is configurable, but this rule implies that certain kinds of failures may be unrecoverable within the SSA.

- Digests are bounded in the number of messages they advertise about in one single datagram packet, and each round only a single digest is disseminated, even if the subset view selected (randomly) has cardinality greater than one.

- Messages that are potentially in transit are not retransmitted to requesting processes. For example if a process p makes an explicit request for a message m and the request lands at process q that has already sent p a copy of m in the recent past then m will not be retransmitted.

- A process creates a digest based upon all the messages received by means of any communication channels, not just the epidemics (e.g.: the messages received by FIFO chained channels).

- The message buffers are bounded, and once a message has been delivered by means of an upcall it is prone to be replaced by the replacement policy. The SSA implements several replacement policies: simple FIFO, most advertised message in digests, most disseminated message, most propagated message.

Although the SSA should work well on clusters with as many as thousands of nodes, companies like Google and Amazon reportedly operate centers with tens of thousands of machines in them, and are said to deploy some popular services on huge numbers of nodes. Were we to use the SSA in such settings, our gossip protocol might need to be revisited to ensure that messages do not become excessively large. One way to accomplish this might be to modify the epidemic protocol using spatial distributions to improve the performance [8]. Such an approach would let us restrict information to the vicinity of the nodes where it might be needed, in effect adding an additional layer of hierarchy to the architecture. We believe the required changes would be relatively minor.

3.2.1 Epidemic Analytical Model

One benefit of using gossip in the SSA is that we can use analytical methods to predict the behavior of a cluster, complementing our experimental work.

A basic result of epidemic theory states that simple epidemics *eventually* infect the entire population with probability 1. Moreover starting with a single infected site this is achieved in expected time proportional to the log of the

population size [3]. The protocol roughly falls under the category of a *push-pull* model, and the exact formula for it can be expressed as $log_2(n) + ln(n) + O(1)$ for large values of n, where n the number of sites participating in the epidemic spread.

Let p_i be the probability that a site remains susceptible (not touched by the *epidemic*) after the i^{th} round of the protocol. A site remains susceptible after the $i + 1^{th}$ round if it was susceptible after the i^{th} cycle and it is not contacted by any infectious site in the $i + 1^{th}$ round. The recurrence relation that we obtain is: $p_{i+1} = p_i \left(1 - \frac{1}{n}\right)^{n(1-p_i)}$.

Since infection starts with one site, for any randomly chosen site $p_0 = 1 - \frac{1}{n}$. Thus, as a function of the rate of gossip, we can predict the delay before a typical process that has been disrupted by a failure will learn about inconsistency introduced by the failure and can initiate repair. For example, if the model predicts that for a given gossip rate, a broken chain should be repaired within 1.5 seconds, one can anticipate that the disruption associated with a failure should be limited to the maximum number of updates that would be sent to a given subservice during a 1.5 second window. Moreover, if we know how large the typical update is, in bytes, and we know the size limit on data sent in response to explicit requests, we can predict the amount of time that will be needed to repair the resulting data inconsistency. These capabilities should help the developer parameterize the cluster to balance overhead for gossip against repair times desired by the application.

4 Membership

Some readers may be curious about what will seem to be a chicken-and-egg problem. On the one hand, we use gossip epidemics to propagate information about membership changes. Yet the gossip protocol uses membership information to select gossip peers. In practice, our solution starts with approximate membership information (extracted from a group management service component that list the nodes in the cluster and the rough mapping of services to those nodes) and then refines this with incremental updates.

A different concern relates to behavior when membership information is perceived differently at different nodes. Although such a condition may arise during transitional periods, these quickly resolve as additional rounds of gossip replace stale data with more accurate, recent information. In our experiments, we have never observed a membership inconsistency that persisted for longer than a few hundred milliseconds. The SSA is quite tolerant of short-lived inconsistencies.

