2009 33rd Annual IEEE International Computer Software and Applications Conference

Clustering Versus Shared Nothing: A Case Study

Jonathan Lifflander, Adam McDonald, Orest Pilskalns
School of Engineering and Computer Science
Washington State University, Vancouver, WA 99686
{lifflander, mcdonald, orest}@vancouver.wsu.edu

Abstract

The massive growth of the Internet paired with the rise in dynamic website content has increased the need for scalable network architectures. While these various network architectures should be transparent to the client, the speed, reliability, and maintainability of the system depends on the particular architecture that is implemented. This paper will discuss the findings from a case study that tests the speed of two software architectures that are often implemented to build scalable web application systems. The first architecture is server clustering with shared resources. Server clustering can be defined as a group of servers that directly share resources and actively partitions work based on work loads. Thus client traffic to the cluster can be distributed across several physical machines each running an instance of the application server. The other architecture is a shared nothing design, where application servers do not share resources, except for a dispatcher (load balancer). This paper addresses the question, what is the performance overhead of adding application servers into a tiered system.

1 Introduction

When designing a scalable web solution the decision to scale horizontally or vertically is domain dependent. Scalable web architecture often include at least three specific tiers: web servers, applications servers (middleware), and the database [1]. Often caching fabric is also added to the middleware. Database requirements dictate middleware or web server configuration [2]. For example the database may be a MySql Cluster where the middleware can access a host of database nodes. This case study was was initiated by a web based mapping company who contracted this research team to examine architectural tradeoffs between using a heavy middleware layer such as JBoss or going with a lightweight servlet engine. To address this question we examine two architectures with various configurations. The first architecture (clustered) employs the JBoss Application Server to share resources between EJB containers living on various VMs on an array of servers. The second architecture (shared nothing) uses a servlet engine and bypasses the container model and uses the database as a resource sharing mechanism thus eliminating middleware inter-communication and session sharing. The requirements for sharing resources in the middleware may influence the decision to use clustering versus shared nothing, however, we examined the problem as a simple performance evaluation. Care was taken to ensure that the functionality remain the same in both systems under study.

Web application systems that support dynamic content must work at a high speed in conditions where many clients are connecting at a high frequency and data is being transferred back and forth. When clients connect for the first time, a session identity is assigned and this identity is used to connect the client back to their current context in subsequent connections. This connection between client and web server allows the client to experience a stateful session, built upon a stateless protocol http. To accomplish this, the machine they were previously connected to must be remembered or the session data will have to be recomputed. Recomputing might be as expensive as navigating to the database to retrieve the data. Depending on the setup, this session information may be cached by the application server or stored temporarily in a database. In this case study, the tests for the systems under consideration where specifically modeled upon the integration tests written by the web company. These integration tests are designed to test the end to end functionality of their system. The traffic simulation was constructed using JMeter, thus the traffic was uniform, which is not representative of a deployed commercial system. This lack of accurate traffic representation was a serious concern to this case study. However, we made the assumption that high traffic and bandwidth tests would be the most stressful, thus the most useful in assessing performance. This case study attempts to discover the time difference between using JBoss, which has overhead to support server clustering, and a more rudimentary setup that uses Apache Tomcat and queries the database for the ses-

0730-3157/09 $25.00 © 2009 IEEE
DOI 10.1109/COMPSAC.2009.124

sion identity to reconnect sessions. This paper addresses the question, what is the performance overhead of adding an application server into the design of a hight traffic web application and what are the tradeoffs.

2 Related Work

In "Testing J2EE Clustering Performance and what we found there" [4] the authors evaluate how JBoss clustering affects the performance of the system as a whole. Their experiment contrasts a single-node JBoss setup with that of a two-node JBoss cluster. The single-node JBoss setup is considered a simple shared nothing solution. JMeter [5] was used to simulate J2EE application usage and put the servers under heavy load. Load balancing bottlenecks in the cluster setup were discovered initially due to the use of mod_jk2 [6]. Load balancing was pushed to the client level to resolve this issue.

