Tempest: Scalable Time-Critical Web Services Platform[*]

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

We describe Tempest, a platform for assisting web services developers. Tempest allows Java programmers to create services that scale across clusters of computing nodes, adapting automatically as load surges or drops, components fail or recover, and client-generated loads vary. The system automates tasks such as replica placement, update dissemination, consistency checking, and repair when inconsistencies occur. We run Tempest over Ricochet, a probabilistically reliable multicast with exceptionally good timing properties. The resulting scalable and self-adaptive web services provide QoS guarantees such as rapid response to queries and rapid update when events relevant to server state occur.

1 Introduction

Service Oriented Architectures (SOAs), including Web Services, J2EE and CORBA, enable developers to interconnect diverse components into large service oriented systems. In the most common SOA configuration, applications a—re structured using a 3-tier model consisting of clients (the first tier), services (second tier), and backend database systems (the third tier). A service fetches and structures data stored in backend databases and then offers the results to web-based front-ends through remote method invocation interfaces. The model is hugely successful, and many perceive it as a one-size-fits-all solution for commercial datacenter deployments.

However, for an important class of *time-critical* applications the 3-tier model is problematic. In these applications costs introduced by the back-end transactional database pose a problem. When rapid response is *always* desired, the overhead and potential delays associated with transactional concurrency control, rollbacks and multi-phase commit protocols that might block as a result of failures are serious issues. These costs can be particularly annoying in applications that don't even need the strong guarantees of the transactional model.

Responsiveness concerns are fueling a rapidly growing market for in-memory database applications and other architectures that eliminate the third tier of the traditional model, permitting client platforms to interact directly with services that maintain all the data needed to compute a response locally. However, platform support for applications constructed in this way has lagged.

Our project, called Tempest, seeks to bridge this gap by offering an easily used tool with which developers familiar with a traditional web services environment can quickly and reliably create 2-tier service systems that automate many of the tasks associated with scalability while also achieving extremely rapid response times, measured both in terms of the latency associated with performing queries and the latency to update the data used by the service. Tempest achieves these objectives even in the presence of bursts of packet loss, node crashes, and adaptations such as launching new replicas that must join the system while it is running. A tradeoff arises between protocol overhead and the latency before which updates are applied; developers can tune these costs to reflect application-specific preferences.

The kinds of applications we've studied in developing our solution, and for which we believe it would be applicable, include financial services that use in-memory databases to store information about client portfolios and market conditions, allowing analysts to make trading decisions rapidly based on "real-time dashboards". Air traffic control systems need to provide controllers with instant information about aircraft tracks, weather updates, and other events. In e-commerce datacenters, rapid response is often the key to making a sale.

Tempest does not replace 3-tier database solutions. We assume that 2-tier applications will often co-exist with other applications that need the stronger properties of a traditional 3-tier platform. For example, an e-tailer might use

[*]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.

a rapidly responsive 2-tier solution to build the web pages its customers interact with, but a more traditional 3-tier solution when a customer makes an actual purchase. An air traffic control application might use a 2-tier technology for mundane interactions with the controller, but revert to the stronger guarantees of a traditional 3-tier database when actually updating the flight plan for an inbound aircraft.

Moreover, even in-memory database systems don't always eliminate the backend database. Developers of these kinds of systems often use in-memory databases as fast-response caches for improving the performance of traditional on-disk databases, or as replacements for them. The cache configuration retains most of the durability guarantees of a traditional database while improving performance by offloading queries from the backend database, freeing the backend system to spend a higher percentage of its resources handling updates, while the number of cache-based front-end systems can be increased as desired to handle high query loads.

Tempest is built around a novel storage abstraction called the TempestCollection in which application developers store the state of a service. Our platform handles the replication of this state across clones of the service, persistence, and failure handling. To minimize the need for specialized knowledge on the part of the application developer, the TempestCollection employs interfaces almost identical to those used by the Java Collections standard. Elements can be accessed on an individual basis, but it is also possible to access the full set by iterating over it, just as in a standard Collection. The hope is that we can free developers from the complexities of scalability and fault-tolerance, leaving them to focus on application functionality.

The TempestCollection offers our platform a means of accessing the application state. Using this, we've designed a suite of protocols and algorithms that handle replication in ways that try to guarantee rapid responsiveness. The platform automates placement of web service replicas onto disjoint nodes, adapting the configuration as demand changes over time. Finally, it provides partitioning functionality for services in which requests have a suitable key, and provides support for request balancing, automatic restart and recovery after failures.

Under the hood, Tempest uses a reliable multicast protocol called Ricochet to disseminate updates. Ricochet was designed explicitly for time-critical applications running over commodity clusters and guarantees extremely high reliability and low delay. The protocol gains speed by offering probabilistically reliable delivery, and there are patterns of correlated failure that could cause packets to be lost. Accordingly, Tempest includes epidemic protocols that continuously monitor service replicas, checking for inconsistencies by comparing the contents of the TempestCollection objects and repairing any persistent problems. One of the goals of the present paper is to experimentally quantify the quality of service that can be achieved in this manner, as a function of the overhead associated with the various protocols employed within the system.

2 Assumptions and Execution Model

A *tempest service* is a web service developed using our platform. Such a service complies with all aspects of the web services standards and in fact could be built using web service builder tools. In particular, it exposes a client interface against which applications issue *requests*. We differentiate between query requests, which leave the service state unchanged, and updates.

We intercept operations on the client side by exploiting a standard platform feature. The key here is that tempest services are *named* in a way that forces the web services platform to communicate over the Ricochet protocols. When a client binds to a service, this naming feature causes the Ricochet protocols to be loaded (if they are not available, the client would get an error). Thus, when the client system invokes a request, our protocol stub is able to perform load-balancing tasks for queries, while mapping updates into Ricochet multicasts.