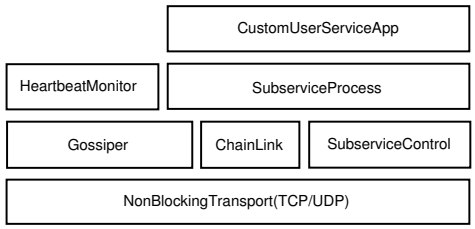

Figure 2: The component stack of one subservice process

4.1 Failure and recovery

Process failure detection is accomplished by means of two mechanisms:

- Detecting FIFO channels that break (in our case they are TCP channels with low value for the SO_TIMEOUT property).
- A gossip-based heartbeat detection mechanism.

Once a process is *deceased*, the information is propagated within the group in two ways. First, the process that has detected the membership change feeds the event description into the chain itself. This is delivered in chain order to every non-faulty process and where necessary, chain repair procedure is undertaken. Second, the same detector process starts up a backup gossip notification stream. This is a "fast dying" epidemic: it spreads rapidly but also dies out rapidly. The FIFO channels are rebuilt appropriately by the processes that identify themselves to be affected by the membership change, and the group converges to a stable configuration. Moreover, update sources can use this update to reconnect to a new head of any chain that may have lost its previous head as a consequence of the crash.

Similarly, if a process wants to join, it starts by sending a request to a random member of the group. As a consequence, the group member will commence a membership change protocol as described above. Again once all the nodes receive the membership event and update their view, convergence is achieved.

5 Implementation Details

The framework was implemented using the Java language and its non-blocking I/O stack. The system design was strongly influenced by prior work on high-performance services platforms, notably Welsh's SEDA architecture [16]: components are highly autonomous, decoupled, and event driven. There are only four distinct control threads in the component stack of a process (see Figure 2), namely one for the non blocking transport, the gossip element, the TCP chain and for the heartbeat component. To date, the SSA is roughly 8700 lines of code.

The tests reported here employ a hard-wired partitioning function. However, the SSA is a work in progress, and the full-fledged system will use a software partitioning mechanism based on the web services request invocation model. Although extracting the partitioning key from incoming requests will impose some overhead, we do not expect performance of the full-fledged system to deviate significantly from what is reported below.

6 Experimental Results and Validation

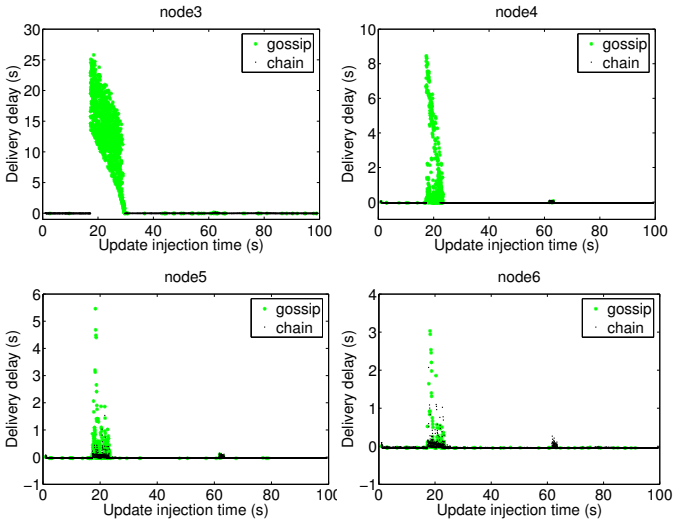

Figure 3: Update delay as seen by individual processes; nodes 3, 4, 5, and 6 in a chain. Gossip rate 1/160 digests/ms, update injection rate 1/80 updates/ms.

Our experiments were conducted using the SSA framework deployed on a tightly coupled homogeneous cluster of 64 processing nodes. The nodes are connected by two separate high speed Ethernet backbone planes. We experimented with several configurations; some placed the control traffic on a different switched Ethernet segment while others aggregated both the control traffic and the data traffic on the same segment. No significant differences were observed, but this may be because our control traffic consisted mainly of "fast-dying" epidemics, which put little stress on the communication channels. In the future we hope to explore scenarios that generate exceptionally heavy control traffic, which would allow us to explore the benefits of isolation of that traffic with respect to data traffic. In the interest of brevity we did not perform any experiments to evaluate the load balancing component but we plan to do so in the future.

