Clustering with Terracotta

A qualifying pre-project Report
submitted to the Faculty
of the
Agder University College

By:

16 May 2008

Approved:

Abstract

In today's java community, modern enterprise application products have more constraints and requirements then ever. High availability, application scalability and also good performance are required, which means an application is needed to be deployed on multiple JVMs, in other words, it has to be clustered or distributed. It is essential for the application to scale out well, has better performance and less complexity during development of clustering.

This master thesis focuses on clustering with Terracotta which is a JVM level clustering technique. First I start analyzing the complexity when an application comes into scale-out, and also analyzing the shortcomings of the common approaches to clustering an application. Then I do a deep dive to the Terracotta and demonstrate how to utilize Terracotta to conquer these problems with emphasis on providing scalability and high performance using natural java programming. Finally, various scenarios are made as benchmark tests. The final result has shown that Terracotta as redundancy solution is strong recommended to be implemented for high availability and high scalability

Preface

This project concludes the two-year Master of Science program in Information and Communication Technology (ICT) at Agder University College (AUC), Faculty of Engineering and Science in Grimstad, Norway. This project has been carried out from January to June 2008.

First of all, I would like to thank Arild Andas from DevoTeam Company and Selo Sulistyo at Agder University, for excellent supervision and guidance throughout the project period. My discussion with them has really helped me clarify several obstacles during my project. Without their help, this project and report would not have been the same.

I would also like to thank all the experts working at Terracotta for their patience and contributions. They gave me a lot of precious advices.

Sun Wei
Grimstad, March 2008

Table of contents

List of Figures 4
Chapter 1 Introduction 6
Chapter 1 Introduction 6
1.1 Thesis Definition 6
1.2 Background 7
1.2.1 Cluster 7
1.2.2 Java Virtual Machine 8
1.3 Current Clustering State of Art 8
1.3.1 Database Strategy 8
1.2.2 Clustering Based on Complicated Java API 9
1.2.3 Clustering Based on Cache Mechanism 10
1.4 Problems of Traditional Clustering Techniques 14
1.4.1 Implement Serializable Interface (coarse-grained replication) 14
1.4.2 Break Object Identity 14
1.4.3 Low Performance 15
1.4.4 Need Callback Mechanism 15
1.4.5 Essential of Complexity of Current Clustering Techniques 15
1.5 Motivation 15
1.6 Delimitations 17
1.7 Thesis Overview 17
Chapter 2 Theory 18
2.1 What is Terracotta (DSO) 18
2.1.1 Network Attached Memory 18
2.1.2 What does NAM do? 19
2.2 Architecture and How Terracotta DSO Works 21
Chapter 3 Clustering with Terracotta 24
3.1 Aspect-Oriented Programming (AOP) 24
3.1.1 Java Language Specification 24
3.1.2 Java Memory Model 24
3.1.3 ASM 25
3.1.4 How it Works 25
3.1.5 Fine-Grained Replication 28
3.1.6 Thread Coordination 29
3.2 Hub and Spoke 31
3.2.1 Cluster Hub to Avoid SPOF 32
3.2.2 Batched Field-Level Changes 33
Chapter 4 Scenarios & Discussion 34
4.1 Pitfall Overview 34
4.1.1 Loading Problem 34
4.1.2 Replication Problem 37
4.2 Terracotta Scenario and discussion 41
4.2.1 Scenario1 – fix separately loading problem 41
4.2.2 Scenario2 – fix limitation of physical memory 44
4.2.3 Scenario3 – fix n-1 ACK problem 45
4.2.4 Scenario4 – where to use Terracotta 46
Chapter 5 Conclusion and Future work 61
5.1 Sum up 61
5.1.1 Complexity problem: read inefficient from database leads to complexity 62

5.1.2 Terracotta solution: Transparent, inter-JVM thread coordination 62
5.1.3 Implicit problem: limitation of heap space which leads undesirable effect on database 62
5.1.4 Terracotta Solution: virtual heap of Terracotta 62
5.1.5 scale-out problem: writes inefficient and Database abuse (based on ORMapping) leads to hard to scale-out 63
5.1.6 Terracotta solution: virtual heap of Terracotta 63
5.1.7 what is Terracotta 63
5.2 Conclusion 63
5.3 Future work 64

List of Figures

Figure 1.1 clustering solution based on database 9
Figure 1.2 clustering based on complicated java API 10
Figure 1.3 cache clustering solution 11
Figure 1.4 cache solution: get method 12
Figure 1.5 cache solution: set method 13
Figure 1.6 cache solution: put method 13
Figure 1.7 cache solution doesn't preserve object identity 14
Figure 1.8 comparison based on traditional clustering solutions 16
Figure 2.1 plug-in Terracotta 20
Figure 2.2 scale-out application based on Terracotta 22
Figure 3.1 bytecode instructions 25
Figure 3.2 object graph in heap 26
Figure 3.3 add Terracotta to the heap 27
Figure 3.4 Terracotta intercepts the changes to the object graph in heap 27
Figure 3.5 Terracotta transfers changes to other JVM 28
Figure 3.6 java synchronization mechanism 29
Figure 3.7 Terracotta synchronization mechanism 31
Figure 3.8 architecture overview of Terracotta 32
Figure 3.9 batching changes to Terracotta server 33
Figure 4.1 naïve loading database solution in distributed environment 34
Figure 4.2 naïve loading database solution in single JVM 35
Figure 4.3 add cache as objects shared layer 36
Figure 4.4 add cache layer in the distributed environment 37
Figure 4.5 add cache layer leads object stale problem 38
Figure 4.6 sticky session solution 38
Figure 4.7 network overhead problem based on copy mechanism of distributed cache 39
Figure 4.8 database overhead problem based on invalidate mechanism of distributed cache 40
Figure 4.9 local JVM Problem of distributed cache 41
Figure 4.10 distributed cache solution without Terracotta 42
Figure 4.11 loading times from database based on distributed cache solution without Terracotta 42
Figure 4.12 accessing time based on distributed cache solution without Terracotta 43
Figure 4.13 distributed cache solution with Terracotta 43
Figure 4.14 loading times from database based on distributed cache solution with Terracotta 44
Figure 4.15 accessing time based on distributed cache solution with Terracotta 44

Figure 4.16 object fault in Terracotta 44
Figure 4.17 load object from Terracotta server instead database 45
Figure 4.18 accessing time of loading object from Terracotta server after 3 seconds 45
Figure 4.19 fix n-1 ACK problem with Terracotta 46
Figure 4.20 baseline application 47
Figure 4.21 add hibernate 48
Figure 4.22 enable 2nd level cache 50
Figure 4.23 problem based on enable 2nd level cache solution in distributed environment 51
Figure 4.24 enable 2nd level cache with Terracotta 53
Figure 4.25 results of Terracotta administer console 54
Figure 4.26 Tomcat 1 54
Figure 4.27 Tomcat 2 54
Figure 4.28 Tomcat 3 55
Figure 4.29 Tomcat 4 55
Figure 4.30 JMeter aggregate report for 1 application server 55
Figure 4.31 JMeter aggregate report for 2 application servers 55
Figure 4.32 JMeter aggregate report for 3 application servers 55
Figure 4.33 JMeter aggregate report for 4 application servers 56
Figure 4.34 detached pojos 57
Figure 4.35 detached pojos with Terracotta 59
Figure 5.1 typical distributed solution 61

Chapter 1 Introduction

In today's world of java open-source, higher availability requirements, better performance, the adoption of cheaper hardware and lower budget are in demand at critical levels of efficiency in the real market. These factors are driving java developers to adopt various clustering approaches to make applications as "failure-proof" as possible meanwhile with high performance and great capability of scale-out. However clustering has never been an easy-done job during the life-cycle of application development. A close look at java language itself, according to the original contract of java, we could conclude the application works almost perfectly as long as it runs within a single JVM. Unfortunately, things start falling apart when the application is intended to be clustered, in other words – if it is intended to be deployed on multiple JVMs, the problem with scaling out breaks this promise that java made. And this also leads developer get parts of their energy focusing on the things beyond the development of business logic and generating extra code to maintain data consistent among multiple JVMs. It turns out that there are two big obstacles in java today, namely: scale-out and complexity.

1.1 Thesis Definition

This thesis proposes a simple and powerful alternative: clustering with Terracotta which is a JVM level clustering solution, as it is the only technology available today that provides the runtime clustering solution that supports highly available and scalable java applications.

In this article, we focus on the scalability using natural java language semantics by introducing Terracotta.

We start from the current state of the art in clustering. First we exploit various clustering techniques, analyze how the strengths and weaknesses of each and come out what is the real reason of complexity of current clustering techniques. Second we are moving on to eliminate clustering concerns out of the application layer down into JVM, which is the simplest solution without impacting the business logic of applications.