Tempest does not attempt to strengthen the usual web services guarantees. The default is rather weak: both requests and replies can be lost, and the application simply re-issues timed-out requests. However, the developer of a web service can optionally implement stronger guarantees, using the WS-Reliability standard; if a service incorporates the associated mechanisms, the Tempest-generated replicated version will preserve the desired behavior.

Similarly, the web services model assumes that applications are correct, that they fail by crashing, and that failures can be detected using timeout. Tempest works within the same assumptions, although as seen below, the protocols we use to detect and repair inconsistencies between replicas might be able to catch and compensate for some forms of buggy runtime behavior.

2.1 Tempest Service Interface

Each service exposes a set of methods that are callable by clients. For example a stock trader service interface that we will use throughout the paper is listed in Figure 2. The interface was taken from the examples provided in the BEA WebLogic Server and WebLogic Express Application Examples and Tutorials [4].

Buy and sell do the obvious things; these are classified as update operations because they change the state of the account. Check is a read operation; it retrieves the current account status (number of shares) for the symbol. The role of the "optimistic" flag will be discussed later.

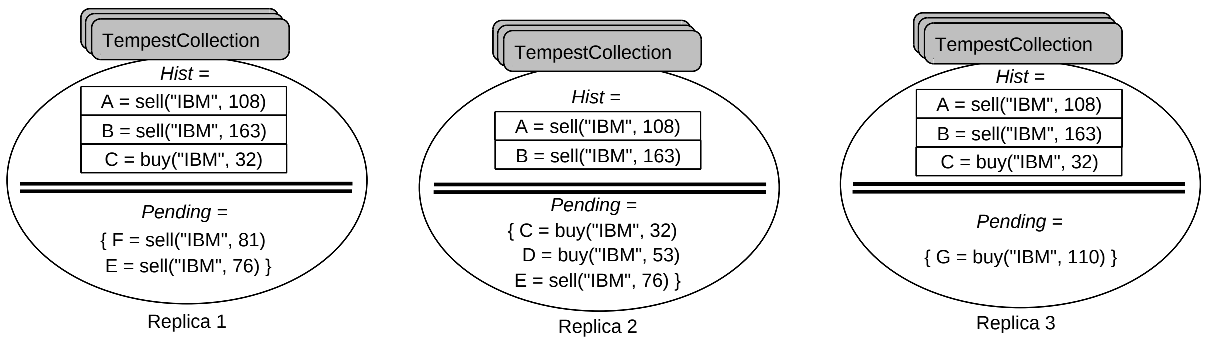

Figure 1: Tempest Trader service state at 3 replicas for *obj* – "IBM" stock trades.

```
public interface TraderIF {
    update int buy(String stockSymbol, int shares);
    update int sell(String stockSymbol, int shares);
    read int check(String stockSymbol, boolean optimistic);
}
```

Figure 2: Stock trader web service interface.

Traditionally, services relying on a transactional database backend offer a strong data consistency model in which every read operation returns the result of the latest update that occurred on a data item. With Tempest we take a different approach by relaxing the model such that services offer sequential consistency [10]: Every replica of the service sees the operations on the same data item in the same order, but the order may be different from the order in which the operations were issued. Later, we will see that this is a non-trivial design decision; Tempest services can sometimes return results that would be erroneous were we using a more standard transactional execution model. For applications where these semantics are adequate, sequential consistency buys us scheduling flexibility that enables much better real-time responsiveness.

We model the persistent state of a service as a collection of objects. Due to our choice for consistency model each object is naturally represented by the tuple $\langle Hist_{obj}, Pending_{obj}\rangle$. $Hist_{obj}$ is the state of object *obj* (either the current value, or the list of updates that have been applied to it), while $Pending_{obj}$ is the set of *pending updates* that cannot be applied yet. Tempest delays the application (*merge*) of updates until it can confirm that the update set and the ordering is consistent across replicas. Each replica maintains a list of updates in the order Tempest currently expects to use them, but this order is sometimes revised as protocols are executed.

Denote $Hist^i_{obj}$ and $Pending^i_{obj}$ as the corresponding history and pending operations stored at replica i. We define $Hist^i_{obj} \preceq Hist^j_{obj}$ to hold if the history of updates at i is a prefix of the updates at replica labeled j, i.e. the same object *obj* at replica i is a past version of the object at replica j. Figure 1 shows a possible configuration for the stock trader service at three distinct replicas. The figure shows a projection of only the objects corresponding to the "IBM" stock trades stored in one of the service's Tempest collections. As shown in the figure $Hist^1_{obj} = [A, B, C]$; $Hist^2_{obj} = [A, B] : Hist^2_{obj} \preceq Hist^1_{obj}$.

The global history for an object is defined as the maximal history held at any replica: $\forall i,\ Hist_{obj} = max_{\preceq}\ Hist^i_{obj}$. For the configuration in Figure 1 $Hist_{obj} = [A, B, C]$. Tempest also ensures that for all i, $Hist^i_{obj} \preceq Hist_{obj}$.

The global set of pending operations for an object is defined as: $Pending_{obj} = \bigcup_i P^i_{obj} \setminus Hist_{obj}$. Considering the configuration from Figure 1 we have $Pending_{obj} = \{F, E\} \cup \{C, D, E\} \cup \{G\} \setminus \{A, B, C\} = \{D, E, F, G\}$.