All the experiments involved a single partitioned and replicated service. For ease of exposition, this service implements a simple wall-clock. The service itself maintains the time, with updates coming from client applications that

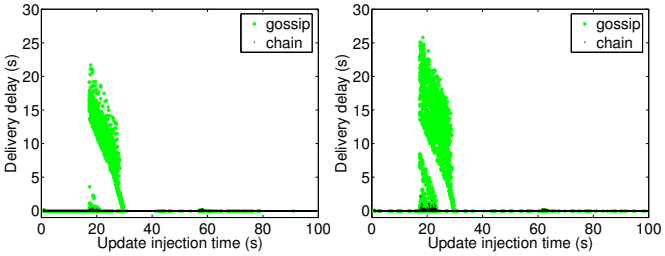

Figure 4: Update delay, as seen by the entire chain. Update injection rate 1/80 updates/ms, gossip rate left: 1/80 digests/ms, right: 1/160 digests/ms.

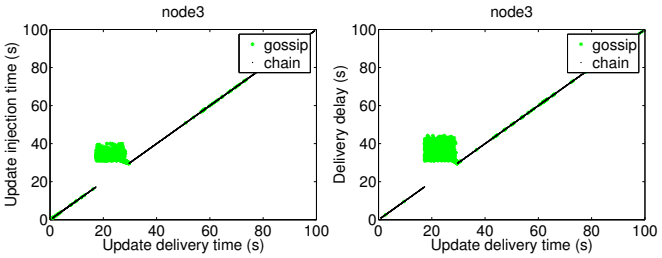

Figure 5: Update injection time against delivery time at *node3*. Update injection rate 1/80 updates/ms, gossip rate left: 1/80 digests/ms, right: 1/160 digests/ms.

read a high-quality clock and send the current value. As processes forward updates along the chain, they will track the clock themselves. All of our partitioning scenarios included at least four subservices, and each subservice included between 8 and 32 processes. We expect these to be typical cases for real deployments of the SSA. It should be noted that small subservice sizes (say less than 4) can result in degenerate behavior and are not appropriate configurations for the SSA architecture.

We created a 1:1 mapping between service processes and physical nodes, in order to avoid OS resource contention. We experimented with groups of 8, 12, 16, 32 processes; by convention the head of the chain for each group was called *node0*, and all update requests for a partition were routed towards this node. Since delivery delays in the chain were measured relative to *node0*, all the statistics pertaining to the group disregarded *node0*'s data.

We simulated two classes of failures:

- Transient failures - At some time t one process (typically *node3*) fails. The system must detect the failure, repair the broken FIFO channel, and continue operation. At time $t + \Delta$, the failed process recovers and rejoins the chain. The join protocol would run, and the previously failed node would become the new tail of the chain. The scenario is intended to model a common case in which the failure detection mechanism senses a transient problem (typically, a node that has become overloaded or is unresponsive for some other

reason, such as garbage collection), and does not respond to the heartbeat within the accepted window). By reconfiguring the chain, the load on node drops, and the problem will eventually resolve. It then requests a rejoin. A node crash that results in a reboot would result in similar behavior.

- Transient link congestion - In this case, all the nodes in the subservice remain operational, but one of them becomes overloaded, causing the TCP link to the upstream node to become congested and starving downstream nodes, which begin to miss updates. This scenario models a behavior common in experiments on our cluster: when a node becomes very busy or the communication subsystem becomes heavily loaded, TCP at the node upstream from it will sense congestion and reduce its window size. If the impacted node is in the middle of the chain, it ceases to relay updates (or does so after long delays), hence downstream nodes fall behind. In effect, the chain replication scheme slows to a crawl. Here, the SSA benefits from its gossip repair mechanisms, which route missing updates around the slow node (and later, route them to that node, when it recovers and needs to repair its state). Moreover, knowing that gossip will kick in, an upstream node can deliberately drop updates on congested TCP connections.