Here we introduce Terracotta step by step: What Terracotta is what its architecture is and how it works? In particular, we focus on the Terracotta DSO (distributed shared objects) which is the core of Terracotta. We will analyze how this DSO component virtualizes multiple JVMs and makes them equivalent to a single JVM, how this helps reducing database load and reducing development efforts and dependencies on applications servers and frameworks.

Terracotta has been deeply investigated in the five aspects of the following in this article:
1. JVM clustering
2. Object identity preservation
3. Shared virtual heap (network attached memory)
4. Clustered thread mechanism
5. POJO clustering

And various benchmark testing scenarios are given to illustrate the resulting performance which will be analyzed across the following functional dimensions:
1. Scalability
2. Availability

3. Fine grained sharing of data
4. High performance

We also describe its impact on the way of our coding from our point of view for better performance.

Finally, we use a real world project to illustrate how easy to seamlessly integrate existing project with Terracotta.

The research result is given as “Terracotta is a strongly recommendation for requirement of high availability, high salability and high performance”.

1.2 Background

In this section, we present relevant background material of clustering. It includes some brief description and definition of relevant terms.

1.2.1 Cluster

A **computer cluster** is a group of loosely coupled computers that work together closely so that in many respects they can be viewed as though they are a single computer. The components of a cluster are commonly, but not always, connected to each other through fast local area networks. Clusters are usually deployed to improve performance and/or availability over that provided by a single computer, while typically being much more cost-effective than single computers of comparable speed or availability [1].

High-availability clusters (also known as failover clusters) are implemented primarily for the purpose of improving the availability of services which the cluster provides. They operate by having redundant nodes, which are then used to provide service when system components fail. The most common size for an HA cluster is two nodes, which is the minimum requirement to provide redundancy. HA cluster implementations attempt to manage the redundancy inherent in a cluster to eliminate single point of failure [1].

1.2.1.1 Scalability

Scalability: From Wikipedia, in telecommunications and software engineering, scalability is a desirable property of a system, a network, or a process, which indicates its ability to either handle growing amounts of work in a graceful manner, or to be readily enlarged, the ability for a distributed system to easily expand and contract its resource pool to accommodate heavier or lighter loads. Alternatively, the ease with which a system or component can be modified, added, or removed, to accommodate changing load [1].

1.2.1.2 Availability

Availability: From Wikipedia, availability refers to the ability of the user community to access the system, whether to submit new work, update or alter existing work, or collect the results of previous work. If a user cannot access the system, it is said to be unavailable. Generally, the term downtime is used to refer to periods when a system is unavailable [1].

1.2.1.3 High Availability

High Availability: From Wikipedia, it is a system design protocol and associated implementation that ensures a certain absolute degree of operational continuity during a given measurement period [1], which can avoid single point of failure (SPOF).

1.2.1.4 Failover

Failover: From Wikipedia, failover is the capability to switch over automatically to a redundant or standby computer server, system, or network upon the failure or abnormal termination of the previously active server, system, or network. Failover happens without human intervention and generally without warning, unlike switchover [1].

1.2.2 Java Virtual Machine

A Java Virtual Machine (JVM) is a platform-specific (operating system and hardware) program that implements a well-defined, platform-independent virtual machine [2]. Here we don't mean to give an introduction to JVM. The only thing about JVM that we need to keep in mind is that JVM operates on Java bytecode, which is normally (but not necessarily) generated from java source code and a JVM can also be used to implement programming languages other than java [1].

1.2.2.1 Bytecode

Java Bytecode: Java Bytecode is the form of instructions that the java virtual machine executes (e.g. normally used getField and putField to access an object's fields).

1.3 Current Clustering State of Art

The following are three common solutions to cluster java applications and we will see how these solutions leave the developers no choice, namely, business logic including the code required for clustering. This means the developers are burdened to build extra code for infrastructure service of clustering, in other words, the developers can't only focus the business logic they really need to focus on and this results in all these solutions are unnecessarily complex.

1.3.1 Database Strategy

This might be the easiest way to provide high availability. We can just go to the relational database and write all of our application data into it. Then we have a highly available solution, and we can plug-in application servers as well. This is a very good strategy and a relatively common development pattern. This is the kind of strategy that hibernate helps us with writing our pojos (plain ordinary java object) and mapping them to the database. This strategy has been used in many solutions: the entire stack simply writes all of its data into the database. As we can imagine, once began to write all of our scratch data into the database, it will become our centralized point for all our application servers and this also means that there will be a significant overhead in our database layer as our usage grew. This is depicted by the following Figure1.1: clustering solution based on database

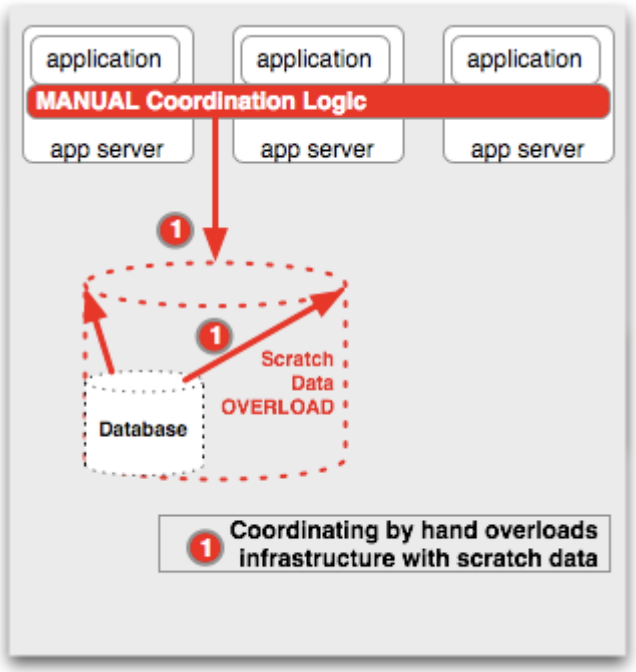

Figure 1.1 clustering solution based on database

The problem within this database solution is we should only use database to store only the "system of Record" information as the black dash line depict.

For example, within a particular web environment, an application will carry on a conversation with the user during parts of the normal web flow. That web flow with the user describes what is happening as the application interacts with the user, namely it is generating scratch data. Scratch data is defined as object oriented data that is critical to the execution of series of java operation inside the JVM, but may not be critical once a business transaction is complete [1]. The term "scratch data": is derived from students' use of a piece of scratch paper to make interim calculations for a multiple choice examination. This scratch paper is critical to the student during the exam, yet once the final answers are turned in; it is not longer of value to the student or the instructor [1]. This is analogous to the data that is generated during the conversation flow the application has with the user. So, in each user's request/response cycle, the application is generating data that doesn't necessarily need to be stored permanently across the lifecycle of the entire application. It only needs to be stored during the lifecycle of the conversation with the user. Hence, storing those data in the database for the purpose of clustering, for the purpose of scale out and high availability means we are overloading our database and the database becomes the chief bottleneck in our clustering solution. Besides, manual coordination is a very difficult and complex task and forces developers work in an unnatural way to code explicitly for the clustering case, which is unnecessary within the situation of running application in a single JVM. Then the code is far more complex and performance suffers greatly as our usage grew.

1.2.2 Clustering Based on Complicated Java API

Another common used clustering solutions is complicated Java API. Basically they are JMS, RMI, or RPC to synchronize the destination and security state across the servers which means various JVMS in the cluster can share in-memory object state data and be able to signal or coordinate among nodes. Illustrated by Figure1.2: clustering based on complicated java API

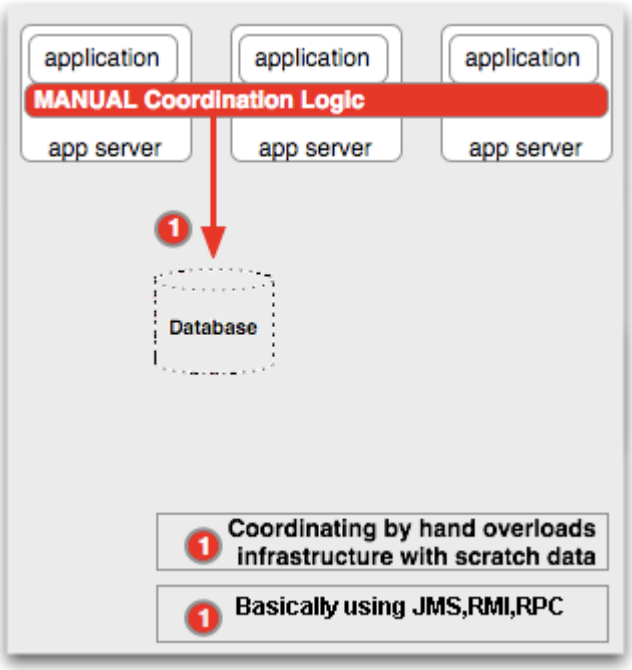

Figure 1.2 clustering based on complicated java API

But they all ultimately boil down to significant amount of custom development in the form of JMS messaging or RMI, RPC coordination. All of these solutions require a very significant change to the developer's application and require custom development by the application developers to implement this application level high availability and high scalability. This mechanism available today can greatly increases the complexity of java applications and usually burden the developers with extra code way out of utopian business logic. It turns out in larger, more error prone source code, longer development period, and difficult management of deployed applications.