The *merge* operation takes a subset of the pending updates, orders them and appends them to the history and then removes them from the pending set. A merge is invoked by the Tempest platform as soon as it can safely do so – which is to say, that it has determined that some prefix of the update set is complete and correctly ordered. For example in Figure 1, Replica 2 is instructed by Tempest protocols to apply the pending update $\{C\} \subset Pending^2_{obj}$. As a result, $Hist^2_{obj} = \{A, B, C\}$ and $Pending^2_{obj} = \{D, E\}$.

Because the Tempest protocol takes time and, during this time, pending updates are known but have not yet been applied to the persistent object state, the Tempest developer faces a choice, with non-trivial performance implications.

At the time a read operation is performed, the platform can enforce sequential consistency by applying queries against the local history at a server. We call this *pessimistic* application of the query, because the state is stable - but it could be stale. In our experimental section we quantify the delay before pending updates are applied and show that this is mostly a function of the rate at which sequencer updates are generated. Thus one option is to opt for pessimistic queries but to adjust the sequencer rate to match application needs.

Alternatively, since the pending operations are also available, a query operation can be satisfied *optimistically* by performing a tentative merge at the local server and responding to the client's request with the resulting provisional but "unconfirmed" object state. Here, we can run the sequencer less aggressively, but we incur a different set of costs: Tempest will need to compute the provisional state; and also runs the risk that pending updates might be applied out of order, or that updates may be missing. Thus, an optimistic result will be more complete in one sense (it includes all pending updates) but could be incomplete in other respects.

Consider for example the configuration in Figure 1. A pessimistic query at Replica 1 would return a result computed based on $[A, B, C]$ while at Replica 2 on $[A, B]$. An optimistic query at Replica 1 would return a result computed based on $merge([A, B, C], \{F, E\})$; at Replica 2 it would return a result computed given $merge([A, B], \{C, D, E\})$.

We considered adopting a similar approach for updates, but concluded that developers would find this confusing. The issue is that Tempest updates execute asynchronously, hence it is more appropriate to return some form of operation id (Tempest assigns each request a unique id), or perhaps an exception code. Applications needing to query the new object state would then implement a (pessimistic) read operation that waits for the specified update to be executed before returning the result of the operation.

2.2 Tempest Containers

A Tempest service resides in a *container*. A single container represents the platform configuration on a single computer and might run several web services. Each service is replicated across a number of containers for purposes of load-balancing and high availability.

Tempest includes a GMS (Group Management Service) component that we use to coordinate the configurations of the Tempest containers. The GMS reads a configuration file that provides the broad parameters for a given environment: the set of nodes, the set of services, the desired replication levels, etc. Containers are started on each node when it boots, and connect to the GMS for instructions. The GMS orchestrates initial setup and sends updates as conditions evolve.

Earlier, we cited the need to intercept operations so as to redirect them through the Tempest protocol stack. To this end we have extended the Apache Axis [14] framework to support our new Ricochet-based transport protocol [2], which runs within the Apache SOAP engine stack. Tempest services are named using URIs (Uniform Resource Identifiers) of the form: `ricochet://gms.cs.cornell.edu/StockTrader` with *ricochet* denoting the transport protocol, the host component pointing to the address of the GMS and the Ricochet group name (StockTrader) as the path.

A client binding will fail if the client system doesn't have the Ricochet protocol installed as one of its available transport protocols. For a client that does have Ricochet installed, the Ricochet module interacts at bind time with the GMS to obtain the appropriate internal mapping between the group names and the groups themselves, minimizing delay when a service invocation is later performed.

2.2.1 TempestCollections

The core of the Tempest platform is the *TempestCollection* class. Application state is represented within these classes as a set of member objects ("items"). Our approach was to make the TempestCollection framework as similar as possible to the widely used Java Collections Framework [13]; items are modeled on Java Beans. TempestCollections and Java Collections differ in the following respects:

- Objects stored in a TempestCollection are automatically replicated across the service replicas. In particular, Tempest may sometimes update the state in response to its own protocol messages, and hence the application is not the only source of updates.

- Tempest collections come in pairs. The first holds the stable state of an object $Hist_{obj}$ while the second holds pending operations $Pending_{obj}$, for which ordering is not yet stable. The object state is of type `HItem`; this could be the current value of the object, or (in the limit) a complete history of the updates applied to it in the order they were performed. The pending updates are of type `PItem`.

- Objects stored in a TempestCollection cannot be modified directly; they can only be changed by appending an update request to the pending operation list and waiting for the platform to apply a merge operation.

In the case of update operations, Tempest employs an interlock to ensure that if a single request results in multiple updates to the TempestCollection classes, the platform sees them all at once.

For many applications (including our trading service), the Tempest platform includes all needed functionality. Development of this sort of service is almost completely mechanical and, indeed, could probably be fully automated. However, some applications depend on elaborate data structures that go beyond the simple list that can be stored in a collection. To support them, Tempest allows the application developer to maintain pointers into the collection class, and hence to implement any data structure that the developer finds convenient. The application updates these "external" data structures when a merge operation is initiated by Tempest.

When using external data structures, the application developer must be aware of one additional issue. In some cases, such as when a container is rebooted after a crash, Tempest may perform a wholesale update to the contents of the TempestCollection, by transferring information from one replica to another. When this kind of state transfer occurs, Tempest signals that the developer should rebuild any secondary data structures, for example by discarding the old versions and then iterating over the (new) contents of the TempestCollections on an item by item basis.

Helper structures are in some ways incompatible with optimistic request execution. To support optimism, Tempest provisionally merges the pending updates into the history, creating a temporary version of the $Hist$. If we were to also update the external structures, we might need to roll back to the previous version if the optimistic update ordering later turns out to have been wrong; a deep clone of this sort could be expensive. Accordingly, our belief is that external helper structures will be used primarily in conjunction with pessimistically executed operations.