We used our wall-clock service to evaluate the behavior of the overall system in various scenarios and with different parameters. A stream of updates of various rates is injected into the head of the chain (*node0*) for groups of nodes. At a pre-established point in time, a *victim* node receives a command that forces it to halt (the node continues to listen for commands that would restart it). This is accomplished by having *node0* send a crash command to the victim node once a certain number of updates were injected into the chain. At this time, t, the victim node will stop participating in the normal protocol and will handle only wakeup commands from this moment onwards. The chain detects the failure, repairs and announces the membership change. After a number of updates have been injected since the crash command was issued, *node0* sends a wakeup command to the victim node. At this time, $t+\Delta$, the victim node rejoins the group, it has to catch up by obtaining copies of updates that it has missed. We experimentally determined that 8 repetitions of each experiment were enough to yield accurate measurements with low variance.

Figure 3 shows the update delivery delay for a set of four consecutive nodes in a chain, starting with the victim node, *node3*. The chain length is 8, and we report on a gossip rate of 1 digest every 160 milliseconds at a steady update injection rate of 1 update every 80 milliseconds. There are three anomalies that can be seen on the graphs. The first one is experienced by the victim node for updates injected between 16 and 32 seconds after the start of the experiment. The second is experienced by all the other nodes for update messages injected at around 20 seconds after the start of the experiment, while the third one is a smaller mixed burst for updates injected at 65 seconds into the experiment. Note that the y-axes have different scales to observe how the system handles the transient failure better, therefore the third anomaly appears to grow with the chain distance from the victim node. The growth is not significant, since the cause of this anomaly is an artifact of Java's garbage collection mechanism kicking in. As can be noted, *node3* performed recovery for the updates it has missed during the period it was down, because the chain delivers new updates at the moment of rejoin, all past updates were solely recovered by means of epidemics. The second anomaly that shows up in the update delivery delay for the nodes downstream from the victim node reflects the period when the chain is broken. During the time it took for the failure detection mechanism to declare the node deceased, to start up the membership change protocol, and for the membership information to propagate, the chain is interrupted between *node2* and *node4*, and hence the updates circumvent the gap by means of gossip. Updates can bypass nodes in the chain using the gossip as it can be seen in the figure, but this phenomenon is less likely as the node receiving the update is farther away downstream from the victim node.

Figure 4 contains an aggregated view of the data in Figure 3 for the entire chain, at gossip rates of 1 digest every 80 milliseconds and 160 milliseconds showing that the behavior of the scheme is not a fluke. Note that the delay of the updates delivered at the victim node is significantly larger than that of the nodes downstream of it in the chain.

We observed that even with sufficiently high gossip rate, the only node to experience any significant inconsistency window is the node that failed. Note that when the failed node rejoins, queries are performed against its data before it has time to fully recover. Once the chain is restored, all new updates are received. There were rare cases when gossip circumvented the chain replication even though the chain was not broken, but this happened only for gossip rates close to the update injection rate. Later in this section we will show that even with these rapid repairs, the gossip overhead is actually low (less than 0.07% of the messages were delivered by gossip ahead of the chain for gossip rate identical to the update injection rate).

Figure 5 contains a plot of update injection time against update delivery time for the victim node. Ideally this is a straight line because of chain replication. Note that once the victim node recovers, it gracefully catches up and does so quickly for both gossip rates identical and half the update injection rate.