The problem with this solution is, taking JMS as an example, we have to write a JMS layer to keep various object state data in the stack coherent across all the nodes in the cluster. If we want to distribute the user session, we distribute shared data through the JMS layer and we also have to make sure that all of these data consistent. For the sake of that, we have to define points at which changes made by one JVM are distributed across all the other nodes. Basically what we do is to serialize the changed object graph on the local machine and ship them across the wire. The main pain point here is there are overheads of serialization and maintaining the JMS layer. As if we do this distribution too often or we add more nodes into our cluster to handle swelling traffic, the performance of the entire system will suffer and there is no way to less this, otherwise the data coherent would be broken.

1.2.3 Clustering Based on Cache Mechanism

Over the years, a lot of products have been conceived on purpose to help relieve the burden of the developers from writing underlying infrastructure services like complicated communication code and eliminate the overheads of database as a temporary store for scratch data. But basically, in traditional cache clustering solution, the cache is resident inside the JVM. Typically it is exposed to the developers in some kind form of hashmap. Illustrated by Figure 1.3:

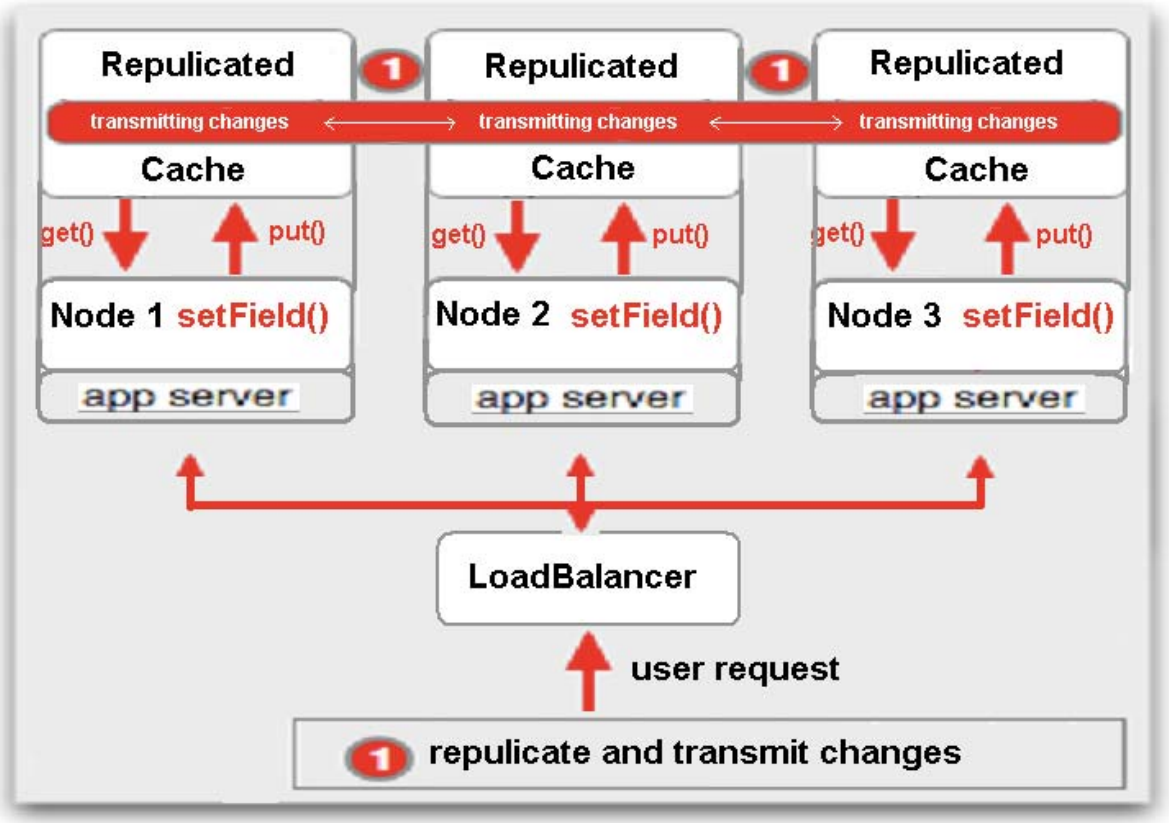

Figure 1.3 cache clustering solution

As you can see in the figure we have to use **put()** methods to store the updated data into the cache and at this point the cache service will take responsibility for transmitting these changes to the other application servers across all the nodes. The cache server will notify the clustering caches when changes occur, and listener callback mechanism will inform other caches when detecting something elsewhere has changed. Then we could use **get()** methods to retrieve the data that we put into the cache previously.

Based on the description above, we can tell this kind of elaborate hashmap is very intuitive to use and the only thing that we need to do is to make sure that the objects stored in the cache have to implement the Serializable interface. However, this type of cache mechanism has some known limitations relating to clustering.

Now let's take an online store as example to illustrate these problems of traditional cache solution. This is a conceptual online store which only contains a product A and has a distributed cache across the clustered servers. Consider that we have three application servers as Figure1.3. We have a hashmap structure installing on each application server and wire them up. When a user requests on one of the application servers to view product A. Node 1 retrieves the object from the cache using some kind of get method as Figure 1.4 illustrate:

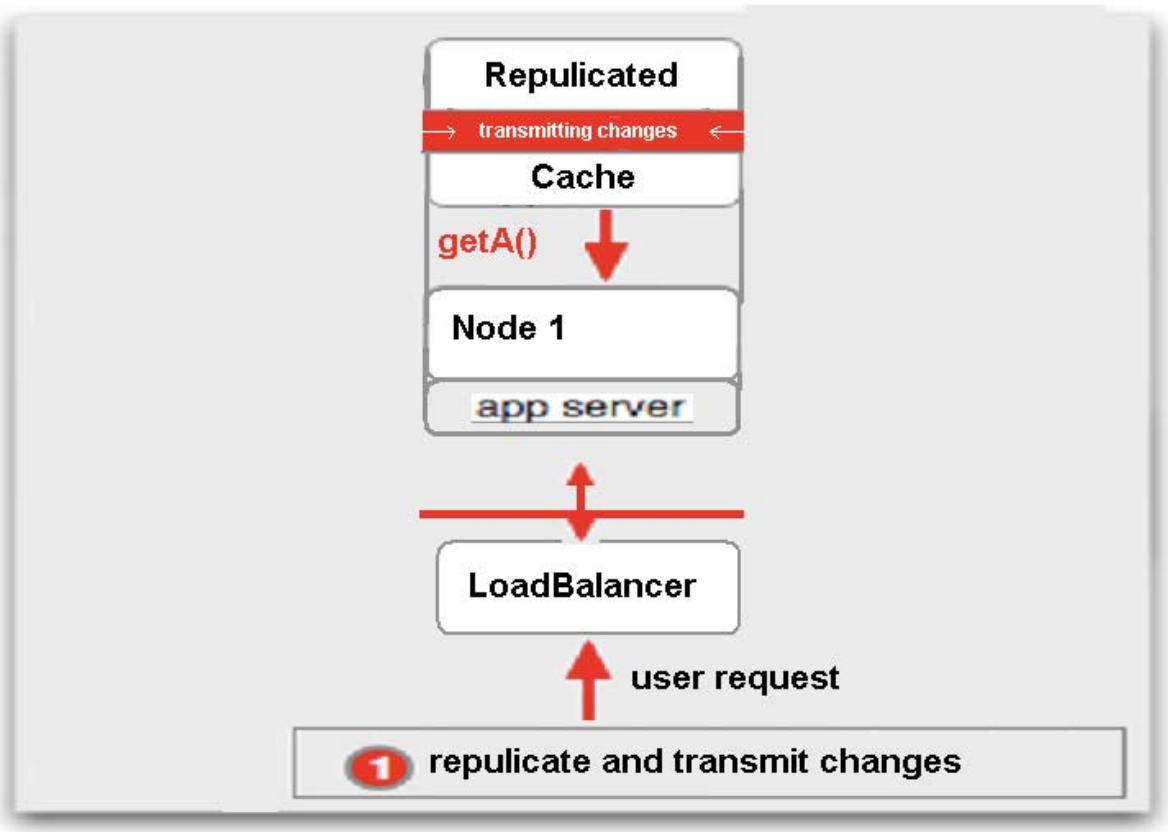

Figure 1.4 cache solution: get method

Then if the user wishes to update some information (say, the price of product A), he would update the product A he previously retrieved from the cache by calling **setPrice()** as Figure 1.5. And then put it back by calling **putA()**as depicted in Figure 1.6.