2.3 Example: Developing a simple Tempest service

Although not entirely transparent, we believe that Tempest is a remarkably easy framework to use, and one that closely parallels the prevailing style of application development in Java. To illustrate this point, consider developing a stock trader service that implements the interface presented previously. We'll refer to the service that implements the `TraderIF` interface the `Trader` service.

The application developer starts with a normal web service, and might even debug the service before porting it to Tempest. In our example, porting would require just a few changes. First, the original interface to the `check` method must be extended with a boolean parameter indicating if the query is optimistic or not. Next, the service is changed to store information in the Tempest collection, and to break update operations into an asynchronous stage that creates a new pending update item, and a separate operation that applies a pending update to the persistent state.

Whereas a normal `Trader` service implementation stores its state in any form the builder finds convenient, the Tempest version stores its state using a TempestCollection. For this kind of very simple service, the developer would simply extend the `HItem` and `PItem` types to tailor these to the needs of their service. Developers are required to provide an implementation for the abstract method `HItem applyDelta(HItem obj)` that belongs to `PItem` objects. Given a pending operation, `HItem applyDelta(HItem obj)` applies the update to the current history. Tempest has a built-in merge method that uses `applyDelta` automatically to handle optimistic and pessimistic queries and to create the optimistic state used when replying to an update.

Both `buy` and `sell` methods work by adding `PItem` objects into the collection reflecting how many shares of a particular type have been bought or sold. At the time of insertion these are pending update operations; Tempest will ultimately decide the order in which they are applied.

If our service maintained some form of external helper data structure, `applyDelta` method would update the external structure at the same time that it updates the persistent state of the service. For example, suppose that the `Trader` service maintains a b-tree index into the TempestCollection $Hist$. Each merge would update the b-tree at the same time as it updates the collection. The service would also implement an upcall `collectionModified`, which the platform would invoke in the event of a state transfer that changes the collection contents other than through a series of calls to `applyDelta`. In this (rare) case, the current b-tree would be discarded and a new version computed on the basis of the new $Hist$.

To summarize, developers building web services using the Tempest framework must implement:

- The web service itself – this would often occur offline using standard tools, after which the service can be ported to Tempest.
- The abstract method by which a `PItem` can alter the state of a `HItem`.
- The `collectionModified` upcall – if external data structures must be updated as a result of a repair action.

Having constructed a Tempest service, the developer compiles and links it, then registers it with the Tempest GMS. Tempest replicates and deploys it across multiple containers, and at the same time creates a GMS binding between the service name and a Ricochet group. The Ricochet group is subsequently joined by all the operational containers at which the `Trader` service was deployed.

3 Tempest implementation

Tempest was implemented in Java, using the Apache Axis Soap [14] web services stack and the Ricochet [2] reliable multicast protocol. The system components are built with Java's non-blocking I/O primitives using a high performance event driven model similar to the SEDA [16] architecture. Including Ricochet, Tempest comprises roughly 29000 lines of code.

3.1 Platform architecture

Tempest has three components: the GMS (Group Management System), the Tempest web service containers and the clients that send requests to the web services. The GMS has multiple roles. First it acts as a UDDI (Universal Description Discovery and Integration) registry providing appropriate WSDL (Web Services Description Language) descriptions for the web services deployed on Tempest containers. Second it acts as a group manager for both the Ricochet and the gossip protocols. The GMS also fills the administrator role for Tempest containers, monitoring the overall stress and spawning new containers to match the load imposed on the system. Finally, it monitors components to detect failures and adapt the configuration.

As mentioned earlier, Tempest assumes that processes fail by crashing and can be detected as faulty by timeout. However our model also admits the possibility of transient failures — a process could become temporarily unavailable but later restart and recover any missing updates (for example, a node might become overloaded and the service could slow to a crawl, causing it to look as if it had crashed, but later shed load and recover). Accordingly, Tempest processes monitor the peers with which they interact using a gossip-based heartbeat mechanism. Processes that are thought to be deceased are reported to the GMS, which waits for k distinct suspicions before actually declaring it *deceased*. It then updates and disseminates group membership information to all interested parties. If the number of replicas for a service is too low, the GMS instructs a tempest container to spawn new replicas; if necessary, it can even start up a new container on a fresh processing node.

3.2 Client Invocations

Tempest intercepts client requests when the Ricochet protocol stack is invoked. Every client request is tagged with a web service invocation identifier (wsiid) consisting of a tuple containing the client node identifier and sequence number. Client node identifiers are obtained by applying the SHA1 consistent hash function over the client's IP address and port pair. Each Tempest read or update request is thus uniquely identified by its wsiid.

For read requests, Ricochet interacts with the GMS to fetch the list of active containers running instances of the service, selects a server at random and issues the request to it. Should a timeout occur, retransmissions are sent to some other instance of the service. The GMS notifies the module in the event of a membership change.

For updates, Tempest uses Ricochet to multicast the operation directly to the full set of Tempest containers that hold replicas of the service for which the requests were intended. These post pending updates to their respective $Pending$ sets for eventual execution.

Tempest containers that have replicas of the same service(s) use gossip protocols to reconcile differences between $Pending$ sets. So long as the containers are not permanently partitioned every update will eventually reach every container with probability 1.0 [7]. Given consistent views of the $Pending$ sets across all replicas and a *total ordering relation* on the elements, Tempest can periodically merge the pending updates and ensure that the $Hist$ are also consistent across replicas.

Both read and update operations return a single value. In the case of a read, this will be returned by whichever service instance performed the operation; for an update, a hashing function is employed to select the server instance responsible for replying.