The initial setup disregarded state replication in order to set a baseline measurement for response time and throughput. The cluster setup resulted in one half the response time and almost double the throughput when compared to the single-node shared-nothing approach. These performance gains can be attributed to optimizations performed by the JBoss cluster. When state replication was turned on, the clustered setup still out performed the single-node setup but the performance degraded significantly. The response time was now five times that of the initial setup and the throughput was cut to one third. This experiment illustrates how expensive state replication can be. However, JBoss by itself is still not as simple as a stand alone servlet engine. If you add entity beans (Enteripse Java components for persistence) into the picture, you may encounter even more performance problems.

Another analysis of application server clustering using JBoss [7] compares a single-node setup with a four-node JBoss cluster. The focus of this experiment was to study how the performance of JBoss can be improved by exploiting the variety of deployment options provided by the JBoss platform, such as using different web containers (Tomcat 4/5, Jetty), tweaking the deployment descriptors, and optimizations with data access. The SPECjAppServer2004 [9] benchmark was used to determine beneficial modifications to the platform. The only significant performance improvements were due to data access optimizations, which are outlined in detail by the paper. Although the authors did not note any significant differences in performance between the single-node and cluster setup. This study did not address the issue of removing the application server overhead.

3 Server Clustering

The first networking architecture that was tested implements *server clustering*, where the dispatcher (or load balancer) forwards the connection to a server in the cluster using a built-in algorithm. Server clustering allows the application servers in the cluster to share resources and communicate by passing messages. For example, a client may connect to one machine when they login, however, that machine becomes too busy to service the client, the client's session can be transfered to a machine with a smaller work load. For this experiment, an application server JBoss was used to implement server clustering. JBoss is open-source software that serves as a container for Enterprise Java Beans (EJB2 and EJB3). Enterprise Java is a scalable component model that uses the Java API. Compared to using Apache Tomcat by itself as a web server, JBoss is a heavier layer that has much more overhead, but provides many extra features.

The main reason that server clustering is used is to allow the web application system to be easily expanded in a parallel fashion. Instead of upgrading a server when more performance is needed, a clustering architecture supports the simple augmentation of a new server. JBoss makes it easy to implement clustering when setting up the system and it simplifies the later addition of more servers. Therefore, JBoss provides the functionality to create and maintain an easily scalable web application system.

3.1 The Load Balancer

In order to try to distribute the connections evenly to the servers in the cluster, the dispatcher must use a load balancer. Typically the load balancer is part of the cluster because the cluster can provide information about the architecture of the network that helps the load balancer evenly distribute the connections.

A load balancer acts as a proxy, forwarding the connection to the appropriate server, dependent upon some predetermined algorithm (least connected, round robin, etc) or which machine was previously connected to the client. If the load balancer is persistent it stores in cache a history of relations between client and server, allowing clients to reconnect to the same machine. Although this information speeds up the process of locating the machine, it requires overhead and can slow down the server, creating a bottleneck. Alternative methods include a scheme where each machine stores its clients and the connection is forwarded through the network until it reaches that machine. However, this increases the traffic on all the servers in the cluster and if the cluster is large it could greatly reduce performance.

If the load balancer is not persistent it will forward the connection to any machine (based on the routing algorithm)

in the cluster each time the client connects and all the session data will have to be recomputed by that machine. This technique introduces performance issues because often the computed session information over a long connection period can be substantial. However, this method does free up memory (in the load balancer and the other machines) and it often allows the load balancer to maintain a more balanced cluster.

There are other methods that do not use persistence that can perform well. Instead of the session data being stored on the server, it can be stored by the client in the form of a cookie if the client is a web browser (which is almost always the case when session data is an issue). If this information is encrypted by a server-side algorithm, sensitive data can be kept secret, even if the client stores the data. This alleviates the server from storing the information and also allows any server in the cluster to handle the client's requests. In a similar scheme the session data is stored by the client using *URL rewriting*, when the data is actually stored in the URL. The major disadvantage in using this method is that as the session data grows more information will have to be transferred with each request, deteriorating performance.

Whether or not the session data should persist on the server or client or be recomputed depends on how the web application is programmed. If fewer session variables are used, then recomputing this data may not be a problem or this data could just be stored in the URL. An example would be a website that displays dynamic data to the user with minimal user input. Alternatively, if a lot of session data is needed, then it may be beneficial to just store the data server-side.

In summary, we assume that session data may need to be replicated across server side machines. To test the performance issues associated with replicating data across machines we examine the load balancing configuration where persistence is turned both on and off.

