Tempest: Soft State Replication in the Service Tier*

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

Abstract

Soft state in the middle tier is key to enabling scalable and responsive three tier service architectures. While soft-state can be reconstructed upon failure, replicating it across multiple service instances is critical for rapid fail-over and high availability. Current techniques for storing and managing replicated soft state require mapping data structures to different abstractions such as database records, which can be difficult and introduce inefficiencies. Tempest is a system that provides programmers with data structures that look very similar to conventional Java Collections but are automatically replicated. We evaluate Tempest against alternatives such as in-memory databases and we show that Tempest does scale well in real world service architectures.

1 Introduction

Service-Oriented Architectures (SOAs) have emerged as the paradigm of choice for structuring large datacenter-hosted systems. Most contemporary large-scale applications are built as SOAs: online stores, search engines, enterprise software and financial infrastructure are some examples. The canonical design for such systems is a three-tier architecture: a first tier load-balancing proxies sends requests to a second tier of state-less service logic which in turn accesses and updates a third tier of durable databases or filesystems.

Soft state in the service tier is key to building highly responsive and scalable SOAs. Soft state is characterized as data that does not have to be stored durably and can be reconstructed at some cost [33, 18, 14] — examples include short-lived user sessions, stored aggregates and transformations on large datasets, and general purpose write-through caches for files and database records. Third-tier constructs are extremely fault-tolerant but correspondingly slow and expensive, and soft state is typically used to limit their role in performance-critical data paths. For example, the developer of an online travel service might use the memory of the service instance to store intermediate choices made by a user during the booking process, so that only the final sale transaction — a small fraction of all user activity — hits the third-tier database.

In this paper, we consider the availability of soft state stored in the service tier. When soft state is lost or made unavailable due to service instance crashes and overloads, reconstructing it through user interaction or third-tier re-access can be expensive in time and resources. Replicating soft state provides applications with two critical capabilities: rapid fail-over to other instances during crashes and fine-grained load-balancing across instances to prevent overload [33]. For example, a user request can be transparently redirected during a crash or overload to a different service instance that has up-to-date session context, without requiring her to log in again.

Many options exist for adding high availability to programs that manipulate soft state and these can be broadly classified into three categories: clustered application servers [3], messaging toolkits, and collocated in-memory databases. However, all these options require the developer to write code in "state-aware" ways, mapping data structures to special replication-aware containers, replicated state-machine stores and database-style records, respectively. Such mapping needs to be done carefully to avoid performance issues — for example, storing fine-grained variables in a database could result in severe locking contention [1]. However the natural way for programmers to store and manage soft state in a service is to use conventional in-memory data structures such as hash tables or linked lists.

In this paper, we present Tempest, a Java runtime library designed for easy storage and replication of service-level soft state. Tempest provides developers with *TempestCollections*: custom data structures that look similar to conven-

*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.

tional Java Collections [27]. Data stored in these structures is transparently replicated across multiple machines, providing fail-over and load-balancing for soft state with zero extra effort by the developer. Under the hood, Tempest uses a fast but unreliable IP multicast operation to spread/broadcast invocations to multiple service instances and then uses gossip-based reconciliation to maintain replica consistency in the face of faults and overloads. Additional adaptive mechanisms are used to maintain high responsiveness during failures.

High-performance in-memory databases are used extensively to store soft state in currently deployed systems [5, 22] and we show that Tempest outperforms them by more than an order of magnitude in large-scale SOA settings. Real-world SOAs often have many services interacting with each other to perform complex tasks — for example, a first-tier front-end could contact a hundred second-tier services to assemble a webpage [15]. Further, each service is potentially contacted in parallel by a large number of load-balancing first-tier front-ends. Tempest scales in both the number of front-ends querying a single service and the number of services being queried by a single front-end. In contrast, in-memory databases fail to scale in these dimensions due to contention, large latency variations and inefficiencies in cross-process interactions between the service and the database.

Accordingly, the contributions of this paper are as follows:

- We present a Java runtime library that exposes data structures to programmers that are transparently replicated across multiple nodes.
- We describe the gossip-based mechanisms used within the system for rapidly replicating data and speeding-up access to it.
- We evaluate Tempest on two datacenter-style testbeds — the Emulab testbed at Utah [30] and a 255 node cluster at Cornell. We show that Tempest maintains rapid responsiveness under heavy loads and outperforms in-memory and on-disk databases while scaling in two important dimensions — the number of front-ends accessing a single service and the number of services composing a single response.

The remainder of this paper is structured as follows: Section 2 describes the interface and semantics provided by TempestCollections to service developers. Section 3 describes the protocols and mechanisms used by Tempest to implement the TempestCollection abstraction, and Section 4 provides an evaluation of Tempest on datacenter testbeds.

2 The TempestCollection Abstraction

2.1 Service Model

Services are self-contained entities designed to support interoperable machine to machine interaction over a network [31]. Each service exposes an API through which a set of methods can be invoked by clients, and each service offers its own quality of service and availability guarantees. Take for example the interface of a shopping cart service as listed in Figure 1.

```
public interface ShoppingCartIF extends Iterable {
    update int add(String itemSymbol, int count);
    update int remove(String itemSymbol, int count);
    update int update(String itemSymbol, int count);
    read int check(String itemSymbol);
}
```

Figure 1. 'Shopping Cart' service interface.