In future work, we are considering extensions of the Tempest interfaces. One near-term extension will allow Tempest to partition a service using some form of key that can be extracted from the request and used to index within a list of sub-services, each of which would independently use the Tempest replication mechanisms. A second set of extensions might allow a client to issue a multi-read that would be processed by two or more service instances, a form of update that waits until the operation has completed and returns the result, a quorum update, etc. None of these would be particularly hard to support, but we want to limit Tempest to mechanisms that can maintain time-critical responsiveness, hence each will need to be evaluated carefully prior to inclusion into the platform.

3.3 Ordering and state commit

Tempest ensures that all the replicas of a web service merge the pending updates in the same order. This is accomplished by using a "BB" (broadcast-broadcast) fixed sequencer scheme [6, 5, 12, 9]. Specifically, for every service, the GMS assigns one of the replicas the role of being the sequencer for the other replicas. The sequencer tracks the local arrival order for the update requests, then periodically uses Ricochet to multicast the ordered list of web service invocation identifiers (either when a threshold number of invocations has been reached or when a timeout expires). If a sequencer fails, the GMS assigns the role to some other replica. We assume a fail-stop model.

3.4 Ricochet

Ricochet [2] is a reliable multicast protocol designed explicitly for clustered time-critical settings. It delivers packets using IP Multicast and recovers from packet loss by having receivers exchange repair packets containing XORs of multicast data. A multicast receiver that loses a packet can recover it from an XOR sent to it by another receiver, provided it has all the other packets contained in the XOR. Most lost packets are recovered by this layer of proactive XOR traffic within a few milliseconds; any packets that cannot be recovered at this stage are retreived using a re-active negative acknowledgment (NAK) layer, either from the sender or some other receiver.

The two-stage packet recovery mechanism results in a bimodal probabilistic distribution of recovery latencies. The percentage of packets recovered within a specific latency bound can be tuned by increasing the XOR repair overhead - more repair packets are generated per data packet, and consequently more lost packets are recovered using the XORs. Ricochet provides scalability in multiple dimensions - the number of receivers in a group, the number of senders in the group, and the number of groups joined by each node. Existing multicast schemes scale very badly in the latter two dimensions, providing packet recovery latencies that degrade as each node splits its incoming bandwidth between different senders and different groups. Ricochet's scalability in the number of groups per node is achieved by exploiting group overlap *regions* — two nodes that both belong to a common subset of groups can perform recovery at the data rate within that subset.

Additionally, Ricochet gracefully handles the dominant failure mode in datacenters - buffer overflows within the kernels of inexpensive end-hosts. By constructing heterogenous repair XORs from data across different groups, Ricochet avoids the susceptibility to correlated packet loss[1] exhibited by conventional forward error correction schemes.

3.5 Epidemic communication

Although Ricochet is a highly reliable protocol, it admits (by design) the possibility that some updates might not be delivered. Thus, replicas of a service can become inconsistent: there are conditions under which some replicas might never receive some updates.

Tempest uses a gossip protocol to repair these kinds of inconsistencies. For example, recall the configuration presented in Figure 1. If the situation shown resulted from some kind of low-probability delivery outcome in Ricochet, Replica 3 will gossip update G to replicas 1 and 2, while simultaneously fetching D, E and F. The gossip protocol is as follows. Periodically, each service replica process computes a digest (summary) of the web service invocation identifiers it has received. It then sends this to a randomly chosen peer. Reliability is not critical for these messages, and we send them using UDP. Upon receiving a digest, a service replica compares the digest with its own state. If it determines that the sender has updates missing from its own pending updates queue, the missing information is pulled from the sender of the digest, which responds with a third packet containing the missing updates.

[1]In work reported elsewhere, we experimented to determine the frequency and causes of message loss in datacenters, and discovered that loss in the communications fabric is extremely uncommon. Nearly all loss occurred within the kernel, and bursty (correlated) lost was common.

During a gossip round, there can never be more than 3 messages issued per process, and these messages are bounded in size (if there are too many updates to fit, Tempest just sends fewer than were requested). Thus, the load imposed on the network is no worse than linear in the number of processes, and any individual process experiences a constant load, independent of the size of the entire system. A piece of information emerging from a single source takes $log(N)$ rounds to reach N processes.

The strength of gossip protocols lies in their simplicity, the fact that they are so 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 formal description of the basic protocols and some of the optimizations can be found in [11].

3.6 Node recovery and checkpointing

Periodically, each Tempest container batches the persistent collections of every service it has deployed and writes them atomically to disk. When a node crashes and reboots, upon starting the Tempest container, the services are brought up to date with the state that was last written to disk before the crash. Checkpointing happens in the same manner, writing down atomically to the stable storage all the $Hist$ collections for every service.

When a container is newly spawned, or when a container that has been unavailable for a period of time missed many updates, Tempest employs a bulk transfer mechanism to bring the container up to date. In such cases, a source container is selected and the contents of the relevant TempestCollections are transmitted over a TCP connection. An upcall then triggers reconstruction of any helper data structures external to the collection. When multiple services are co-located in a single container, the transfers are batched and sent over a single shared TCP stream.

4 Experimental evaluation

We evaluated Tempest in several different scenarios to measure its performance characteristics and behavior under stress. The experiments all use the `Trader` web service deployed and replicated on several Tempest containers.

4.1 Performance

First we compared Tempest against two 3-tier baseline scenarios as shown in Figure 3. In both configurations we had the same set of clients interacting with the `Trader` web

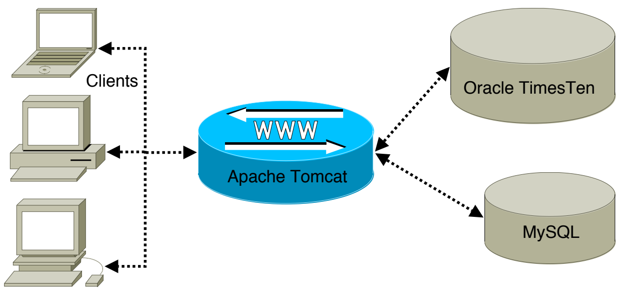

Figure 3: Baseline configurations. Clients perform requests against the same Trader service. The service first uses an Oracle TimesTen in memory database, and later the MySQL engine.