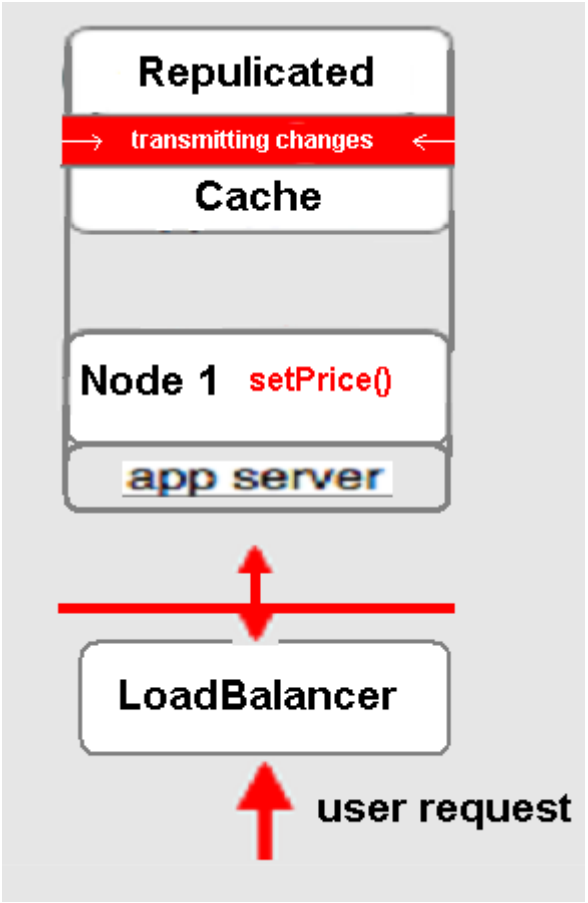

Figure 1.5 cache solution: set method

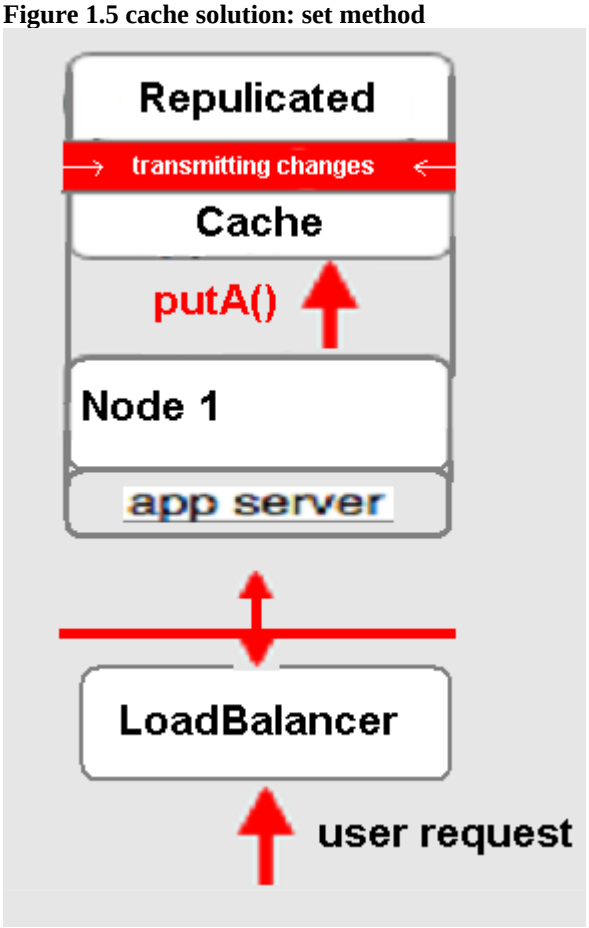

Figure 1.6 cache solution: put method

At this point, the cache service takes responsibility to transmit this update to other two nodes in the cluster through serializing the entire product A over the wire.

The problem within this solution is not too much different from other two above. Worse, developers have to do like **get()** fields from a clustered hashmap, after update the field by calling **set()**, they also have to remember to **put()** fields back into the map and share the data through signaling. Code is still hard to write and maintain in this kind of unnatural programming way.

1.4 Problems of Traditional Clustering Techniques

In this section we conclude some of the limitations of traditional clustering techniques.

1.4.1 Implement Serializable Interface (coarse-grained replication)

All of these solutions force that objects must implement java.io.Serializable interface. Simply put, serialization, whether native java serialization or custom serialization, is a transmission and rebuilding mechanism for java objects. This would normally not be an issue if we got all of the source code, but it could be a real affliction if we are working without other people's source codes or with a third-party library.

1.4.2 Break Object Identity

The most important problem of these traditional solutions for clustering is that all of them typically rely on java serialization to ship changes across the nodes and serialized object is written out in native bytes. This means that the cache mechanism, either RMI or JMS has to maintain an exact copy of a given object for objects moving between servers and any change to an element in the map will force the developer to code explicitly to guarantee data coherent. In other words, the developers may check out any number copies of a given object for use. Although these objects are logically the same, they are out of sync and cannot be updated. It is all up to the developers to decide how to handle it. In short, this cache solution doesn't preserve object identity.

Take a classic example to illustrate this problem as showed in Figure1.7 [3]

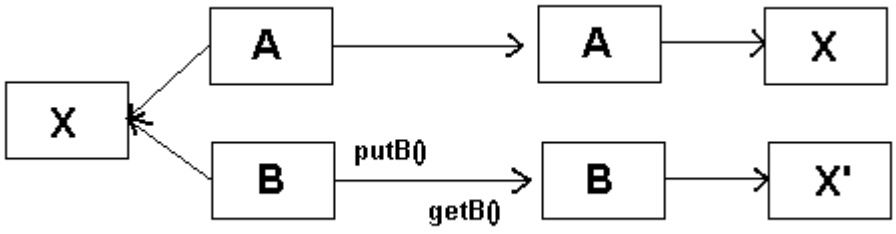

Figure 1.7 cache solution doesn't preserve object identity

In this scenario, we have objects A and B both maintaining a reference to object X. If we **put()** B in cache, release it and then **get()** it. When B is returned to the system, X is returned along with it as a completely new object [3]. The true is java Serializable interface tells java virtual machine to disconnect our serialized classes or objects into bytes and then reassemble them back which will result in X and X' seems two different objects in the system. Actually they are semantically the same object. Then this is the burden on developer to figure out how to maintain referential integrity. And usually it forces the developer to program like a database

designer to use a foreign key to treat X and X' are identical, to break our domain model apart and manually stitch it back with key mechanism which is a unnatural programming way and make our code extra complex.

1.4.3 Low Performance

Worse, we are hampering our application performance here: if we are only changing one field of an object, yet we have to serializing the entire store and ship it across the entire nodes. If the object size is huge, then even a tiny little change to one single field of an object would trigger serializing the whole object across the cluster. This would be unnecessarily expensive.

1.4.4 Need Callback Mechanism

Besides above, there is a trivial problem left in our online store example: after node 1 updated product A's price, the cache has already finished distributing our update. Now the price of product A in cache 2 has updated, but the price in node 2 (the user previously called **getA()** on the cache 2) still is the stale one. We have to appeal to some sort of callback mechanism to register our objects to get informed when changes occur, which no doubt makes our code even more complex.

1.4.5 Essential of Complexity of Current Clustering Techniques

When java was first introduced, developers were living in a magical utopian world. The original promise of java was that the developer was supposed to only focus on their own business logic, other things like infrastructure services will all be handled by JVM. This means that JVM was supposed to take care of everything below the application automatically and extremely. In fact, this all works well and separation of business logic from infrastructure services is a brilliant and elegant design as long as the application runs within the confine of a single JVM. However the developers have to deal with the underlying infrastructure services if the application is intended to be clustered or scaled out. So now, we have scale out happening today, the contract made by Java start falling apart.

1.5 Motivation

All of these above solutions are unnecessarily complex. The idea here is to find some way to focus developers on their own business logic and separate business logic from infrastructure services when clustering.

A good clustering solution should at least offer the following in addition to high availability:
1. **Simple**:
It should be as simple as possible which means that we only need to write less code to achieve clustering or to add more nodes to our cluster for high availability.

2. **Fast**:
It should be high performance as adding more nodes, namely, linear scale. In other words, deploying an application on one node or more nodes should not have much different of decrease of the overall system performance.