Add, remove and update do the obvious things; these are classified as update operations because they change state. Check is a read operation; it retrieves the current number of items in the shopping cart for the symbol of interest. Clients issue `add/remove/update` and `check` requests against the service; the service processes each request and in return sends back a reply. This simple example can be trivially extended to services like item browsing history, product availability, product rating, or caching services.

In this work we assume that business logic is collocated with soft state stored in the memory of the service instance; as mentioned before, this is a natural design choice for applications requiring scalability and responsiveness. For example, storing shopping cart information in-memory allows the service to handle a large quantity of browsing traffic that otherwise would have reached the third tier. A developer implementing the shopping cart service in Java could use different data structures to store the state of the cart; a natural way would involve using a hash table to store mappings between item identifiers and corresponding counts.

Service state is modified by updates sent to it through its *interface* — in the conventional three-tier setup, this refers to database state hidden by the service, but in our case it includes soft state maintained by the service. In our shopping cart example, items are added to or subtracted from the cart.

The implementation of a service as a Java application running on a single node is obviously prone to crashes, overloads and slowdowns. Our goal is to transparently replicate a service on multiple nodes while retaining the programming ease and familiarity of Java's built-in Collection data structures. Accordingly, we provide developers with *TempestCollections* — data structures very similar to vanilla Collections but providing automatic replication of the data stored in them.

2.2 TempestCollection: Syntax and Semantics

TempestCollections are syntactically identical to standard Java Collections. For example, a TempestHashtable exposes `get` and `put` methods while a TempestSet has `add`, `remove` methods. Like most Java Collections, objects stored in a TempestCollection cannot be modified in place. For example, to change a field inside an Object stored in a TempestSet, the programmer would have to remove the Object, modify it and then re-insert it into the set.

This is a very common programming idiom within the Java Collections framework. For example, Java TreeSets provide ordered iteration over their elements, and changing the value of an item in-place can push the TreeSet into an inconsistent state by modifying the outcome of compare operations. Programmers are expected to instead change values by removal, modification and re-insertion if they want the TreeSet to remain consistent and ordered. In general, many Collections involve comparisons through `equals` and `compareTo` — such as HashMaps, TreeSets or HashSets — and do not allow safe in-place modification of objects stored within them. In this respect, TempestCollections offer identical semantics.

To prevent accidental modification of stored items, TempestCollections implement *by-value* parameter passing. Deep clones of added Objects are stored within the TempestCollection and clones of stored Objects are returned by accessor functions. For example, calling `put(`K`,` A`)` on a TempestHashMap will result in a clone A' being stored within the collection, and calling `get(`K`)` will return A'' to the programmer.

However, the Tempest runtime can alter the contents of TempestCollections by adding and / or removing items to keep collections consistent across replicas. TempestCollections provide *eventual consistency* — all replicas converge to the same set of objects [10, 8]. An implication of this model is that the programmer is not provided with ACID transactions; however, this is not a major limitation for soft state management [8]. In many soft state applications, data stored within structures is naturally immutable — for instance, a browsing history service that stores a list of item identifiers. For others, updates do not depend on current state — for example, a map from users identifiers to last viewed items. Even if the soft state is manipulated with arbitrary operations, it is expected by definition to not have strong semantics — the user is always asked to verify the contents of a shopping cart or the final itinerary of a travel plan before committing to it.

To summarize, TempestCollections are data structures exposing interfaces identical to those in the Java Collections framework and supporting similar semantics by not allowing in-place modifications of stored Objects. The sole deviation from the Java Collections framework – aside the weak consistency implications – is that Tempest enforces Object immutability by passing parameters by-value — a side effect of this is the possibility for services to operate on stale data.

3 Tempest Architecture

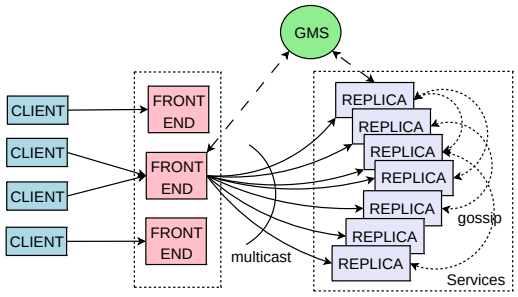

Figure 2. Tempest architecture.

In this section we describe the mechanisms used to implement replicated TempestCollections. Tempest services reside on second-tier *servers*; a single server represents the platform configuration on a single computer and might run several services. A service instance stores data in one or more TempestCollections. Multiple instances of a service execute across different servers, and invocations to this service are sent by first-tier front-ends to all the service instances — see Figure 2 (front end initiates a multicast to the servers that contain replicas of the same service instance).

The life-cycle of a Tempest invocation begins when a client sends a request over the Internet to the datacenter, which gets load balanced to a web-facing front-end node. The front-end is then responsible for contacting a set of services and aggregating individual service responses into a composite result that it returns to the client. Front-ends use IP multicast to perform web-service invocations on service instances, allowing very rapid communication in the general case; when multicast packets are dropped or delivered at different orders across instances, gossip-based reconciliation is used to repair gaps and errors in the TempestCollections maintained by them.