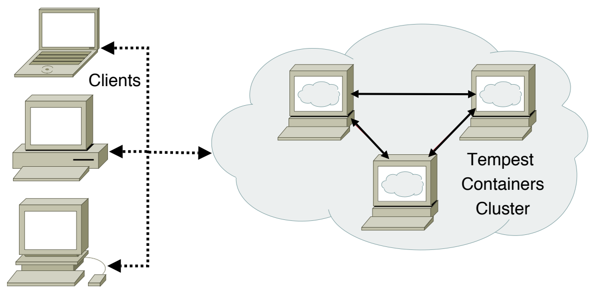

Figure 4: Tempest configuration. Clients multicast requests to a clustered group of processes using Ricochet.

service. We deployed the service on top of the Apache Tomcat container running on Linux 2.6.15-23. The service stores the data using a relational database repository. In the first configuration we use 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 2.6.15-25) with the file system mounted in synchronous mode and with barriers enabled, and with the disk write-back cache disabled. For the second baseline we use the Oracle TimesTen in memory database, configured for best performance. On the other hand we have the `Trader` service deployed on 3 replicated Tempest containers as shown in Figure 4. The Tempest containers gossip at a rate of once every 500 milliseconds, the sequencer works at a rate of once every 20 seconds or every 50 new updates (whichever comes first). The Tempest `Trader` service stores the data inside a TempestCollection, while the baseline configurations store the data in relational tables.

The workload consists of multiple clients issuing 1024 byte requests at various rates against the `Trader` service in each of the three configurations described above. We experimented with request distributions varying from read-intensive to write-intensive (0%, 30%, 50%, 70% and 100% reads, the rest being writes). In the case of Tempest, the reads were drawn from optimistic-intensive, equally likely or pessimistic-intensive distributions. Every experiment had a startup phase in which we populated the data repository with 1024 distinct objects. Client requests were drawn

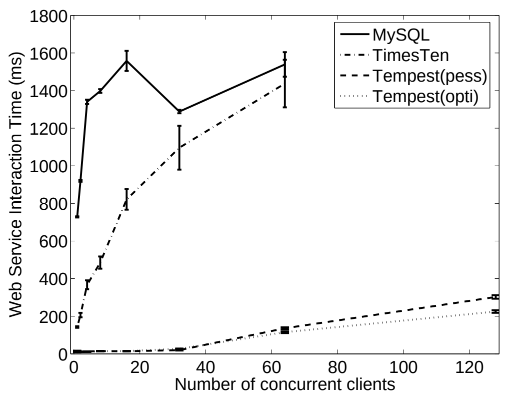

Figure 5: Request latency – client requests are read intensive (70% reads, 30% writes) and drawn from a zipf distribution. For Tempest the reads are either all optimistic, or all pessimistic.

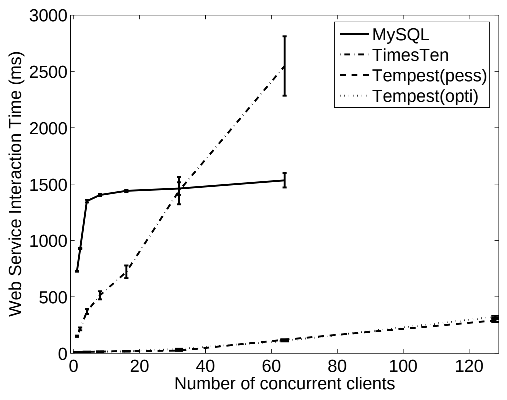

Figure 6: Request latency – client requests are read intensive (70% reads, 30% writes) and drawn from a uniform distribution. For Tempest the reads are either all optimistic, or all pessimistic.

either from a uniform distribution or from a zipf distribution (with $s = 1$) over the space of object identifiers.

We report measurements of the Web Service Interaction Time, i.e. the request latency as observed by 1, 2, 4, 16, 32, 64 and 128 concurrent clients, each client performing 10 requests per second. Neither baseline technology could support 128 concurrent clients (all requests timed out). We kept the default timeout parameter as given by the Apache AXIS web service invocation mechanism. The results are averaged over 10000 runs per client for each distinct number of clients, and include standard error. All the graphs in this subsection have the number of concurrent clients on the x-axis, and web service interaction time on the y-axis.

Figures 5, 6, 7 and 8 show that Tempest latency is at least an order of magnitude less than any of the baselines, thus confirming that fault-tolerant services with time-critical properties can be built on top of the Tempest platform. Also note that the standard error grows at a smaller rate with the number of concurrent clients than in the case of any of the baselines. The graphs also indicate that Tempest

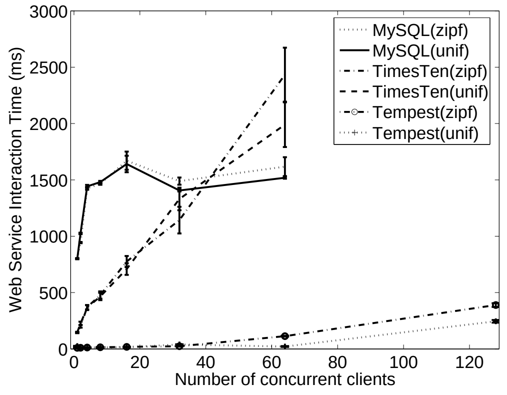

Figure 7: Request latency – request distribution is write-intensive (30% reads, 70% writes), 50% of Tempest queries are optimistic.

