HTTP Session Replication for Tomcat Web-server (May 2012)

Chris Simoes[1], Alex Bednarczyk[1], Sponsoring Professor: Vijay Garg

[1]Software Engineering Master's Program at the University of Texas at Austin

Session replication is an important problem facing modern webservers. Amazon's S3 data store provides for an excellent mechanism to allow webservers to store their session externally to allow for easy session migration between webservers. We have built an implementation of Apache Tomcat that uses Amazon S3 to backup all of its sessions. We will discuss our design and discoveries, and then we investigate the overhead performance we will incur by using Amazon S3's service as an external data store instead of utilizing Apache Tomcat's default session replication techniques.

I. Introduction

HTTP is the Hypertext Transfer Protocol that is the foundation for the World Wide Web. HTTP functions as a request-response protocol allowing web servers to connect with clients such as web browsers. The web servers will return via HTTP the HTML files that make up all of the web pages on the Internet. HTTP is a stateless protocol, meaning that every request it handles is independent of all other requests made to the same web server. This allows the server to not have to retain state information about all of the requests made to it. In principle this simplifies server design because there is no need to dynamically allocate storage to deal with multiple requests in process. Also if a client dies in mid-transaction, no clean up should be necessary. However this has a big downside in practical applications where we want to know if a user is returning to our website.

To track users using HTTP various methods of session management have been created. The most common utilizes a cookie that is stored on the client browser to identify the client with each request. This also add the added overhead to the server of needing to track which cookie belongs to which user and allocating memory space to track information about each user. The server will pass to the client a cookie with a "session ID" that is then also stored in the web servers internal memory. This works well for development environments and small servers that are not required to run 24 hours a day, 7 days a week. For modern, robust web application that power sites such as Amazon's store a single web server is not sufficient. To power a large site, 10's if not 100's of web servers will be needed. If all of these web servers store their sessions in internal memory this presents a new problem.

What happens when one of these servers crashes? What happens when a server needs to be taken down for service? A simple answer has been to stop taking on any new sessions, and to allow existing sessions to logout or timeout before we shutdown the server. Unfortunately for modern busy websites, this can take hours or even days to complete. What we would really like, it to be able to immediately direct web traffic from one web server to another one without any disruption to the user's experience, and without having to wait hours or even days.

In order to accomplish this we have to allow the sessions on our web servers to "migrate" from one server to the next. We have to stop storing the session information only in internal memory on a single server. We have studied 2 approaches to this problem, and this paper will discuss them both. Our 2 researched solutions are to store our sessions:

- Central data store that is highly reliable and fault tolerant
- Distributed on our other webservers

Below we will discuss our research and findings by comparing and contrasting these 2 approaches. We will also

present our performance measurements of each approach, followed by our next steps in our research.

II. Environment

The Apache Tomcat project is an open source web server that is used to power some of the largest websites on the World Wide Web. We chose to do our research using the Apache Tomcat web server for several reasons. First, it is an open source project written in Java that will allow us to easily make modifications and investigate behaviors. Second, it has a large community around it that provides data and support to our research. During our initial investigation, it was very easy to find others that had our same goal in mind who were open and interested in sharing their work in performing session replication with Tomcat. Third, Tomcat already has a built in "high availability" mode (HA) that would allow us to compare with our approach of using a centralized data store.

Our Tomcat instances were all installed on Linux Ubuntu servers running in the Amazon EC2 (Elastic Computing) cloud. The Amazon cloud allowed us to easily create new servers to simulate a cluster of computers, and copy server configurations from one machine to the next. Also Amazon's EC2 cloud was preferable over other cloud providers since we intended to use Amazon's Dynamo distributed hash table for our centralized data store tests.

III. Dynamo

Amazon outlined their proprietary implementation of a highly available key-value store they named "Dynamo" [1]. Amazon needed a massively scaled key-value data store that provided high reliability and performance to run their huge ecommerce store. Dynamo is designed to provide an easy to use interface for the programmer that allows a guaranteed level of service. The actual implementation of Dynamo is hidden from the developer, and it is built on a distributed network of servers spread across the country that provide to the user an "always-on" appearance.

In Janaury 2012, Amazon announced the beta release of their DynamoDB web service. So our initial research we based on Amazon's DynamoDB web service. Unfortunately, it quickly became apparent to us that there were some significant drawbacks to successfully using DynamoDB as our central data store. DynamoDB is marketed as a NoSQL database service, but in reality it only stores information as strings. This is undesirable since our session information will be most naturally represented as an array of bytes. A further limitation is that the value of any given column is limited to 64,000 bytes of information. This is also undesirable since our sessions can be an arbitrary sized array of bytes that could likely be larger than 64,000 bytes. Upon learning of these limitations we abandoned consideration of DynamoDB for our key-value store.