3.1 Client Invocations

When a client request enters the datacenter at a front-end, it's tagged with a web service invocation identifier (wsiid) consisting of a tuple containing the front-end node identifier and sequence number. Front-end node identifiers are obtained by applying the SHA1 consistent hash function over the front-end's IP address and port pair. Each Tempest request is thus uniquely identified by its wsiid.

As mentioned previously, Tempest differentiates between updates and queries or reads. For updates, Tempest uses IP multicast to send the operation directly to the full set of Tempest servers that hold replicas of the service for which the requests were intended. A hashing mechanism is employed to determine which server instance is responsible for replying. In the absence of message loss, which common, IP multicast within datacenters is reliable and ordered.

For read requests, front-ends use an adaptive querying mechanism. Each front-end periodically multicasts a beacon to each service and waits for unicast responses from each instance. It selects the k instances that respond first — where k is the *redundant querying* parameter — and subsequently directs service read invocations to these instances.

3.2 The Tempest Gossip Mechanism

Tempest is designed under the assumption that the multicast protocol used might not be fully reliable or might recover lost packets at high latencies. If some replicas miss an update, they can become inconsistent. Tempest uses a gossip protocol to repair these kinds of inconsistencies rapidly. Servers use a custom tailored gossip protocol to reconcile differences between the TempestCollection replicas.

Tempest keeps track of all the operations performed at the data structure boundary — this is possible due to our *by-value* semantics of altering the collections. When an object is added to a collection, it is annotated with the web service invocation identifier of the corresponding invocation; when an object is removed from a collection, a death certificate for it is created and annotated with the wsiid. A death certificate is simply a means of retaining the information necessary to identify which objects were removed from a collection. In particular each TempestCollection keeps a history of the removed objects in an internal private data structure not exposed via the standard interface.

The anti-entropy mechanism works by having each server "gossip about" the sets of web service invocation identifiers (wsiids) that annotated objects in TempestCollections. Suppose for example that during one gossip round we have two service replicas r_1 and r_2 respectively engaged in an exchange; let their *sets* of wsiids be denoted by $w(r_1)$ and $w(r_1)$. If $w(r_1) = w(r_2)$ no action is taken, otherwise some invocations were missed by one (or both) and a "reconciliation" phase is triggered:

- If $w(r_1) \subset w(r_2)$ then r_1 missed invocations and holds a stale version of the state – as a result r_1 retrieves from r_2 the objects and death certificates annotated with the wsiids from the set $w(r_2) \setminus w(r_1)$. Objects referred by the death certificates are removed, newly received objects are added; also r_1's set of wsiids is updated accordingly: $w(r_1) \leftarrow w(r_2)$.
- If $w(r_1) \not\subset w(r_2)$ and $|w(r_1)| \neq |w(r_2)|$ (the sets have different cardinality) both replicas have missed at least one update each, therefore to make progress it is safe for any of the replicas to assume the other replica's state – without violating the "eventual consistency" guarantees offered by the system. Choose the replica that has the smaller w set – let it be r_1 without loss of generality; r_1 performs the following steps:
  - For every identifier i in the set $w(r_1) \setminus w(r_2)$, if i annotates an object then the object is discarded, otherwise if i annotates a death certificate the object referred by the death certificate is "resurrected" (added back to the collection).
  - Fetch from r_2 all objects and death certificates annotated with identifiers from the set $w(r_2) \setminus w(r_1)$. Remove objects referred by the death certificates, add the new objects, and update $w(r_1) \leftarrow w(r_2)$. Here we used the heuristic of discarding the state of the replica that received less invocations, however one can imagine other criteria.
- If $w(r_1) \not\subset w(r_2)$ and $|w(r_1)| = |w(r_2)|$ then the initiator of the gossip round between r_1 and r_2 "plays the role" of the replica with the smaller w and performs the same operations as in the previous case.

An upcall is provided such that the service developer is notified when a gossip reconciliation was triggered.

If no new invocations are issued against the system, and if no permanent network partition that splits the servers into two or more disjoint communication parties occurs the TempestCollection replicas will eventually contain identical elements with probability 1.0 [9].

During a gossip round, there can never be more than 3 messages issued per process (by protocol design). Currently the sets of web service identifiers are monotonically increasing as new invocations are issued, therefore gossip messages size increases with time. We are working on a method for garbage collecting the stale wsiids by appending an epoch number at wsiid generation time — tempest servers will discard wsiids that are more than δ epochs old for some choice of parameter δ. Another option is to use efficient set reconciliation methods like the ones in [20, 4].

The strength of gossip protocols lies in their simplicity, the fact that they are robust (there are exponentially many paths information can travel in between two endpoints), and the ease with which they can be tuned to trade speed of delivery against resource consumption. The epidemic protocols implemented in Tempest evolved out of our previous work on simple primitive mechanisms that enable scalable services architectures in the context of large-scale datacenters. A more thorough description of the basic protocols and some of the optimizations can be found in [19].

3.3 Membership and Failure Detection

Membership in Tempest is handled by the Group Membership Service (GMS), which maintains the mapping between servers and service replicas. In addition, it also acts as a UDDI (Universal Description Discovery and Integration) registry providing appropriate WSDL (Web Services Description Language) descriptions for the services deployed on Tempest servers. The GMS also fills the administrator role for Tempest servers, monitoring the overall stress and spawning new servers to match the load imposed on the system. Finally, it monitors components to detect failures and adapt the configuration.