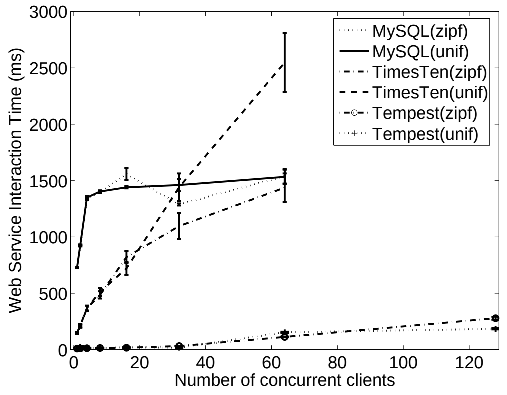

Figure 8: Request latency – request distribution is read-intensive (70% reads, 30% writes), 50% of Tempest queries are optimistic.

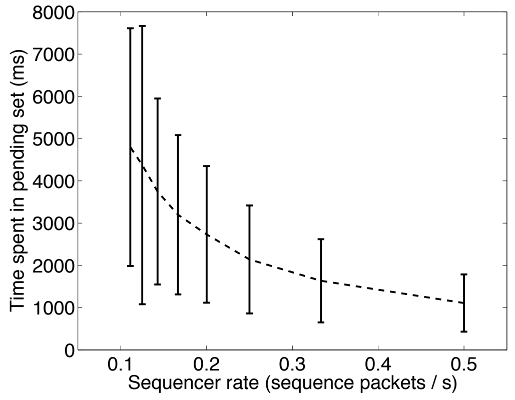

Figure 9: Pending set residency time, update rate 1/200ms. Error bars denote standard deviation.

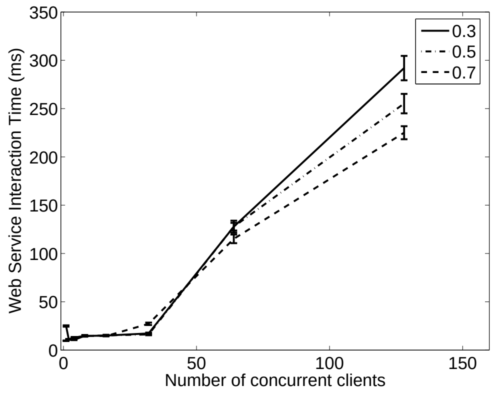

Figure 10: Tempest request latency – requests are write intensive (0.3 means 30% of the requests are reads), equally likely (0.5) and read intensive (0.7). All queries are optimistic.

scales well with the number of concurrent requests.

Figures 5 and 6 show that Tempest is not significantly affected by the request distribution over the objects. In Figure 5 the requests are against objects drawn from a zipf distribution, while in Figure 6 the objects are drawn from a uniform distribution. However the in memory database performs significantly better if the requests come from a zipf distribution. We attribute this to the benefits that can be drawn by exploiting caching opportunities — a mechanism currently lacking in Tempest. MySQL appears to provide less variation in the latency when requests come from a uniform distribution, however it is not clear that it performs better for one of the distributions. Each figure contains two Tempest plots, in the first all the reads are optimistic while in the second all the reads are pessimistic.

As previously, Figures 7 and 8 show Tempest outperforming the two baselines irrespective of the write-intensive (Figure 7), or read-intensive (Figure 8) workloads. However one can observe that Tempest performs better when requests are drawn from a uniform distribution over the set of objects. Hence we believe that Tempest would benefit from a caching infrastructure. Similarly, TimesTen appears to perform significantly better when it can potentially take advantage of caching, namely in the read-intensive, zipf-distributed requests scenario.

Figure 10 shows the web service interaction time of Tempest alone, given write intensive (30% reads), balanced (50% reads) and read intensive (70% reads) loads. The requests are drawn from a zipf distribution over the set of objects. All reads are optimistic; more precisely each query returns values that the `Trader` service computed based on cached optimistic running values of both $Hist$ and $Pending$. Write operations are more expensive than reads due to the fact that reads do not touch the Tempest-Collection, while writes in doing so incur the cost of the deep cloning. Writes also involve synchronization within Tempest, whereas reads can be performed concurrently.

The high cost of optimism evident in Figure 10 may seem surprising to the reader, but it is important to realize that this is partly a consequence of the experimental setup. As just noted, with update rates that rise linearly in the number of clients, lock contention and the length of the pending update queue are bottlenecks here. A developer anticipating

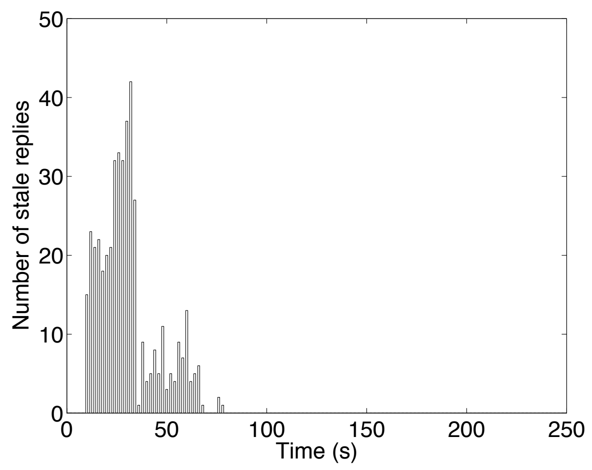

Figure 11: Number of stale results per 2 seconds returned by the affected replicas during a 40 second DDOS disruption. The stale results are grouped into 2-second bins.

a high update rate might increase the sequencer firing rate to keep the length of the pending queue short (Figure 9).