Now consider the link congestion case (Figure 6). In

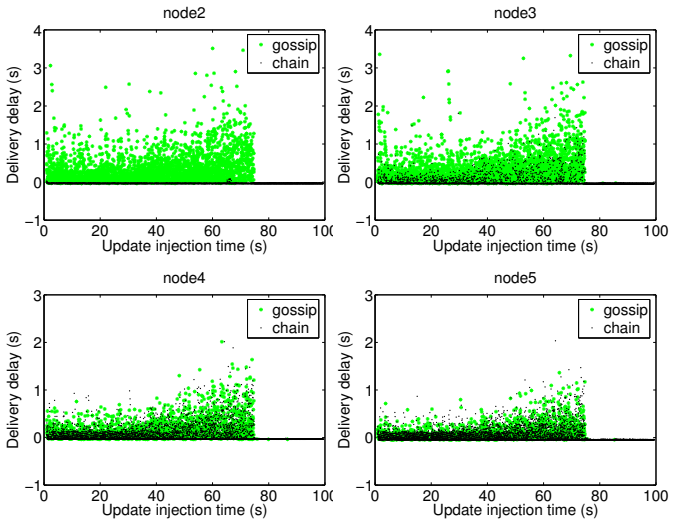

Figure 6: Update delay as seen by individual processes during *persistent link congestion – node2* drops 40% updates on upstream and downstream FIFO channels; nodes 2, 3, 4, and 5 in a chain. Gossip rate 1/160 digests/ms, injection rate 1/80 updates/ms.

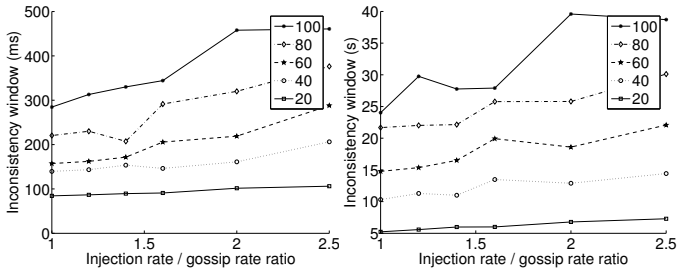

Figure 7: Inconsistency window against the ratio between injection rate and gossip rate (left - average, right - maximum), different update injection delay.

this scenario, we modeled *node2*'s overload by dropping updates on its inbound and outbound FIFO channels according to a random distribution throughout the first three quarters of the experiment (we experimented with 25%, 30%, 40%, 50%, 75% and we report on 40%). Updates that were initially dropped and eventually made their way through gossip could later be sent via FIFO channels as shown by the increasingly large density of dark ("FIFO delivered") plots closer to the tail of the chain. As before note that the y-axes have different scales to observe the delays better. The figures show that even for a gossip rate half the injection rate (recall that this is the rate at which digests, not messages, are exchanged between two or more processes) the epidemics could deliver messages with a delay of about 3.5-4s for nodes 2 and 3, and at most 2s for the rest of the chain during a congestion that took 75 seconds. The plot also shows that delays increased with time, therefore if congestion may span large periods of time, the gossip rate must be carefully tuned to compensate for the losses induced by the

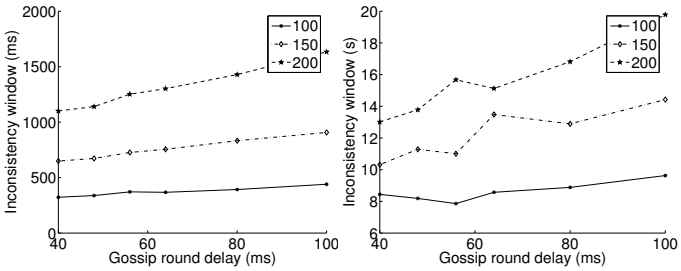

Figure 8: Inconsistency window against gossip rate at the failed node (left - average, right - maximum), update injection rate 1/40 updates/ms, different Δ - time between node failure and rejoin as number of consecutive updates missed by the victim node.

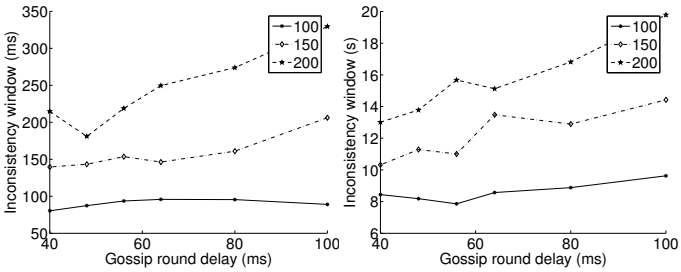

Figure 9: Inconsistency window against gossip rate for the whole chain (left - average, right - maximum), update injection rate 1/40 updates/ms, different Δ - time between node failure and rejoin as number of consecutive updates missed by the victim node.

congested TCP channels.