Tempest assumes that processes fail by crashing and can be reliably detected as faulty by timeout. Accordingly, Tempest processes monitor the peers with which they interact using a secondary gossip-based heartbeat mechanism. Processes that are thought to be deceased are reported to the GMS, which waits for f distinct suspicions before actually declaring it *deceased*. It then updates and disseminates group membership information to all interested parties. While in our experiments the GMS is hosted on a single high-end node, in a datacenter it could potentially be replicated and partitioned across multiple machines for scalability and fault-tolerance.

3.4 Node Recovery and Checkpointing

TempestCollections are automatically checkpointed. Periodically, each Tempest server batches the items in each TempestCollection and writes them atomically to disk. When a node crashes and reboots, upon starting the Tempest server, the services are brought up to date with the state that was last written to disk before the crash.

When a server is newly spawned, or when a server that has been unavailable for a period of time missed many updates, Tempest employs a bulk transfer mechanism to bring the server up to date. In such cases, a source server is selected and the contents of the relevant TempestCollections are transmitted over a TCP connection. When multiple services are collocated in a single server, the transfers are batched and sent over a single shared TCP stream.

Newly spawned services and services that rebooted after a crash will consequently "catch up" gracefully with the rest of the service replicas by means of the epidemic protocols.

4 Experimental Evaluation

Tempest was implemented in Java, enhancing the Apache Axis Soap [28] web services stack with a new transport protocol that uses a multicast primitive, i.e. SOAP over TempestTransport instead of SOAP over HTTP. The

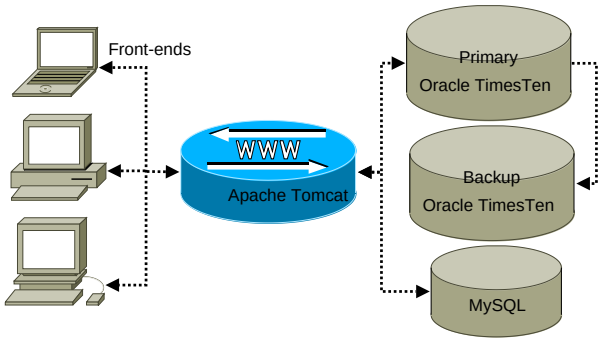

Figure 3. Baseline configurations.

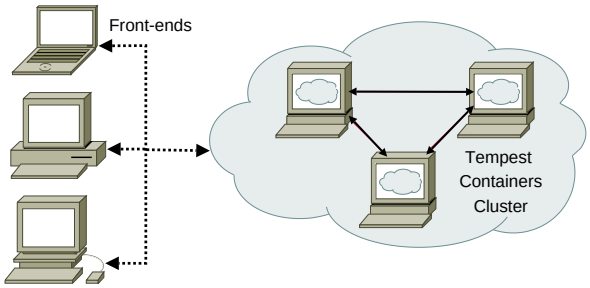

Figure 4. Tempest configuration.

deep cloning capability was implemented using the Java Reflection API. The system components are built with Java's non-blocking I/O primitives using a high performance event driven model similar to the SEDA [29] architecture.

The evaluation is structured as follows: in subsection 4.1 we show that a single replicated Tempest service can provide rapid response to large numbers of concurrent front-end requests. In subsection 4.2 we show that this is true even when services are heavily loaded. Finally, in subsection 4.3, we show that the two knobs provided by Tempest — number of replicas per service and number of redundant queries — enable rapid predictable response for "service-clouds" composed of many collaborating services with differing timing characteristics.

4.1 Scalability in the Number of Concurrent Connections – Micro bechmarks

We ran a set of micro benchmarks to compare Tempest against four multi-tier baseline scenarios. In all configurations we had the same set of front-ends interacting with the `ShoppingCart` web service. On one hand we deployed the service on top of the Apache Tomcat server. The service stores the data using a relational database repository as shown in figure 3. We stored the data using the Oracle TimesTen in memory database (configured in "high performance cache-mode" for in-memory operations only) first co-located with the Tomcat server, second on a remote third-tier machine and lastly deployed in a primary-backup

configuration with the primary co-located with the Tomcat container and the backup on the third-tier machine. The primary-backup scheme provided by TimesTen that we used is called *return receipt*, and it ensures that upon submitting a request to the master the application is blocked until the replication scheme confirms that the update has been received by the backup. Since we configured TimesTen to work without committing durably to disk every transaction, the stronger *return twosafe* replication mode was not necessary. We also use an ubiquitous on-disk database engine, and for that purpose we relied on MySQL 5.0 with the InnoDB storage engine configured for ACID compliance — flushing the log after every transaction commit, and the underlying operating system (Linux 2.6.15) with the file system mounted in synchronous mode and with barriers enabled. On the other hand we have deployed the `ShoppingCart` service on 3 replicated Tempest servers gossiping at a rate of once every 100 milliseconds (see figure 4) – we did not replicate Tomcat for load balancing since all Tempest replicas were configured to receive every update. The Tempest `ShoppingCart` service stores the data inside a *TempestMap*.