We discovered that the correct web service to use is Amazon's S3 (Simple Storage Service). While Amazon S3's documentation does not specifically state that it is using Amazon's Dynamo technology, it does state that:

> *"Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, secure, fast, inexpensive infrastructure that Amazon uses to run its own global network of web sites."* [2]

Upon further inspection it became clear to us that Amazon's S3 service was the correct technology for us to build on. It was released in March of 2006, and it allows for writing, reading and deleting of key value pairs where the value can be from 1 byte to 5 terabytes in size. It allows for an unlimited number of objects to be stored, and it provides a 99.9% monthly uptime guarantee. As of March 2012, Amazon S3 is currently storing over 905 billion objects [3]. For these reasons we chose Amazon S3 as our centralized data store.

IV. Central data store that is highly reliable and fault tolerant

Our main body of research was to see if we could externalize Apache Tomcat's session management to a

centralized data store. Before we began writing any code, we first researched if anyone else had already tried this approach.

A. memcached

We discovered that no one had used Amazon S3 to externalize Apache Tomcat's session management, however we did find an interesting project called "memcached-session-manager" [4] that externalized session management. It used memcached [5], which is an open source, high performance, distributed memory object caching system. It provides an in-memory key-value store for small chunks of arbitrary data. While similar to our needs, memcached had short comings in comparison to Amazon S3's service. Memcached expects clients to understand which server to send data to, and which servers to fetch data from. In this sense memcached is not a centralized data store. Also memcached is built to use physical memory, and it is not ideal for persisting data on machines that may need to restart. Thus the durability of our data is in question.

The "memcached-session-manager" project did provide us with an excellent starting point for our research. The project was first released in October 2009 by Martin Grotzke, and it has subsequently has numerous updates and improvements. It supports Apache Tomcat 6 and 7, and it handle many special cases such as sticky sessions and server failover. We investigated the code thoroughly and decided to follow their design for integrating with Apache Tomcat.

B. Implementation

Our implementation is straightforward in theory. We would refactor the memcached-session-manager project to use Amazon S3 as a data store instead of memcached. In practice we learned a lot about the inner workings of Apache Tomcat and session management to complete this work.

Before we could begin, we first had to download and investigate the source code for Tomcat. We studied to see how does Tomcat load track and store sessions? We also investigated different approaches for integrating with Tomcat.

Tomcat has a "ManagerBase" class [6] that we extended to interface with Tomcat's session management. This class controls at a high level session persistence and storage. We then implemented a class called DynamoSessionService, that was responsible for actually finding and storing our sessions. It also provides the methods for serializing and deserializing our session objects. For serialization we chose to use Java's default serialization API, and this requires that all objects placed into our web server's session implement the "java.io.Serailizable" interface. For faster performance other serialization libraries exist. We also extended Tomcat's "StandardSession" class with our own version called "DynamoBackupSession" that tracks changes to our session so we can know if it is dirty in relation to our in memory cache. This wrapper class allows us to track all the extra attributes we need to in order to implement our externalized data store.

Apache Tomcat uses "Valves" to represent a component that will be inserted into the processing pipeline of a web request. We implemented our own valve called "SessionTrackerValve" that will monitor anytime a session is modified in internal memory. Our design that we copied from memcached-session-manager will only persist our session to Amazon S3 if the session has changed. If the session is accessed but not changed, then we continue to use our valid copy in our internal memory cache. This optimization is critical to minimize the number of external calls our web server makes to our external system.

In our first pass of refactoring the memcached-session-manager code, we changed all references using the memcached client to instead write and read sessions to the local disk. This allowed us to investigate and debug problems quickly of how do sessions get loaded and invalidated from memory. During this phase we learned that all backing up of session information happens asynchronously through a task service. So we wrote our own "BackupSessionTask" to handle the storing of sessions to disk. Once we got file system backups working correctly we built a stand-alone Amazon S3 client to store data in S3. This is the S3Client class in the org.simoes.session.s3 package. The S3Client is responsible for authenticating our program with Amazon Web services. It

is also responsible for providing us with a simple interface for putting and getting key value pairs from our external data store on S3. Upon completion of this component we integrated our S3Client with the core Tomcat codebase, and successfully ran Tomcat while it's sessions externally replicated to Amazon S3.