And In fact, as I mentioned above that enterprise java has become more and more complex in the last few years, developers seem have various accesses for clustering an application, however none of them doesn't burden our network, burden our class hierarchy and burden our developer. We already have java its original semantics: **synchronized(), wait()/notify()**, locking mechanism, etc. Isn't this enough to extend its original semantics for multiple machines to cooperate as one by certain plug-in? Here comes out Terracotta. And clustered applications would be much easier to write and maintain by using Terracotta as a JVM plug-in and we can safely run the application on multiple nodes without any program modification.

The bottom line is to focus developers on the things they really need to focus on, in other words, business logic. All of these clustering mechanism of moving objects between JVMs has nothing to do with the core business logic whether we are using JMS or custom API solutions. And java language has off the shelf multi-thread API, plus there is nothing theoretically different between multi-thread and multi-node computing. The only thing that we need to do is to extend the java thread mechanism and memory model on multiple nodes environment. And that's exactly Terracotta did. Then we can offer scalability and high availability using natural java.

The following Figure 1.8 illustrates the current state of the art in clustering for high availability and how the strengths and weaknesses of each tool lead to a resulting impact on the business logic.

	Scalable	**Serialization-Based**	**Manual Cooperation**	**Performance**	**Impact on Business Logic**
Database	**Low**	**Need**	**Medium**	**Low**	**High**
JMS	**Low**	**Need**	**High**	**Medium**	**High**
Cache	**Low**	**Need**	**High**	**Medium**	**High**
JVM	**High**	**No Need**	**No Need**	**High**	**Low**

Figure 1.8 comparison based on traditional clustering solutions

Instruction of above functional dimensions:
Scalable: Assume that all the changes must be replicated, data integrity is guaranteed, and such solution will not burden the network or the database otherwise scalability can be done due to network or database as a bottleneck.

Serialization – Based: Serialization normally leads to a lot of overhead to the system and also increased traffic on the network. It is an important factor that we need to take into account with regards to performance and maintenance as more nodes adding.

Manually cooperation: Explicit code for cooperation across JVMs is needed today for most clustering solutions, because thread mechanism works fine only within a single JVM boundary. Infrastructure service' transparency is also important to the easy use of a solution.

Performance: we also hope that a solution could provide high performance while it offers high availability.

1.6 Delimitations

Since our research is based on real market clustering techniques, far more resources would be needed which is beyond a master thesis project. The ideal test environment is at least 7 computers for testing to avoid intervention:
1. for JMeter simulation test,
2. for Apache HTTP Server as a load balancer.
3. for Terracotta as a primary clustering server.
4. for Terracotta as a standby clustering server.
5. as the first application server.
6. as the second application server.
7. as a database server

I can't simulate this clustering environment in my computer due to cup capability and process limitation of operation system. So I can only test the high availability and all the test data showing high performance are not able to be trusted.

1.7 Thesis Overview

Chapter 1 gives an introduction to the problems of traditional clustering techniques

Chapter 2 is an introduction to the concepts of Terracotta

Chapter 3 explains the architecture of Terracotta and how it works
Chapter 4 shows all our results from different scenarios and discusses the results

Chapter 5 gives the conclusion of our project. It also point out possible further work based on this thesis.

Chapter 2 Theory

When we get ourselves into any major, large scale data center running mission critical java applications today, whether it be a stocking system, a train booking system or online shopping system, we are more likely to see 2 or 4 cheap commodity hardware running Linux or Windows. Unprecedented requirement of higher performance, higher availability and adoption of cheaper commodity hardware are driving developers to scale out the applications. However scaling out a java application traditionally has not been an easy way as we have seen.

According to the above reasons, JVM level clustering takes advantage of the natural semantics of java language and it completely hides the underlying cluster services from the application. It is the simplest solution to achieve high scalability and high availability without impacting business logic. And this is exactly how Terracotta was conceived.

Here there is a term that we should know: **Terracotta DSO (distributed shared object)**. Normally, we refer Terracotta as Terracotta DSO, former is just a name and latter is the technique that Terracotta uses for clustering.

2.1 What is Terracotta (DSO)

Basically, we could take Terracotta as a clustering plug-in at java runtime and it extends the java thread mechanism and memory model to enable multi-node distributed system working as simple as single java application. The most important thing is that all the underlying infrastructure service of clustering is totally transparent to the developers, which makes developers can only focus on their own business logic.

From Terracotta organization, Terracotta is Network Attached Memory (NAM) [4].

2.1.1 Network Attached Memory

Terracotta coined the term "Network Attached Memory" to explain what Terracotta does for Java applications [4]. Any Network Attached Memory (NAM) implementation must honor 2 fundamental principles [4]:

2.1.1.1 NAM must look just like RAM to the application

Constructors, Wait / Notify, synchronized(), == and .equals() should all work as expected [4]. An easy way to think about NAM's interface to the application is that an application that supports NAM should run with or without a NAM implementation [4]. For example, if you use a Network Attached Storage (NAS) file server, the application is not aware, nor does it need to be, of whether it is writing to a local or a networked disk [4]. The application itself does not change if NAS is present or not. NAM is exactly the same [4]. If NAM is not present, the application will continue, writing to local RAM. Likewise, the application need not change with the introduction of NAM [4].

2.1.1.2 NAM must work as an infrastructure service

That is, NAM must run as a driver inside the JVM but also as a separate process apart from the application cluster [4]. This is due to the fact that like networked file systems, the memory must survive whether or not your application is running [4]. Otherwise, NAM does not provide the true value of connect/disconnect on-the-fly that one would expect when compared to Network Attached Storage [4].
Terracotta provides Java applications with this powerful NAM concept whereby developers get clustering and high availability because [4]:

1) The memory is shared and external to the JVM, allowing for coordinated cross-JVM I/O[4].

2) If the JVM crashes, the data survives any crashes and restarts [4].

Developers get HA (high availability) and scale as a result of plugging in NAM [4].

2.1.2 What does NAM do?

As we have already mentioned traditional database approach achieve high availability by putting all the data into a database server. However, sending all the data including scratch data to the database will burden the database and the database server will become a bottleneck as too much overhead. Although this database solution could provide us high availability, the capacity is bound by the database. And this is why it can't offer scalability as more nodes added to the system.

With NAM, we could think of Terracotta as giving us a way to mark certain objects as critical, and when these objects join the heap. This critical part of the heap needs to be stay persistent, transactionable and safe. That is what NAM really doing. Basically, this part of the heap will be moving out to the network server, Terracotta server, in case of some nodes fail or any nodes need to see what other nodes are doing. So network attached memory is really a powerful paradigm. It is just like network attached storage, where we can swap in a network file system without rewriting. Illustrated by Figure 2.1

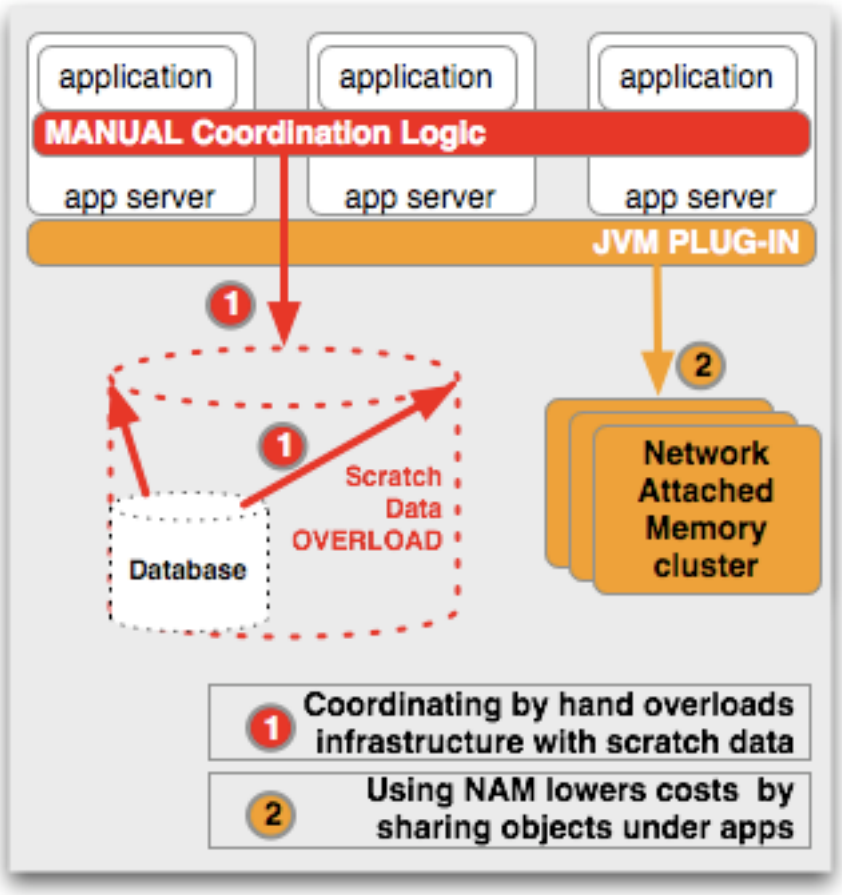

Figure 2.1 plug-in Terracotta

In the above figure, NAM is depicted as NO.2. Here is a high level of what Terracotta does: it plugs into the JVM, and the effects are the following.