The workload consists of multiple clients issuing 1024 byte requests at a rate of 100 requests per second against the `ShoppingCart` service. Requests are issued in a closed loop [25]. Every experiment had a startup phase in which we populated the data repository with 1024 distinct objects. Client requests were drawn from a Zipf distribution (with $s = 1$) over the space of object identifiers – reads and writes equally distributed. We report measurements of the Web Service Interaction Time, i.e. the request latency as observed by 1, 2, 4, 16, 32, 64, 128, 256, 512, 800 and 1024 concurrent clients. Results are averaged over 40000 runs per client.

Figure 5 shows that Tempest latency is significantly less – often by over an order of magnitude – than any of the baselines, thus confirming that fault-tolerant services with time-critical properties can be built on top of the Tempest platform. The graphs also indicate that Tempest scales well with the number of concurrent requests.

As can be seen from the breakdown of the latency, most of the overhead comes from the round trip time and the Tomcat container, which is to be expected since the workload consists of operations on small data records over the database — we hypothesize that database lock contention has not kicked in yet, the Tomcat container being the first one to experience severe overload.

Looking more carefully at the breakdown of the latency in figure 5 (the 1-to-32 concurrent clients spectrum) one can notice that the time spent by a Tempest service manipulating the data (i.e. performing object deep cloning, data structure lock contention, web service invocation identifier tagging and index maintenance) is small compared to the database interaction — as a matter of fact it grows remains around 1 millisecond no matter what the number of concurrent clients is — showing that fine grained data structures allow for better performance under contention.

4.2 Graceful Recovery under Heavy Load

Next, we ran a set of experiments to report on Tempest's behavior in the face of failures. Node crashes turned out not to be especially interesting since Tempest's gossip failure detection protocols quickly detect that the node has failed, expel it from the group and shift work to other nodes. More details on the timeliness of a variant of the gossip based failure detector we used can be found in our previous work [19]. We did however identify a class of overload scenarios that have a more visible impact on the Tempest replicated services. These scenarios degrade some service components without crashing them. The services become lossy and inconsistent, and queries return results based on stale data. Two questions are of interest here: behavior during the overload, and the time required to recover after it ends.

We replicated the `ShoppingCart` service on 6 Tempest servers running on the Cornell cluster – each machine is a 1.33Ghz Intel single CPU blade-server with 512MB of RAM. We inject a single source stream of updates at a particular rate of one update every 20 milliseconds. The same client perform query requests on 8 concurrent threads at the same time. The query stream is at a higher rate than the updates (in this case 4 times higher). Client requests were drawn from a Zipf distribution (with $s = 1$) over the space of object identifiers – reads and writes equally distributed.

The overload unfolds in the following way:

- At time t from the start of the experiment 128 "rogue" clients bombard with requests 3 of the Tempest servers. Call the Tempest services *victims*.
- At time $t + \Delta$ the rogue clients terminate.

In the experiments that follow, t is 10, and Δ is 30 seconds.

The rogue clients bombard the victims with multiple streams of continuous IP multicast requests in the attempt to saturate their processing capacity. However, we found that this was not enough to perturb the normal behavior of the servers, hence at the same time we superimposed additional background load on the victim servers. These attacks do not actually cause the servers to crash, but they do cause them to become overloaded in processing incoming updates and hence return stale results.

Server overloads will not influence the performance of Tempest at non-attacked services, hence *we report only on the impact of the disruption at the affected replicas*. Figure 6 shows the number of "stale" query results on the y-

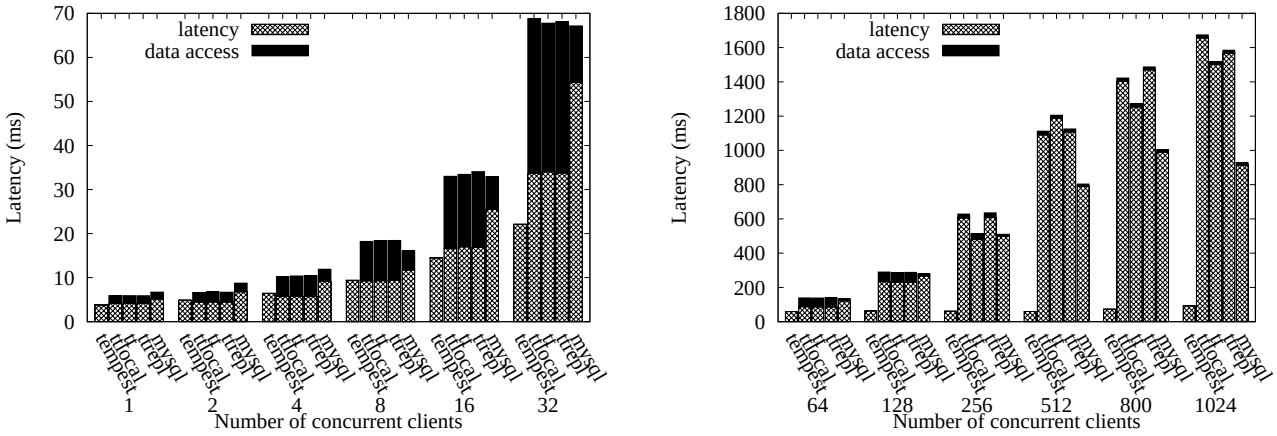

Figure 5. Request latency. Each group of bars represent tempest (*tempest*), times ten on the local machine with tomcat (*ttlocal*), times ten on a remote machine (*tt*), times ten in primary-backup mode with the primary on the same machine as tomcat and the backup on a remote machine (*ttrepl*), and mysql on a remote machine (*mysql*).