4.2 Denial of Service Attacks

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: Tempest quickly detects that the node has failed and shifts work to other nodes, while Ricochet is unaffected by crash faults. However, we identified a class of distributed denial of service (DDOS) attacks that have a more visible impact on the Tempest replicated services. These attacks 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 attack, and time needed to recover after it ends.

We replicated the `Trader` service on 6 Tempest containers. The GMS and every container run each on a 1.33 Mhz Intel single processor blade-server with 512MB RAM. We inject a single source stream of updates at a particular rate. The updates originate from a single client. The same client performs query requests on 8 concurrent threads at the same time. Each query stream is at a higher rate than the update rate (usually 4 times higher). Both updates and queries are drawn from identical zipf distributions over to the objects queried or updated.

We provoke faults by launching a denial of service attack that unfolds in the following way:

- At time t from the start of the experiment a separate rogue client launches a denial of service (DOS) attack on 3 of the Tempest containers. Call them *victims*.
- At time $t + \Delta$ the rogue client ceases the attack.

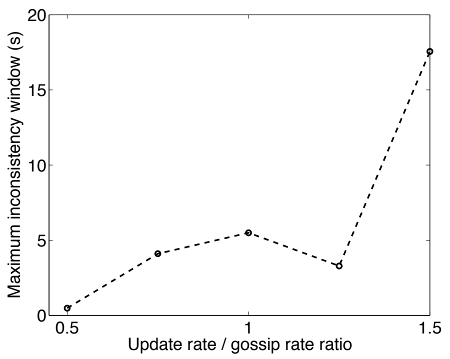

Figure 12: Inconsistency window during a 40 second DOS disruption. Update rate fixed at once every 40 milliseconds.

The attacker bombards 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 containers, hence the attacker also sends a computationally costly request. Victims spend CPU cycles responding to these requests while also dealing with the excessive incoming network traffic. These attacks don't actually cause the server to crash, but it does become stale.

A DDOS attack on a server 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 11 shows the number of "stale" query results on the y-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. The rogue client launches the attack about 7 seconds in the experiment and the duration of the attack is 40 seconds. Throughout the attack, 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. Once the attack ends, Tempest is able to gracefully recover.

Clearly, the ratio of the gossip rate to the update rate will determine the robustness of Tempest to this sort of DDOS attack. To quantify this effect, Figure 12 shows the inconsistency window as perceived by clients during a DDOS disruption. This is the period of time during which clients of a service see more than one stale query result during 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 one update every 40 milliseconds. The window is minimized when the gossip rate is at least as fast as the update rate.

5 Related work

Multi-tier solutions. The three tier model has been a tremendously successful paradigm for developing web services, especially since relying on a database management system (DBMS) for data storage simplifies the application. DBMS have long supported clustered architectures, offering load-ballancing, replication and restart mechanisms. However most databases provide ACID guarantees, and services built on transactional databases may incur performance penalties, especially during faults. Current state of the art application servers leverage such DBMS technologies. For example IBM WebSphere Q Replication [8] provides support for reactively replicating large volumes of data at low latency, typically targeting mission critical environments. Transactional data from the source replica is converted into messages, relayed to the target replica through a message queueing middleware, converted back and applied.

Traditionally, application servers offer persistent state support by mapping stateless business logic components to relational or object-oriented database items. For example the BEA WebLogic Application Server [3] provides clustering to ensure scalability and high availability for web services. It supports transparent replication, load balancing and failover for stateless Entreprise JavaBeans components. Stateful services must store the state on a persistent database — concurrency conflicts are avoided by relying on the underlying database locking mechanisms.

Replication. Chain replication [15] is a primary backup scheme built for high throughput generic storage systems. The protocol offers high data availability and strong consistency guarantees. Replicas are arranged in a linear chain topology, with update directed to the head of the chain and serially passed downstream until the tail is reached at which point the tail replies to the client. All queries are performed against the tail of the chain.

In [1] the authors present a one-copy serializable transaction protocol optimized specifically for replication. The protocol is scalable and performs as well as replication protocols that provide weak consistency guarantees. Updates are sent to all replicas while queries are processed only by the replicas that are known to have received and processed all completed updates.

6 Conclusion

In this paper we have presented Tempest, a new framework for developing time-critical web services. Tempest enables developers to build scalable, fault-tolerant services that can then be automatically replicated and deployed across clusters of computing nodes. The platform automatically adapts to load fluctuations, reacts when components fail, and ensures consistency between replicas by repairing when inconsistencies do occur. Tempest relies on a family of epidemic protocols and on Ricochet, a reliable time-critical multicast protocol with probabilistic guarantees.

References

[1] C. Amza, A. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites, 2003.

[2] M. Balakrishnan, K. Birman, A. Phanishayee, and S. Pleisch. Ricochet: Lateral Error Correction for Time-Critical Multicast, 2006. In Submission.

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

[4] BEA Systems, Inc. BEA WebLogic Server and WebLogic Express Application Examples and Tutorials, 2006. http://e-docs.bea.com/wls/docs91/samples.html.

[5] X. Défago, A. Schiper, and P. Urbán. Comparative performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Trans. on Information and Systems*, E86-D(12):2698–2709, 2003.

[6] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[7] 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.

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

[9] M. F. Kaashoek and A. S. Tanenbaum. An Evaluation of the Amoeba Group Communication System. In *International Conference on Distributed Computing Systems*, pages 436–448, 1996.

[10] L. Lamport. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, 1997.

[11] 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.

[12] A. Schiper, K. Birman, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.

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

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

[15] 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.

[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.