The second round of experiments quantified the average and maximum *inconsistency window* for a service (the same *wall-clock*), under various update injection rates and gossip rates respectively. We define the inconsistency window as the time interval during which queries against the service return a stale value. Figure 7 shows that the inconsistency window grows slowly as the gap between the update injection rate and the gossip rate widens (the graph's x axis represents the ratio between the update injection rate and gossip rate). This confirms that epidemics are a robust tunable mechanism providing *graceful degradation*. Likewise, the inconsistency window shifts in accordance with the update injection rate. Finally, notice that the difference between the maximum inconsistency window and the average inconsistency window is two orders of magnitude. This reflects the degree to which the victim node lags the other nodes during the period before it has fully caught up.

Next we evaluated the inconsistency window of a service running at a particular update rate, and for three different intervals in which the victim node is halted. Figures 8 and 9 show average and maximum inconsistency windows for both the *victim* and for the other processes of one subservice. As expected, the more messages the victim node needs to recover, the larger the inconsistency window. Again the difference between the average and maximum in-

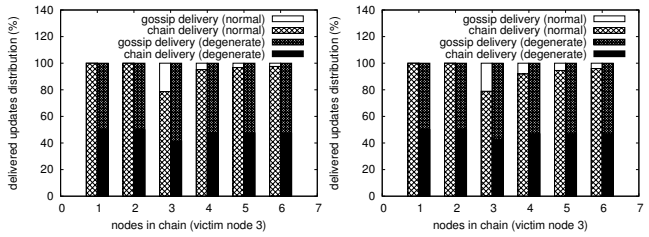

Figure 10: Delivery distribution for a chain, injection rate 10 updates/s. Gossip rate left figure: 1/140 digests/ms, right figure: 1/250 digests/ms. On each graph left bars denote transient failure (normal), right bars denote a transient failure corroborated with a link congestion phenomenon modeled by 50% message drop on the adjacent FIFO channels of *node2* (degenerate).

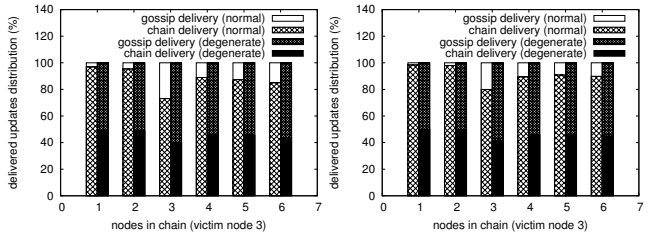

Figure 11: Delivery distribution for a chain, injection rate 50 updates/s. Gossip rate left: 1/28 digests/ms, right: 1/50 digests/ms. Same as Figure 10.

consistency windows is slightly more than an order of magnitude, and this is attributable to the victim node – observe that the two graphs denoting the maximum inconsistency windows for the victim node and for the entire chain are identical, which means that clients perceiving significant inconsistency are the ones that are querying the victim node while it is still recovering state.

Finally we performed a set of experiments to determine the distribution of messages delivered by the chain vs delivered by gossip. As before, one transient failure affects the wall-clock service. The runs are eight times longer than the runs before (both in total experiment time and time the victim node is halted).

Figure 10 and Figure 11 show the number of messages delivered by the chain replication mechanism and the ones delivered by the epidemics, for each of the nodes in a chain (again we omitted the head of the chain node because its behavior is not representative, and in this experiment we have chains of length 7). Obviously, *node3*, the victim, delivered updates by means of the gossip repair mechanism. As the nodes get further away from the victim node, more of the messages were delivered by means of the chain, because the repair mechanism relinked the chain and chain replication began to function normally. The speed with which the chain is restored depends on the rate of the fast-dying control gossip, and on the responsiveness of the failure detection mechanism.

We found that less than 0.07% of the messages were delivered by gossip for the nodes to the left of the victim. This confirms that gossip rarely is used to circumvent chain replication in the normal case.