2.1.2.1 Reduce Database Load

It's like Terracotta provides an adorable store to the heap of JVM. Without it, we have using O/R mapping to store java objects into the database. Now, we eliminate the database overload because the intermediary writes that were happening during the conversation with users are no longer being pushed down to the database. Alternatively, all these scratch data populate in Terracotta and this eliminates the database overload.

2.1.2.2 High Availability and Less Resource Cost

Now, we have all our intermediary data in Terracotta. This means that each JVM can get the data from Terracotta instead of database to provide high availability in case any other JVM down. And more than that, because we no longer put all the scratch data into database, we don't need serialization mechanism which is a big overhead to the system. High performance can achieve by avoiding frequent database operations.

2.1.2.3 High Scalability

The reason that why it's better than database layer if pushing all of these scratch data into Terracotta is simply that Terracotta is working on the object land. We don't need to handle cluster infrastructure by ourselves anymore. All the O/R mapping that is required to convert an object into a relational table, store it in the database and then take it back out of the database are not necessary. Terracotta instruments our objects as they are loaded into the JVM and provides clustering for us without us having to manually maintain the synchronization between different objects.

2.1.2.4 Simple and Natural Code

As Terracotta provides this adorable store to the heap, then, from the developer's point of view, building a clustering application is as easy as writing to the heap: there is no more manual coordination logic needed, there is no more clustering logic needed. The application is the same as writing as simple pojos (plain old java object).

2.1.2.5 High Performance

Terracotta can provide higher performance. **The first point** that we have seen is that we don't need serialization with Terracotta. It instruments our objects as they are loaded into the JVM and provides clustering for us, in other words, it can automatically maintain objects synchronized and coherent without an intervention by the developers. No serialization and heap level replication means higher performance. **The second point** is Terracotta doing fine-grained replication on shared objects. **The third point** is that changes are sent are only the changes that are made to the objects while the objects are resident in memory.

We will explain the second and third point later after we deep dive on how Terracotta DSO works.

2.1.2.6 Large Virtual Heap

As Terracotta was devised as a network attached storage file system, we can actually page out of any shared objects we don't need on the fly to the Terracotta server and Terracotta server can spill the data to the disk, on the other side, once we need those data to get our job done, we could ask Terracotta server retrieve the data back from the disk. It seems that our application has a very large heap far exceeding machine' physical RAM.

2.2 Architecture and How Terracotta DSO Works

As we've said that Terracotta DSO is a runtime solution of clustering at JVM level instead of the application's level and it works 100% transparent to the developers without manually handling underlying infrastructure service of clustering as multiple nodes working within one single JVM.

Terracotta uses ASM to manipulate application classes as those classes load into the JVM [5]. Here we will not get into this in technical details, if readers got interest, please reference appendix.

Figure2.2 denotes a conceptual high level perspective of Terracotta's architecture and how it works.

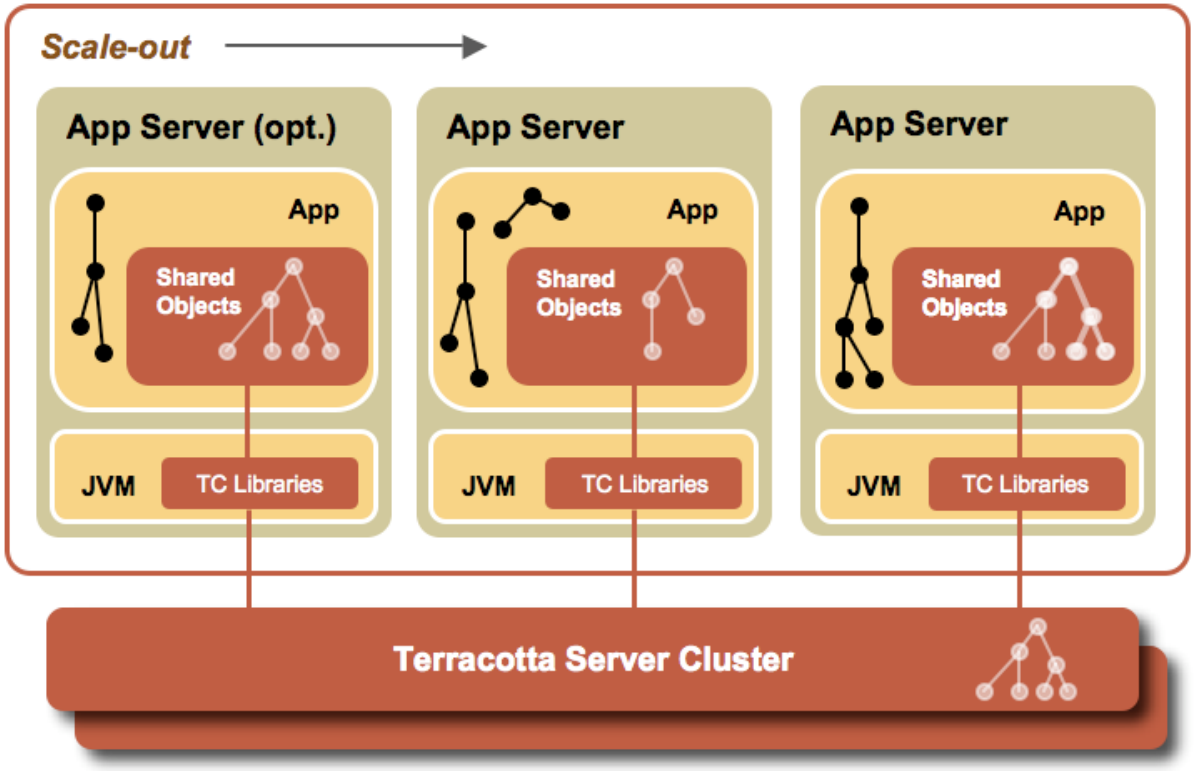

Figure 2.2 scale-out application based on Terracotta

First thing that we should notice: Terracotta is that big red server at the bottom and we can cluster this server, so there will not be a single point of failure as long as we can run two of them. In short, Terracotta server supports cluster by itself.

And from the Figure above, we have the web application started inside each JVM. The picture is kind of inverted (JVM outside the application server). The app server box meant to denote one virtual machine, but we use that big and short of can at the bottom represent it for convenience. And we could observe that there is a Terracotta library installed inside each virtual machine. When one JVM starts up, that JVM will start up with Terracotta's library, which means a bunch of jars basically hook into it and every application loaded into that JVM through its class loader is capable of being clustered.

The white and black things with dots and lines are objects graphs, which denote Terracotta to get in there and think about which objects to cluster and which not: the black dot-shape things are object graph not being clustered by Terracotta. Coexisted with white dot-shape, which are the object graph being shared by Terracotta. This means that with Terracotta we have the ability to decide for performance reasons what to share and what not to share.

Note that all the web applications connect down to Terracotta depicted by red pipes above. Each pipe normally is a TCP/IP point to point socket. Terracotta multiplexes in its own library inside that socket, so we could have a lot of multi-thread conversations going on simultaneously without lots of net resource.

We number the above application servers: server1, server2, server3 from left to right. And we could see that although three servers all get different unshared-object graphs, there is something different on server2: from the above picture, server1 has those shared objects which are exactly identical to those on server3 and those in Terracotta server. But server2 in the middle has grabbed a subset of the objects. So server2 only share part of the object graph, because it only needs this part of the objects to get its work done. It pulls those four little white dots that it needs from the Terracotta server on the fly and no more, which means if some other parts of the objects change Terracotta won't act pushing the changes into the node unless the node needs them. In other words, this is how Terracotta starts to deliver salability for clusters and we will explain this later.

So basically, we can conclude Terracotta's features as following:

1. **Fine-grained Replication**
2. **JVM Coordination**
3. **Large Virtual Heaps**

Chapter 3 Clustering with Terracotta

In this chapter we will do deep dive on what we just mentioned in the pervious chapter.