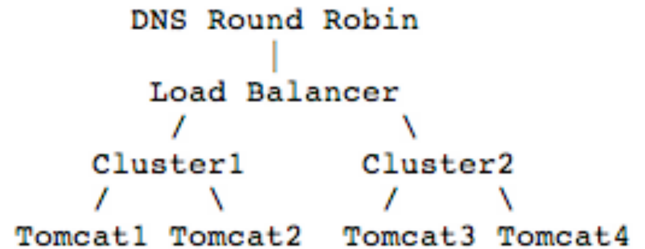
We next investigated the performance characteristics of our implementation. Obviously the big advantage of external session storage is the ability to easily change the web server a client is connecting to with no downtime. The big disadvantage is the latency that is potentially introduced by needing to make serialization calls over the network to load and store session information. We wanted to study this potential limitation to see how much latency we would need to trade for portability. In order to perform a fair performance assessment we also wanted to establish a baseline. We chose to also research using the "high availability" feature built into Tomcat that allows the web server to replicate its sessions to other Tomcat web servers on the network.

V. Distributed on our other webservers

Tomcat comes bundled with the ability to replicate sessions to other Tomcat webservers. The class Tomcat uses to perform this replication is the "SimpleTcpCluster" class [7]. It is a cluster implementation using a simple multicast protocol, and it is responsible for setting up a cluster and sending and receiving messages to other servers. The SimpleTcpCluster configuration enables all-to-all session replication that will track when a session changes, and then send the modified session to all other servers. This is a common configuration used, and we hoped that our implementation would perform close to as well as this reference implementation, but with the added benefits of a centralized store for the session information.

One down side of the SimpleTcpCluster approach is that it complicates your network architecture. While this implementation works fine for 2-4 servers, as you expand to 10's or even 100's of servers the network overhead grows linearly with the number of servers. This wastes a lot of network bandwidth, and introduces many unnecessary messages. A better approach recommended by Apache Tomcat is to group web servers into clusters behind a load balancer.

Tomcat Recommended Configuration



Like our Dynamo implementation, Tomcat's SimpleTcpCluster also assumes that all of the objects added to your web server's session implement the java.io.Serializable interface.

VI. Performance Analysis

A. Sample Programs

In order to test the performance of our Amazon S3 backed version of Apache Tomcat, we needed a sample servlet program that would store values in our session. We created a SampleLogin program that allowed a user to login to a website. It stores the user name and password in the session along with the current time for each request of the servlet. We added the time attribute so that the session's contents would change with every page reload, thus triggering the session to be replicated externally. We used our SampleLogin program to test both the Amazon S3 backed version and the default Apache Tomcat SimpleTcpCluster version.

B. Amazon Cloud

To test our Amazon S3 backed Tomcat implementation we launched 2 modified Tomcat webservers on the same server, where one used port 8080 and one used port 8081.

- http://ec2-23-22-79-203.compute-1.amazonaws.com:8080/SampleLogin/index.html
- http://ec2-23-22-79-203.compute-1.amazonaws.com:8081/SampleLogin/index.html

Both of these instances of Apache Tomcat would access Amazon S3 to load and store their sessions. In our development environment we saw a noticeable lag introduced the first time an S3Client was initialized. This is due to the time it takes to setup connections and verify credentials. In order to minimize this lag, Amazon recommends that programmers reuse the S3Client class, instead of instantiating a new one each time. We followed this design recommendation to improve performance. We also chose to locate our test server on the Amazon EC2 network for performance reasons. The lag in upload speeds from a home computer using a cable modem is noticeable when you are measuring in the 100's ms. The Amazon EC2 cloud provides impressive network response times and throughput particularly for calls between Amazon services (in our case between Amazon EC2 and Amazon S3).

For our SimpleTcpCluster configuration we launched 2 servers in Amazon EC2 with the exact same configurations. We then modified their configuration files so that they would broadcast session changes to each other. We again suspected that these 2 servers would benefit from being collocated on the Amazon cloud infrastructure. Since these were 2 separate boxes we struggled with showing that the sessions were replicating properly between the 2 servers. By observing the log files for the 2 servers it was clear that many network calls were occurring between the servers, but modern browsers discouraged us from trying the hack the session id. We ultimately deemed that since this was our base line, and also due to the extensive use of caching by the default Tomcat implementation, using time to create a way to hack setting the session was not a top priority. We instead focused on our Apache Tomcat backed by S3 implementation tests, and wrote tests to ensure the default Tomcat implementation did have to store and replicate many session changes.

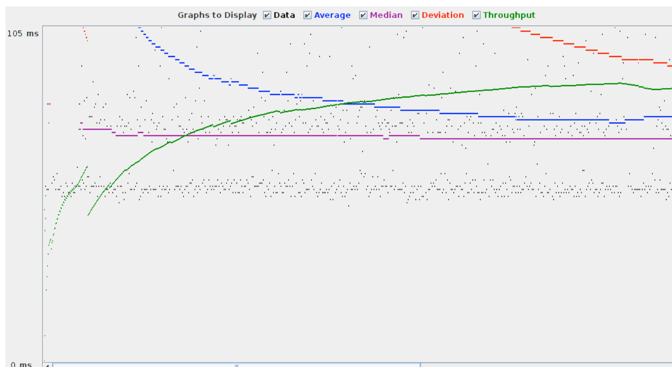
C. JMeter

To automate our testing we chose to use the open source project JMeter [8]. JMeter is designed to load test functional behavior and measure performance. JMeter is used by the Apache family of projects for load testing of Tomcat and the Apache web server. JMeter however is not a web browser, and it is not well suited to test web pages that contain Javascript or require a lot of client side processing. Thus we kept our SampleLogin test program free of any of these dependencies so we could focus on testing the performance of servers storing their session externally.