A peculiar effect is noticeable in Figure 11, in that more messages are delivered via gossip, even in the prefix part of the chain, although the effect is also evident in the suffix. It is more significant on the left hand side figure, where the gossip rate is higher. Because we observed this phenomenon only with update rates of 50 updates/s or more, we suspect that the network stack is more efficient in dealing with UDP packets then with TCP ones under heavy load.

7 Future development

The current SSA implementation uses gossip in situations where faster notifications might be helpful. For example, we believe that when a node fails or joins, it would be useful to spread the news as quickly as possible. We realize that for some particular tasks gossip could be done more efficiently. We are therefore exploring the use of IP multicast for dissemination of urgent information as long as the physical nodes are not on a public network segment. Additionally, we plan to include support for the partitioning of the services by means of registering partition function handlers with a global data-center registry. Finally, as mentioned previously, we have implemented only the server side load balancing scheme. We are considering ways to extend our approach for use in settings where partitioning is done on the client side, and hence client-side access to subservice membership information is needed.

We are also developing a GUI assisted automated web service deployment tool, focused on web service applications. Developers could simply drop a WSDL service description, tune a set / subset of parameters, and the system will generate a XML description that can be used later on to actually deploy the service automatically. The service will be partitioned, replicated, and deployed on the fly on top of the processing nodes.

7.1 Scaling up

To turn the SSA into a full scale platform, one of the immediate future challenges is the necessity of evaluating a full "RAPS of RACS" deployment. Multiple partitioned and cloned services running on our tightly coupled cluster would lead to a series of other issues that should be investigated, such as:

- Placement – given a set of services, how to place the clones on physical nodes in order to satisfy certain constraints (e.g. best response time).

- Caching placement – deciding if some services would benefit if they are fitted with response caches, and ultimately placing the cache components in a smart way.
- Service co-location – placing multiple service clones on the same physical node to exploit fast IPC communication as opposed to network messages if the benefits overweigh the cost incurred by resource contention on the shared host.
- Management tools – developing tools that monitor service properties such as response time, and react accordingly (e.g. by restarting new clones).
- Using VMM tricks – virtual machines can be used to migrate transparently a collection of services on a different physical processor, or provide isolation guarantees between co-located services.

8 Related Work

Broadly, the SSA can be seen as a platform that leverages tradeoffs between weaker consistency (with a compensating gossip repair mechanism) for higher availability and simplicity. This is an old idea first explored in the Grapevine [1] system, and later in systems like Bayou [4] which offer a broad operational spectrum between strong (ACID in the distributed database cases) and weak consistency. Several database and distributed systems take advantage of the same tradeoff, for example allowing multiple updates to occur simultaneously at distinct replicas by specifying a maximum accepted deviation from strong consistency (continuous models) [17] [18], tolerating a bounded number of consistency violations to increase concurrency of transactions [9], or replication according to the need-to-know principle [10]. Our work on the SSA is the first to apply such thinking to a cluster computing environment.

The TACC [6] platform was designed to provide a cluster based environment for scalable Internet services of the sort used in web servers, caching proxies and transformation proxies. Service components are controlled by a front end machine that acts as a request dispatcher and incorporates the load balancing and restart logics. If back-end processes are detected to have failed, new processes are forked to take over the load. TACC "workers" can be composed to address more complex tasks (TACC stands for *transformation, aggregation, caching and customization*). SSA can be seen as revisiting these architectural ideas in conjunction with chain replication.

Database management systems (DBMS) simplify applications, have long supported clustered architectures, and were the first systems to exploit the style of partitioning that leads to a RAPS of RACS solution. However, unlike the SSA, most database systems adhere closely to the ACID model, at potentially high cost in terms of reduced availability during faults. Jim Gray et. al. discuss this problem in [5], ultimately arguing for precisely the weak update model that we adopted here.