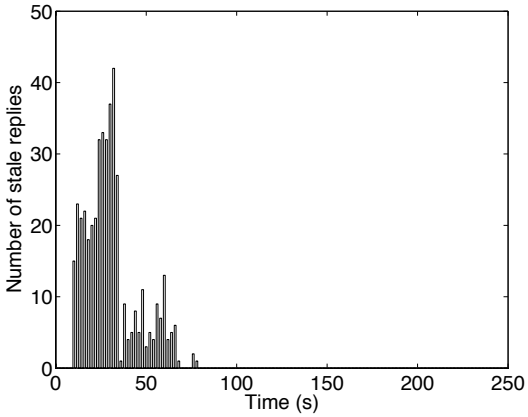

Figure 6. Number of stale results.

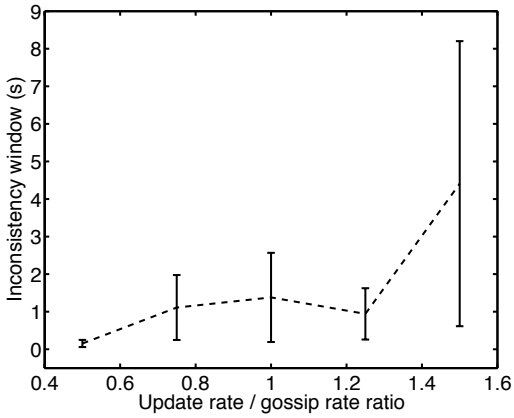

Figure 7. Inconsistency window.

axis against the time in seconds on the x-axis, binned in 2-second intervals. The client issues an update every 20 milliseconds and the Tempest gossip rate is set at once every 40 milliseconds. Throughout this period, the victim nodes are overloaded and drop packets, while the Tempest repair protocols labor to repair the resulting inconsistencies. Meanwhile, queries that manage to reach the overloaded nodes could glimpse stale data (not reflecting recent issued updates since the updates were lost). Once the attack ends, Tempest is able to gracefully recover.

The ratio of the gossip rate to the update rate will determine the robustness of Tempest to this sort of overload scenario. To quantify this effect, Figure 7 shows the inconsistency window as perceived by clients during the disruption. This is the period of time during which clients of a service see more than one stale query result within a 2-second interval. The inconsistency window is plotted against the ratio between the update rate and the Tempest gossip rate, with the update rate fixed at 1/20 milliseconds. The window is minimized when the gossip rate is at least as fast as the update rate.

4.3 Scalability in the Number of Services

To estimate how Tempest scales in different dimensions — in particular size of the collaborating services, number of front-ends and number of replicas — we built a synthetic PetStore on top of Tempest and evaluated it on the Emulab testbed. The application consists of a battery of front ends issuing requests to a "cloud" of services.

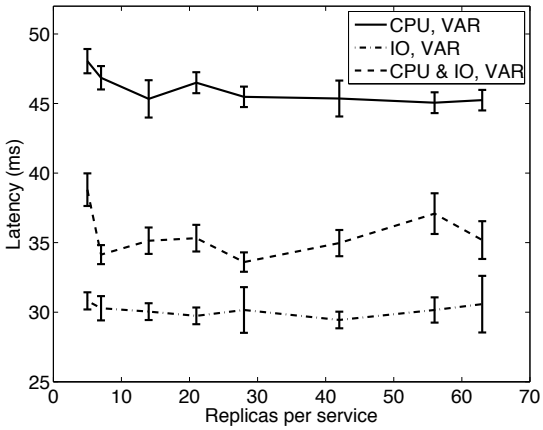

Figure 8. Large variance service latency.

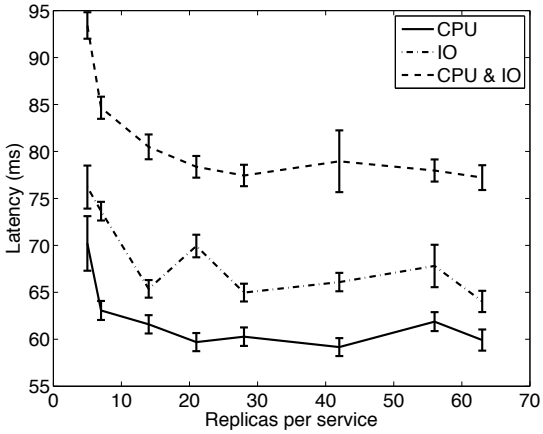

Figure 9. Small variance service latency.

The services have different response time characteristics: some are IO intensive – for example an indexing service may access disk much more often than the average service, others are CPU intensive – for example a recommendation service may require considerably more CPU cycles than the average service, while other services are both IO and CPU bound. We also consider the response time variances for these types of services, in particular the PetStore services have both small and large response time variance. We observed that services performing multiple IO operations are likely to suffer from scheduling delays. Lock contention within Tempest may be another cause for large response time variance.