First, let's recall java language itself. Java is a multi-threaded language with convenient facilities to enable parallel multi-threaded applications and all the language semantics work well in one single JVM. Now, if we want to cluster JVMs running and cooperating just like one. It implies that we have to extend java virtual machine to make the JVM heap virtualization and thread execution working across the whole clusters; in short, JVM level clustering is a crosscutting infrastructure service which maintains java language specification and java memory model. The following are the techniques used in Terracotta.

3.1 Aspect-Oriented Programming (AOP)

Terracotta use Aspect Oriented Programming (AOP) to inject clustering concerns into JVMs at loading time and implement transparent clustering for developers, which works like a plug-in to the java virtual machine to provide clustering infrastructure service. And this has been done based on ASM bytecode manipulation framework. We will see later.

3.1.1 Java Language Specification

We say Terracotta provides seamless clustering solution without impacting our existing projects as working within one JVM. An important reason is Terracotta maintains java language semantic across clusters. The following are the core components among semantic features.

1. **Preserve Object Identity**
2. **Pass by Reference**

Actually, they are the identical, which means if object identity is broken, then there is no pass-by-reference.

As we have discussed in section 1.4.1 of chapter 1, most traditional clustering solutions relies on the use of java serialization. However this will break object identity which means pass-by-reference semantics can not be used any longer. Here comes developers usually have to manually maintain the relationship between objects; codes are becoming complex and hard to maintain. Worse, this kind of based on java serialization replication is an object level clustering solution. This implies there is no way that we could transfer exact changes of the object graph over the wire instead we have to move the entire object to guarantee no stale data. Performance will degrade as object graph swells. And this is why we call these kinds of solutions as coarse-grained replication.

3.1.2 Java Memory Model

Here we will not go to details about JSR-000133 [6] Java Memory Model [6]. We emphasize that Terracotta provides shared virtual heap and clustering locking mechanism by using AOP to plug into the java memory model. It intercepts accesses to the memory model about read and write operations and keeps the consistent data within the virtual clustered heap in Terracotta server. And it also reinterprets build-in thread coordination mechanism of java language by detecting multi-thread programming keyword like synchronized, wait, notifyAll,

etc. So, all the thread coordination mechanism to the local JVM is extended to the cluster-wide.

3.1.3 ASM

Terracotta uses ASM bytecode manipulation framework to achieve this transparent clustering solution we described above.

Terracotta intercepts bytecode instructions like (not an exhaustive list) [5]:

BYTECODE	Injected Behavior
GETFIELD	Read from the Network for certain objects. Terracotta also has a heap-level cache that contains pure Java objects. So GETFIELD reads from RAM if-present and faults in from NAM if a cache miss occurs.
PUTFIELD	Write to the Network for certain objects. When writing field data through the assignment operator "=" or through similar mechanisms, Terracotta writes the changed bytes to NAM as well as allowing those to flow to the JVM's heap.
AASTORE	Same as PUTFIELD but for arrays
AALOAD	Sames as GETFIELD but for arrays
MONITORENTRY	Get a lock inside the JVM on the specified object AND get a lock in NAM in case a thread on another JVM is trying to edit this object at the same time
MONITOREXIT	Flush changes to the JVM's heap cache back to NAM in case another JVM is using the same objects as this JVM
INVOKE0	Stop a constructor from firing if it has fired elsewhere in the application cluster already. Instead page in the object graph from NAM that was flushed down to NAM by the previous JVM

Figure 3.1 bytecode instructions

3.1.4 How it Works

We depict the Figure3.2 on purpose to show how this works.

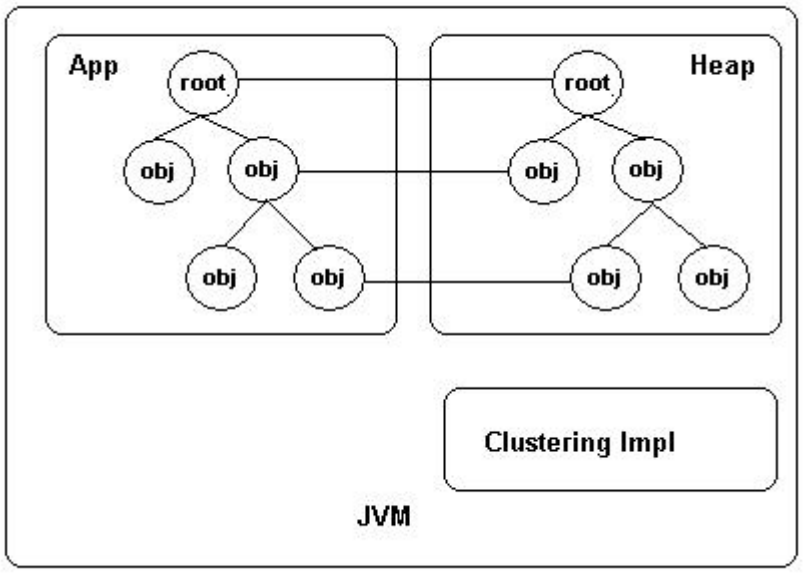

Figure 3.2 object graph in heap

The figure above shows that we have an application tragged App, and this application has a notion of object-graph. We could see there is a root [7] object on the top, which has a bunch of objects inside it that we want to cluster and make it available to multiple JVMs.

In this particular case, that 4 objects hanging off the root, we name them object1, object2 counting from left t right at the second level and object3 and object4 at the bottom. The lines between app and heap represent business logic and this business logic actually manipulates the heap.

We do assignment operations with the equal sign or we do the access operations with dot sign. In fact, we just get in that field and manipulate that field of the object. At runtime, it's updating the heap in that JVM above the ram. Terracotta basically works like to build a clustering implementation down between the application and the heap. It's illustrated by Figure3.3.

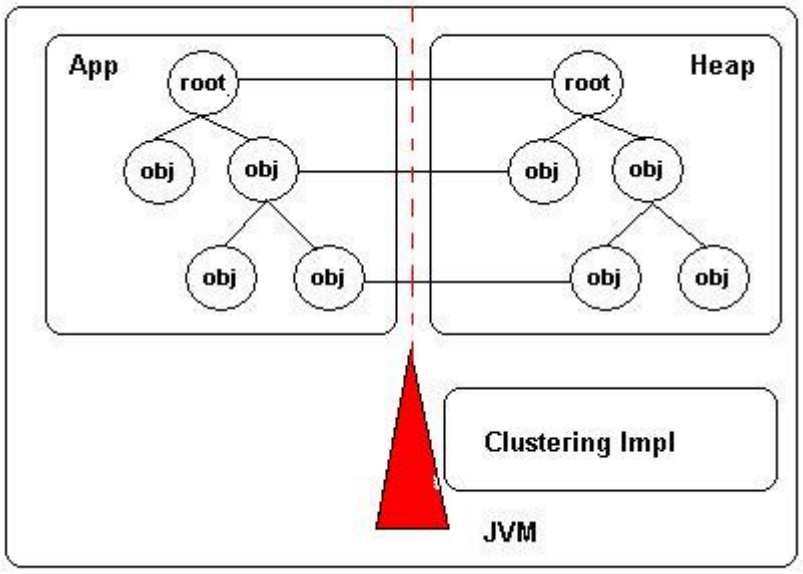

Figure 3.3 add Terracotta to the heap

Terracotta watches where our business logics trying to access the memory. It just inserts itself right in the middle as the red triangle. So it can figure out what's happening in the memory, which is touching the memory, and then replicate the changes to multiple JVMs.

Now, let's go forward. We want to update the object at the bottom of this object graph, the red one. Figure3.4

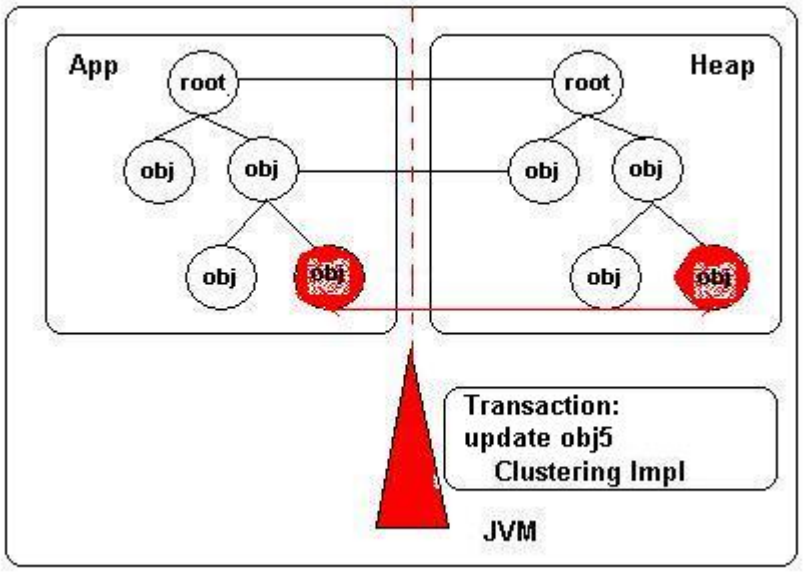

Figure 3.4 Terracotta intercepts the changes to the object graph in heap

The clustering implementation did something interesting. It started up the transaction and inside this transaction we changed object4. At the end of this transaction, we need to push it to other JVMs. Now, here come the problems: Are there any other JVMs which need to know we have changed the object4? How would we know that?