3.2 Other Considerations

An application server allows you to create components that live in a feature rich container. By complying with a component model, via a standardized interface, developers can take advantage of security protocols, clustering, transactions, data persistence as well as many other features with a minimal development overhead [3]. However, with EJB2 developers do have to learn a fairly extensive framework riddled with configuration files to take advantage of JBoss. The configuration issue has been addressed in EJB3, where default values and only one configuration file is required per component. Some developers may not need many features of the application server, for example, your load balancer may adequately handle session persistence relieving clustering concerns. One of the major differences between an EJB and a pure servlet model is the ability or lack of ability to generate code that accesses the database. Container managed entity beans represent rows of data in the database and the code for persisting or accessing that data is auto generated by the EJB container. With the servlet (in our case study the shared nothing approach) or bean managed entity beans, the code for accessing the database is written manually. That being stated, you can always create entity beans that handle their persistence manually. However, keeping this code synced with the schema of your database can be a daunting task. Looking from a purely performance perspective you might assume that the manually written code is faster, so we purposely addressed this question while testing our two architectures. Another concern in EJB2 was the expressive capability of the query language which was needed for the auto generation of access code to the database. The EJB query language was not equivalent to SQL which made certain queries impossible, forcing the developer to write with bean managed entity persistence.

4 Shared Nothing

The shared nothing network architecture is a design in which all the application servers do not share resources or communicate. The major advantage of this setup is that it reduces overhead, while still implementing parallelism for the web application along with horizontal scalability. The only server that communicates with all the application servers is the dispatcher, which forwards the client's connection like the clustering dispatcher. Most of the overhead that JBoss carries is from all the services it employs for sending and receiving messages, and supplying the interface for various plug-ins. With a shared nothing architecture, server-side software such as Apache Tomcat can be used instead because the extra layers and services are not needed.

4.1 The Load Balancer

When using a shared nothing architecture, the load balancer must either cache the session identity and the corresponding server identity or store this information in the database. If there is a large amount of this data then the information will most likely be stored in the database and a trigger or stored procedure will be invoked to clear the old data (typically, the session identity will persist for 20 minutes). Otherwise, the load balancer can just store the information locally and look up the machine based on the session identity, if a machine was already assigned.

If the load balancer caches this information, it becomes a single point of failure because if its memory fails all the session information will be unreachable (because the machine it is stored on will be unknown). In this experiment,

we tested the speed difference of storing this data on the load balancer or on the database.

5 The Experiment

For this experiment two application servers were used along with a load balancer and a server that stored the database. The following are the hardware specifications of the servers:

- Dispatcher/Load Balancer, Application Server 1, Application Server 2
  - Processor: Pentium III 930 MHz
  - Memory: 515168 Kb
  - Software
    - Linux Virtual Server (LVS) - NAT
    - Java JDK 1.6.0_05
- Database Server
  - Processor: AMD Athlon 64 3200+, 1004 MHz
  - Memory: 1993304 Kb
  - Software
    - mySQL 5.0.51a
    - Java JDK 1.6.0_05

To test server clustering JBoss 4.0.5.GA was used and for shared nothing Apache Tomcat 6.0.16 was used.

Six trials were conducted, testing various configurations of clustering and shared nothing. The dependent variables are the roundtrip times for four operations: creating a user, deleting a user, creating a project, and uploading a file. JMeter was used to create the series of tests. Table 1 shows the configurations for each trial run:

5.1 Trial Run 1

For trial run 1, a shared nothing configuration was used with Apache Tomcat and JDBC. The load balancer did not persist the relationship between clients and servers. Instead, the database was used to store this data and the load balancer distributed connections from the client using a round robin algorithm. This placed the burden of connecting to the database on each web sever to find the appropriate session identity. The round robin ensured that clients were consistently hitting different machines per session. This case should provide the most performance contrast to clustering since clustering can minimize communication with the database by sharing information between JBoss machines. The shared nothing architecture is forced to make trips back to the database for each client call. Table 2 provides a breakdown of the times for this configuration:

Table 2. Trial Run 2 Roundtrip Times

Operation	Tests	Failures	Average Time
CreateUser	10	0	20 ms
DeleteUser	10	0	20 ms
CreateProject	1000	0	222 ms
Upload	1000	0	3647 ms
Total/Ave	2020	0	1925 ms