For our Apache Tomcat web server backed by Amazon S3 tests, we configured JMeter's cookie manager to enable session tracking. We then pointed JMeter at our server listening on port 8080. JMeter would contact that server, and pass it a username and password to login. When Apache processes this request it creates a new session that it will then store in Amazon S3. For discussion sake, we will say this session has a session id of "1234". Then JMeter would follow the "Click here to stay logged in" link. This would update the time attribute in session 1234, which would again trigger Apache Tomcat to backup the session to Amazon S3. JMeter then goes to the same web server, but this time instead of using port 8080 it uses port 8081. It turns out that Apache Tomcat sees this as a request from the same client so JMeter gets a request for the same session id of 1234. However, the web server on port 8081 is a different webserver running in a different JVM from the one running on port 8080. Our 8081 version of Apache Tomcat now looks up session id 1234 in it's local memory, but it predictably does not find one there. So it then makes a remote call to Amazon S3 to see if session id 1234 can be found in our external session store. Session id 1234 is found, so the 8081 Apache Tomcat web server loads this session into local memory, and to the user they continue to access the website uninterrupted even though they are now being served from a completely different web server.

This test is performed over 1000 times, and then JMeter creates a nice plot of the performance. We were encouraged to find that the average response time was 77ms. As we dug

into this finding we realized this is due to our implementation caching the changed session in internal memory and then asynchronously queues up a task to backup the session to Amazon S3. Below the blue line represents the load time of each page while the green line represents the throughput we are achieving. The overall throughput of our Tomcat version was 761 requests per minute.



We also performed the same type of test against our SimpleTcpCluster default Tomcat setup. This test showed an average response time of 58ms. Again the blue line represents the load time of each page while the green line represents the throughput we are achieving. The overall throughput of the default Tomcat version was 994 requests per minute.

This was not overly surprising given the fact that we only had 2 servers running. We would expect this performance to degrade as we added more servers to a cluster. We also expected this to perform well given that the code has been improving for the past 5 years. We would expect that we could improve our implementation's performance since we have so far spent no time on code optimization. Also given that we are measuring in milliseconds, while the difference of 19ms is statistically meaningful, in the real world this was not particularly concerning.

VII. Future Research

We had several ideas on where our research should next proceed. While our preliminary analysis was encouraging to simulate more real world conditions we will need to test our implementation with larger objects stored in our session. Our tests stored only a few bytes, while a production web server would probably store sessions on the order of 100 kilobytes to even megabytes. We also would like to test our solution on 4 and then 8 servers running concurrently. We suspect that our implementation will scale better, as it is only bottlenecked by the ability of Amazon S3 to scale and Amazon boasts that this scaling problem is effectively solved for S3.

We would be interested to see how much adding 4, 8 or even more servers to Apache Tomcat's default configuration slows down the servers. Testing on servers outside of Amazon EC2 would also be a useful data point. Finally we would like to improve our implementation. While we did implement the ability to put and get from the Amazon S3 external store, we did not implement the ability to delete or expire old session values stored there.

VIII. Conclusion

We were very encouraged by the progress we were able to make on creating a webserver that persisted its sessions to an external data store. Amazon's S3 key value store provides a reliable, scalable, distributed central store that proves to have very fast response times in the Amazon cloud. Our implementation was able to service web request in less than 100ms, and this leads us to believe that this is a viable implementation for modern websites to build upon.

References

[1] Dynamo: Amazon's Highly Available Key-value Store. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter

Vosshall and Werner Vogels. Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007.

[2] Amazon Simple Storage Service (S3) http://aws.amazon.com/s3/

[3] Amazon Web Serviecs Blog http://aws.typepad.com/aws/2012/04/amazon-s3-905-billion-objects-and-650000-requestssecond.html

[4] Memcached-session-manager Project on Google Code http://code.google.com/p/memcached-session-manager/

[5] Memcached Home Page http://memcached.org/

[6] Tomcat's ManagerBase class http://tomcat.apache.org/tomcat-7.0-doc/api/index.html?org/apache/catalina/session/ManagerBase.html

[7] Tomcat Clustering Documentation http://tomcat.apache.org/tomcat-7.0-doc/cluster-howto.html

[8] Apache JMeer project http://jmeter.apache.org/