We ran a set of baseline experiments to measure the behavior of each type of service individually, under normal load. The experiment consisted of two front ends issuing request streams (half updates half reads) of one query every 40 milliseconds in closed loop to a single replicated service. Services have the gossip rate set for once every 100 mil-

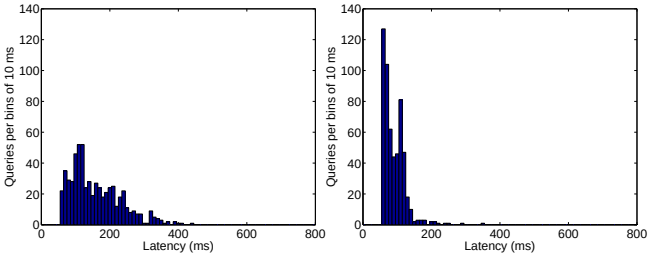

Figure 10. Pet-store response time histograms, left: no replicas, right: 8 replicas.

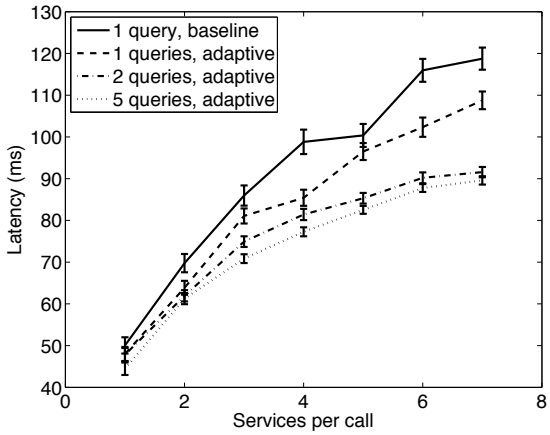

Figure 11. Pet-store latency, 5 replicas each.

liseconds. We repeated the experiment for various number of replicas and for each of the types of services mentioned above. Figure 8 shows the query latency for services with large response time variance, and small response time variance respectively (figure 9). The error bars represent standard error. Note that even for services that we instrumented to have small response time variance, if they are IO bound they do exhibit large variance — in particular note the CPU & IO bound service for 42 replicas and the IO bound service for 56 replicas. We should note that for this client request load, the service instances become overloaded if we drop below about 3 replicas, and we don't report those values (response times are meaningless when the service isn't able to keep up with the request rate).

Next we evaluated the PetStore as a "cloud" of seven services — the six with the characteristics presented in the previous experiments, along with another baseline service that shows the overhead caused by Tempest. Four front-ends perform multi-service requests (half queries half updates) against the PetStore in a closed loop, each at a rate of once every 50 milliseconds — we chose the rate so as to not completely overload the platform and observe queueing effects instead.

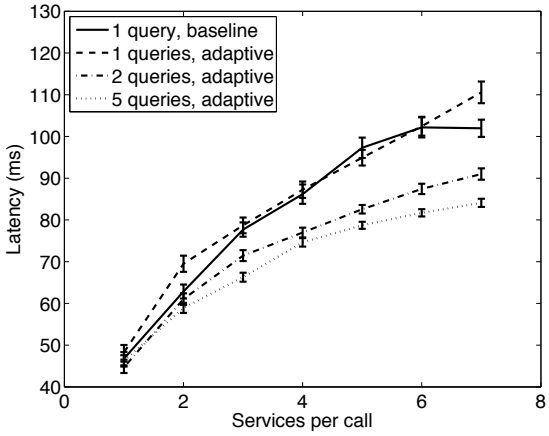

Figure 12. Pet-store latency, 8 replicas each.

Figure 10 show response time distributions for multi-service requests sent to all services — every request issued by a front-end is sent in parallel to each service, the front end returns when replies from every service is received. Requests have the redundant querying parameter $k = 2$. Each histogram shows the number of requests per bins 10 milliseconds wide. We show the scenarios: the one in which neither of the services is replicated, and the one in which services have 8 replicas each. The graphs show that replication provides more opportunities for queries to be absorbed by load balancing and that redundant querying pays off.

Figures 11 and 12 show response times for multi-service requests (with standard error denoting the error bars). Every multi-request issued by a front-end chooses at random n distinct services, where n is the number of services per query, presented on the x-axis. We used the adaptive query algorithm with the k parameter set to 1, 2 and 5. For baseline we used a simple query discovery algorithm by which the first query for a service is multicast, and all subsequent queries are sent to the one replica that replied the fastest to the multicast. In figure 11 every service is replicated 5 times, while in figure 12 every service is replicated 8 times. First we conclude that redundant querying does indeed improve performance, with the largest payoff for $k = 2$. Second, the adaptive querying algorithm pays off mostly in scenarios where the number of replicas is small.

5 Related Work

Soft state mechanisms have been used extensively in network protocols [32, 12], as well as in large cluster-based services like Porcupine [24] and others [14, 6, 26]. Proposals exist for extending the standard web-service model to include soft state — a prominent example is the Grid Computing standard [13]. Recovery-oriented computing [7] is an alternative approach to providing fast failover and availability in the face of failures — however, it does not replace replication as a mechanism for balancing heavy load across multiple machines. Distributed data structures have been proposed before [16] as building blocks for clustered services. The work in [33] is very similar in spirit to Tempest, but examines the orthogonal question of providing customizable durability levels through a single storage abstraction; one of these levels is meant for soft state that needs to be replicated for high availability. SSM [18] is a system for managing and storaging a particular category of soft state — user session information.