5.2 Trial Run 2

For trial run 2, a shared nothing configuration was used with Apache Tomcat and JDBC. The load balancer used a cache to persist the relation between the client and the corresponding machine. Since the load balancer reconnects clients to the same machine, the servers do not have to reconstitute session information from the database. In this case the shared nothing should perform well, since it does not have the overhead of the application server and it communicates with the database only as needed to complete its task. Table 3 provides a breakdown of the times for this configuration:

Table 3. Trial Run 2 Roundtrip Times

Operation	Tests	Failures	Average Time
CreateUser	10	0	31 ms
DeleteUser	10	0	19 ms
CreateProject	1000	0	40 ms
Upload	1000	0	147 ms
Total/Ave	2020	0	93 ms

5.3 Trial Run 3

Trial run 3 used JBoss with server clustering enabled. Entity Beans were used to automatically interface with the database via the container and the load balancer did not persist the session data. This configuration placed the burden of sharing session information on the clustered architecture. In addition the entity beans used container managed persistence, so the the database access code was auto-generated. Table 4 provides a breakdown of the times for this configuration:

5.4 Trial Run 4

Trial run 4 used JBoss with server clustering enabled. Entity Beans used container managed persistence to access the database and the load balancer had persistent connections activated. Since the load balancer handles the issues

Table 1. Test Configurations

Trial Run	Application Server Software	Database Layer	Load Balancer Persistence	Load Balancer Scheduling
1	Apache Tomcat	JDBC	off	round robin
2	Apache Tomcat	JDBC	on	N/A
3	JBoss	Entity Beans	off	round robin
4	JBoss	Entity Beans	on	N/A
5	JBoss	JDBC	off	round robin
6	JBoss	JDBC	on	N/A

Table 4. Trial Run 3 Roundtrip Times

Operation	Tests	Failures	Average Time
CreateUser	10	0	34 ms
DeleteUser	10	0	32 ms
CreateProject	1000	0	82 ms
Upload	1000	0	76 ms
Total/Ave	2020	0	79 ms

of session persistence, it would seem likely that the JBoss server should be more performant than the case where the JBoss cluster must handle sessions persistence. In addition Table 5 provides a breakdown of the times for this configuration:

Table 5. Trial Run 4 Roundtrip Times

Operation	Tests	Failures	Average Time
CreateUser	10	0	66 ms
DeleteUser	10	0	60 ms
CreateProject	1000	0	39 ms
Upload	1000	0	50 ms
Total/Ave	2020	0	44 ms

5.5 Trial Run 5

Trial run 5 used JBoss with server clustering enabled. However, for this trial the entity beans used JDBC code to access the database. In addition the load balancer used a round robin technique to distribute connections. This trial run should be contrasted with the entity bean trial with the same load balancer configuration. One would expect the JDBC to perform better, since the code is not auto generated by the container. Table 6 provides a breakdown of the times for this configuration:

Table 6. Test Run 5 Roundtrip Times

Operation	Tests	Failures	Average Time
CreateUser	10	0	24 ms
DeleteUser	10	0	24 ms
CreateProject	1000	0	47 ms
Upload	1000	0	64 ms
Total/Ave	2020	0	55 ms

5.6 Trial Run 6

Trial run 6 used JBoss with server clustering enabled. Once again the entity bean used custom JDBC code to interface with the database. This time the load balancer persisted the session data in its cache. This trial should be compared to the entity bean with the same persistence configuration. Table 7 provides a breakdown of the times for this configuration:

Table 7. Trial Run 6 Roundtrip Times

Operation	Tests	Failures	Average Time
CreateUser	10	0	78 ms
DeleteUser	10	0	77 ms
CreateProject	1000	0	162 ms
Upload	1000	0	87 ms
Total/Ave	2020	0	124 ms

5.7 Critical Comparisons

5.7.1 Trial 1 vs. Trial 2 (Database vs. Caching)

The first comparison is between using the database to store the session identities and the corresponding machine that holds the state, or having the load balancer cache alleviate this problem. In trial 1, a shared nothing approach used the database to store session data using JDBC as an interface to the database. The second trial used the built-in load balancer mechanism for storing this information.