Application servers like the J2EE [12] offer persistent state support by wrapping soft state business logic components on top of a relational or object-oriented database. They also target large-scale highly available services, and hence we believe they could benefit from SSA-hosted services. In a similar vein, the Ninja [7] framework makes it easy to create robust scalable services. Ninja is arguably more flexible than application servers in that it performs connection management and automatically partitions and replicates persistent state, but the framework takes a different tiered approach to services based on bases, active proxies and units, and represents shared state by means of distributed data structures.

9 Conclusion

Our paper presents the Scalable Services Architecture, a new platform for porting a large class of service-oriented applications onto clusters. The SSA was designed to be as simple as possible, and at the core uses just two primitive mechanisms: TCP chains that support a variant of chain replication, and gossip epidemics which are used to manage configuration data and initiate repair after failures. With appropriate parameter settings (specifically, given a gossip rate that is sufficiently fast relative to the update rates seen in the cluster), we find that the SSA can rapidly and automatically reconfigure itself after a failure and can rapidly repair data inconsistencies that arise during the period when the cluster configuration was still disrupted. Our goal is to make the software available to a general user community in 2006.

10 Acknowledgments

The authors are grateful to the research team at AFRL in Rome, NY, for their help in understanding the challenges of using Service Oriented Architectures in large scale settings, and to the researchers at Amazon.com, Google and Yahoo! for helping us understand the architectures employed in very large data centers.

References

[1] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine, An Exercise in Distributed Computing. *Communications of the ACM*, 25(4):260 – 274, 1982.

[2] A. Datta, K. Dutta, D. VanderMeer, K. Ramamritham, and S. B. Navathe. An architecture to support scalable online personalization in the web. *The International Journal on Very Large Data Bases*, 10(1):104–117, August 2001.

[3] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1 – 12, Vancouver, British Columbia, Canada, 1987.

[4] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou Architecture: Support for Data Sharing among Mobile Users. In *IEEE Workshop on Mobile Computing Systems & Applications*, 1994.

[5] B. Devlin, J. Gray, B. Laing, and G. Spix. Scalability Terminology: Farms, Clones, Partitions, Packs, RACS and RAPS. *CoRR*, cs.AR/9912010, 1999.

[6] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 78–91, New York, NY, USA, 1997. ACM Press.

[7] S. D. Gribble, M. Welsh, R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. H. Katz, Z. M. Mao, S. Ross, B. Zhao, and R. C. Holte. The Ninja architecture for robust Internet-scale systems and services. *Comput. Networks*, 35(4):473–497, 2001.

[8] D. Kempe, J. Kleinberg, and A. Demers. Spatial gossip and resource location protocols. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 163 – 172, Hersonissos, Greece, 2001.

[9] N. Krishnakumar and A. J. Bernstein. Bounded ignorance: a technique for increasing concurrency in a replicated system. *ACM Transactions on Database Systems (TODS)*, 19(4):586 – 625, December 1994.

[10] R. Lenz. Adaptive distributed data management with weak consistent replicated data. In *Proceedings of the 1996 ACM symposium on Applied Computing*, pages 178 – 185, Philadelphia, Pennsylvania, United States, 1996.

[11] A. K. Markus Keidl. Towards context-aware adaptable web services. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers and posters*, pages 55 – 65, New York, NY, USA, 2004.

[12] Sun Microsystems. The Java 2 Platform, Enterprise Edition (J2EE), 1999. http://java.sun.com/j2ee/.

[13] L. Suryanarayana and J. Hjelm. Profiles for the situated web. In *Proceedings of the eleventh international conference on World Wide Web*, pages 200 – 209, Honolulu, Hawaii, USA, 2002.

[14] R. van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *Sixth Symposium on Operating Systems Design and Implementation (OSDI 04)*, San Francisco, CA, December 2004.

[15] D. VanderMeer, K. Dutta, A. Datta, K. Ramamritham, and S. B. Navathe. Enabling scalable online personalization on the Web. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 185 – 196, Minneapolis, Minnesota, United States, 2000.

[16] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.

[17] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 29 – 42, Banff, Alberta, Canada, 2001.

[18] H. Yu and A. Vahdat. Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services. *ACM Transactions on Computer Systems (TOCS)*, 20(2):239 – 282, August 2002.