Clustered application servers like BEA WebLogic Application Server [3] and IBM WebSphere [17] allow storage of state in special containers that are typically stored within persistent databases. There has been a large amount of work in the field of fault-tolerant middleware, especially around CORBA [2, 21, 11], but most of this work does not consider interaction with a database third tier. DBFarm [23] is an architecture for scaling a core of multiple databases through the use of less reliable replicas.

6 Conclusion

Modern three-tier architectures achieve scalability and responsiveness through the extensive use of soft state techniques in the service tier. Availability and rapid fail-over requires data replication, and Tempest provides programmers with data structure abstractions for storing and managing replicated soft state. Tempest scales well in key dimensions — the number of front-ends contacting a service and the number of services contacted by a front-end — and outperforms in-memory databases in realistic settings. As a result, Tempest simplifies the construction of highly responsive systems that seamlessly mask load fluctuations and faults from end-users.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. C. Veitch, and C. T. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, pages 159–174, 2007.

[2] R. Baldoni and C. Marchetti. Three-tier replication for FT-CORBA infrastructures. *Software Practice and Experience, 2003*, 6 2003.

[3] BEA Systems, Inc. Clustering the BEA WebLogic Application Server, 2003. http://e-docs.bea.com/wls/docs81/cluster/overview.html.

[4] J. Byers, J. Considine, and M. Mitzenmacher. Fast approximate reconciliation of set differences. Boston University Computer Science Technical Report 2002-019., 2002.

[5] L. Camargos, F. Pedone, and M. Wieloch. Sprint: a middleware for high-performance transaction processing. In *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys Eu-*

ropean Conference on Computer Systems 2007, pages 385–398, New York, NY, USA, 2007. ACM.

[6] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: a soft-state system case study. *Perform. Eval.*, 56(1-4):213–248, 2004.

[7] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *OSDI*, pages 31–44, 2004.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM.

[9] 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.

[10] 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.

[11] P. Felber, X. Défago, P. Eugster, and A. Schiper. Replicating CORBA objects: a marriage between active and passive replication. In *Second IFIP International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*, pages 375–387, Helsinki, Finland, 1999.

[12] S. Floyd, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking (TON)*, 5(6):784–803, 1997.

[13] I. Foster, K. Czajkowski, D. Ferguson, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. Modeling and Managing State in Distributed Systems: The Role of OGSI and WSRF. *Proceedings of the IEEE*, 93(3):604–612, March 2005.

[14] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier. Cluster-based scalable network services. *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 78–91, 1997.

[15] J. N. Gray. A Conversation with Werner Vogels: Learning from the Amazon technology platform. *ACM Queue*, 4(4), May 2006.

[16] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. E. Culler. Scalable, distributed data structures for internet service construction. In *OSDI*, pages 319–332, 2000.

[17] IBM. WebSphere Information Integrator Q replication, 2005. http://www-128.ibm.com/developerworks/db2/library/techarticle/dm-0503aschoff/.

[18] B. C. Ling, E. Kiciman, and A. Fox. Session state: beyond soft state. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[19] T. Marian, K. Birman, and R. van Renesse. A Scalable Services Architecture. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS 2006)*. IEEE Computer Society, 2006.

[20] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49(9):2213–2218, 2003.

[21] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Lessons Learned in Building a Fault-Tolerant CORBA System. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 39–44, Washington, DC, USA, 2002. IEEE Computer Society.

[22] M. Pezzini. The Evolution of Transaction Processing in Light of .NET and J2EE. *Business Integration Journal Online*, November 2005.

[23] C. Plattner, G. Alonso, and M. T. Özsu. Dbfarm: A scalable cluster for multiple databases. In *Middleware*, pages 180–200, 2006.

[24] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 1–15, New York, NY, USA, 1999. ACM Press.

[25] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open vs closed: a cautionary tale. In *Proceedings of the 3rd Symposium on Networked System Design and Implementation (NSDI)*. Networked System Design and Implementation (NSDI), 2006.

[26] K. Shen, T. Yang, L. Chu, J. L. Holliday, D. A. Kuschner, and H. Zhu. Neptune: scalable replication management and programming support for cluster-based network services. In *USITS'01: Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*, pages 17–17, Berkeley, CA, USA, 2001. USENIX Association.

[27] Sun Microsystems. The Collections Framework, 1995. http://java.sun.com/docs/books/tutorial/collections/index.html.

[28] The Apache Software Foundation. Apache Axis, 2006. http://ws.apache.org/axis/.

[29] 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.

[30] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*.

[31] World Wide Web Consortium. Web Services Architecture, 2002. http://www.w3.org.

[32] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: a new resource reservation protocol. *Communications Magazine, IEEE*, 40(5):116–127, 2002.

[33] X. Zhang, M. A. Hiltunen, K. Marzullo, and R. D. Schlichting. Customizable service state durability for service oriented architectures. *Sixth European Dependable Computing Conference*, 0:119–128, 2006.