The first trial, which used the database took 1832 ms. longer on average to process all the requests. The CreateProject task took 182 ms. longer and the Upload task took 3500 ms. longer to complete using the database, but the CreateUser actually was about 11 ms. faster with the database. The delete user task was the only anomaly in the data set. Overall, having the load balancer cache the data greatly outperformed using the database to look up the machine for a session identity. Although this result seems obvious, there is the issue of scalability and reliability of external load balancers. However, it appears that using the database is not a practical option to address those issues.

5.7.2 Trial 2 vs. Trial 6 (Shared Nothing vs. Clustering)

In trial 2, a shared nothing architecture was used with JDBC and the load balancer cached the state data. This is comparable to trial 6 in which a server clustering architecture was employed with JDBC and the load balancer also cached the data as well.

On average, the shared nothing trial took 93 ms. and the clustering trial took 124 ms. yielding a 31 ms. difference in the averages. Therefore, the overall results of the trial showed that the shared nothing architecture performed slightly better than the server clustering architecture.

For the first two trials of the CreateUser process, which was only tested 10 times per trial, trial 2 (shared nothing) took 31 ms. whereas trial 6 took 78 ms. (server clustering). The third trial, which tested the CreateProject operation and ran 1000 times, took 40 ms. on trial 2 and 162 ms. on trial 6. The fourth trial, which tested the Upload operation and ran 1000 times, took 147 ms. on trial 2 and 87 ms. on trial 6. Overall, the shared nothing architecture exceeded the performance of the clustering architecture on average, although on the fourth trial clustering performed better than shared nothing.

5.7.3 Trial 3 and 4 vs. Trial 5 and 6 (Entity Beans vs. JDBC)

JDBC and Entity Beans are two different layers on top of the database that can be used to interface with it. Entity Beans with container managed persistence offers more automation than JDBC. From the data it appears that the container managed entity bean outperformed their bean managed JDBC counterparts. In both of these trials, server clustering is used and the state is held by the load balancer. Therefore, the only major difference between the trials is the layer used to interact with the back-end database server. This was a major surprise since the functionality was the same, only the the access code was changed. Trial 4, which used Entity Beans, outperformed trial 6 on average by 80 ms. On all four tests Entity Beans performed substantially better than JDBC.

6 Conclusion

We initially guessed that that clustering would out perform shared nothing if the system did not have persistent session based load balancing to distributed client connections. This guess was vindicated, but we did not have a use-case where this type of approach would actually be deployed in practice. However, a system under heavy load may drop sessions from machines that are overburdened, in this case the session would need to be reconstituted from the database. If this happens often enough your system could resemble a system where round robin load balancing is distributing traffic. The real surprise of this study was the lack of performance gain of using bean managed persistence via JDBC.

References

[1] P. S. Yu and A. Dan, "Performance Analysis of Affinity Clustering on Transaction Processing Coupling Architecture", *IEEE Transactions on Knowledge and Data Engineering*, pp. 764-786, 1994.

[2] M. Balasubramanjam, K. Barker, I Banicescu, "A Novel Dynamic Load Balancing Library for Cluster Computing", *The ISPDC/HeteroPar'04*, 2004.

[3] R. Panda, Z. Debu and S. Rahman, L. Reza, S. Lane, C. Derek, 'EJB 3 in Action", Manning Publications Co., Greenwich, CT, USA, 2007.

[4] D. Rossi and E. Turrini, "Testing J2EE Clustering Performance and What We Found There", *Proceedings from the First IEEE Int'l Workshop Quality of Service in Application Servers*, October, 2004.

[5] JMeter, http://jakarta.apache.org/jmeter/, 2009

[6] mod_jk2, http://tomcat.apache.org/connectors-doc/, 2009

[7] S. Kounev, B. Weis, R. Buchmann, "Performance tuning and optimization of J2EE applications on the JBoss platform", *In In Journal of Computer Resource Management, Issue 113*, 2004

[8] S. Mellor, K. Scott, A. Uhl, D. Weise, *MDA Distlled: Prinicples of Model-Driven Architecture*, Addition Wesley, 2004.

[9] SpecJAppServer, http://www.spec.org/jAppServer2004/, 2004