In order to replicate the object changes of this JVM over to the others, we need a central broker, namely, Terracotta server. We need someone to tell us which server references object4 in the cluster. Figure3.5

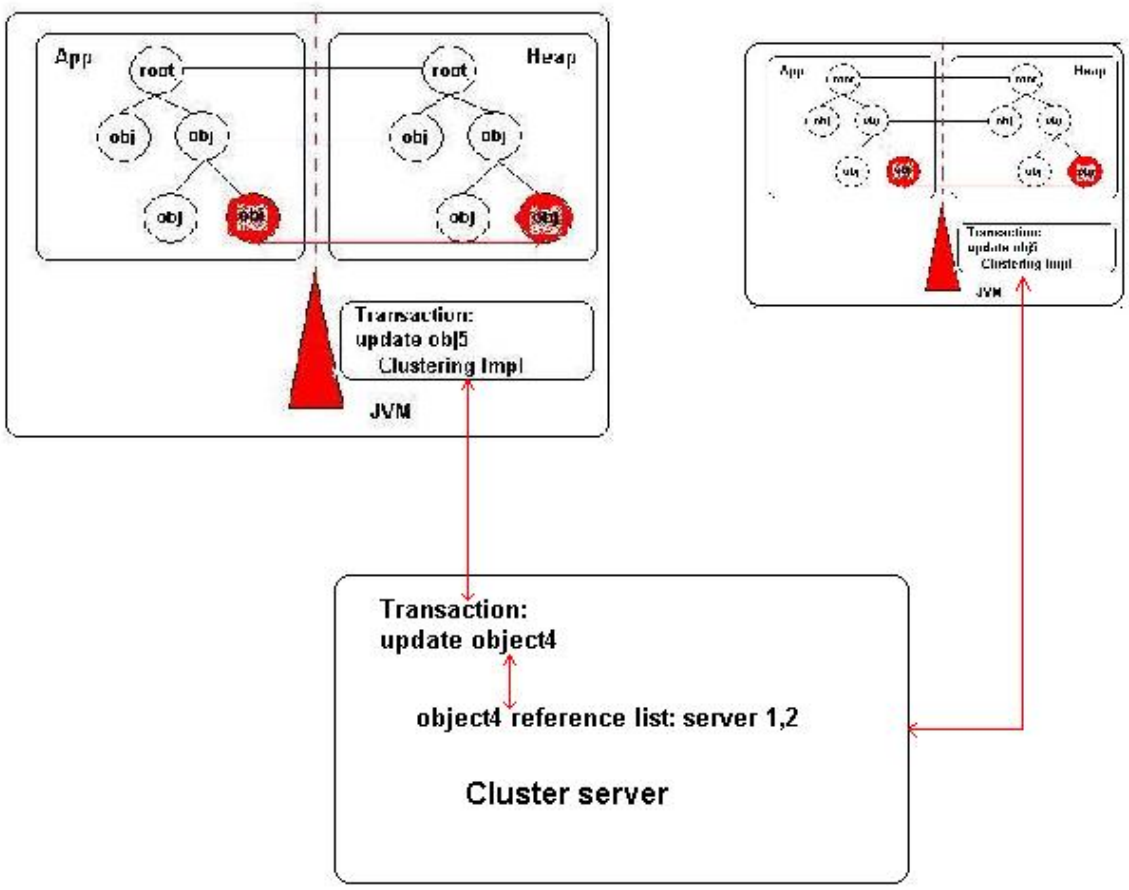

Figure 3.5 Terracotta transfers changes to other JVM

In this case, server1 and server2 both have a copy of the same root [7], both are actively consuming object4. So if server1 on the left changed its object4, server2 will find out and is server2 changed its object4, server1 will also know the change. The reason is the clustering server in the bottom knows both of them are referencing the object4 actively.

3.1.5 Fine-Grained Replication

So basically, what Terracotta does is inserting itself right between the application logics in the heap. So it doesn't have to understand why we are changing a field. It just have to know that's we are changing a field. And the way Terracotta knows the change is that equal sign turns into a putfield call at a bytecode level and the dot sign turns into a getfield call at a bytecode level. So the Terracotta just has to know who is calling putfield, who's calling getfield, what the arguments are. Then it can replicate those changes to other machines.

Since Terracotta disassembles objects into the field's level at class loading time, only changed fields need to be replicated and sent over the wire and more important is these changes only go the server are referencing them actively. As Figure2.2 shows, server2 only stores parts of the objects, because it only needs those for that moment. Faulting in from cluster server only

happens when application server really needs the objects. As server1 and server3, both they have the same object-graph view.

So fine-grained replication means less data move around the network to other nodes. This means we don't need sending around the entire object anymore if there is only tiny change happened, which leads to higher scalability, lower latency and higher throughput.

3.1.6 Thread Coordination

In one single JVM, the memory model works as a cop to specify how memory actions when multithreaded access to shared memory. In a clustered environment, Terracotta extends the multithreaded mechanism across the cluster. Demonstrated by above, we have already seen how Terracotta preserves the semantics of the JVM. Here we will discuss clustering locks in Terracotta.

As we mentioned in section 3.13 ASM, Terracotta detects bytecode instructions like **MONITORENTRY and MONITOREXIT** in synchronized blocks. By using AOP, join point matches synchronized blocks, lock and unlock pointcuts respectively match monitor_enter and monitor_exit. Figure3.6

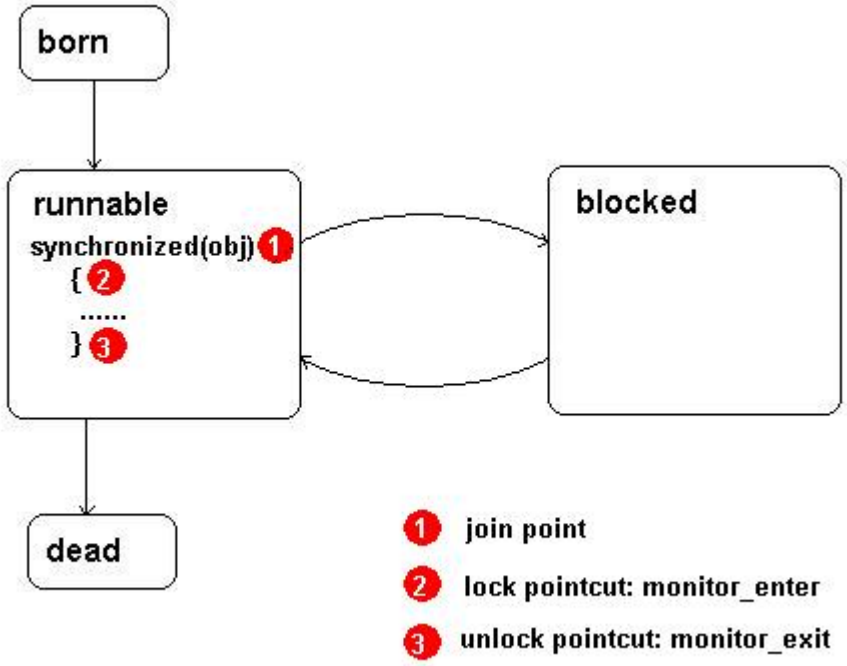

Figure 3.6 java synchronization mechanism

Here are the two pointcuts that pick out all the critical sections (entry and exit) where a clusterable instance is being locked on (and most likely a shared instance is being modified) [9]:

private pointcut clusteredMonitorEnter(Object o):

isClustered() &&lock() &&

```
@args(Clustered) &&
args(o);

private pointcut clusteredMonitorExit(Object o) :
isClustered() &&unlock() &&
@args(Clustered) &&
args(o);
```

Then we can write a before and after advice that will delegate to the monitorEnter() and monitorExit() methods in the ClusterManager[9]:

```
before(Object o) : clusteredMonitorEnter(o) {
ClusterManager.monitorEnter(o);
}

after(Object o) : clusteredMonitorExit(o) {
ClusterManager.monitorExit(o);
}
```

These calls will be delegated to the lock manager in the coordinator, which will use the object identity of the pointcut argument instance to acquire (or release) a distributed cluster wide lock on the instance [9].

Basically, a synchronized request will lead to lock the object on the both application server and cluster server, which will guarantee that monitor enter and monitor exit are synchronous. Figure3.7:

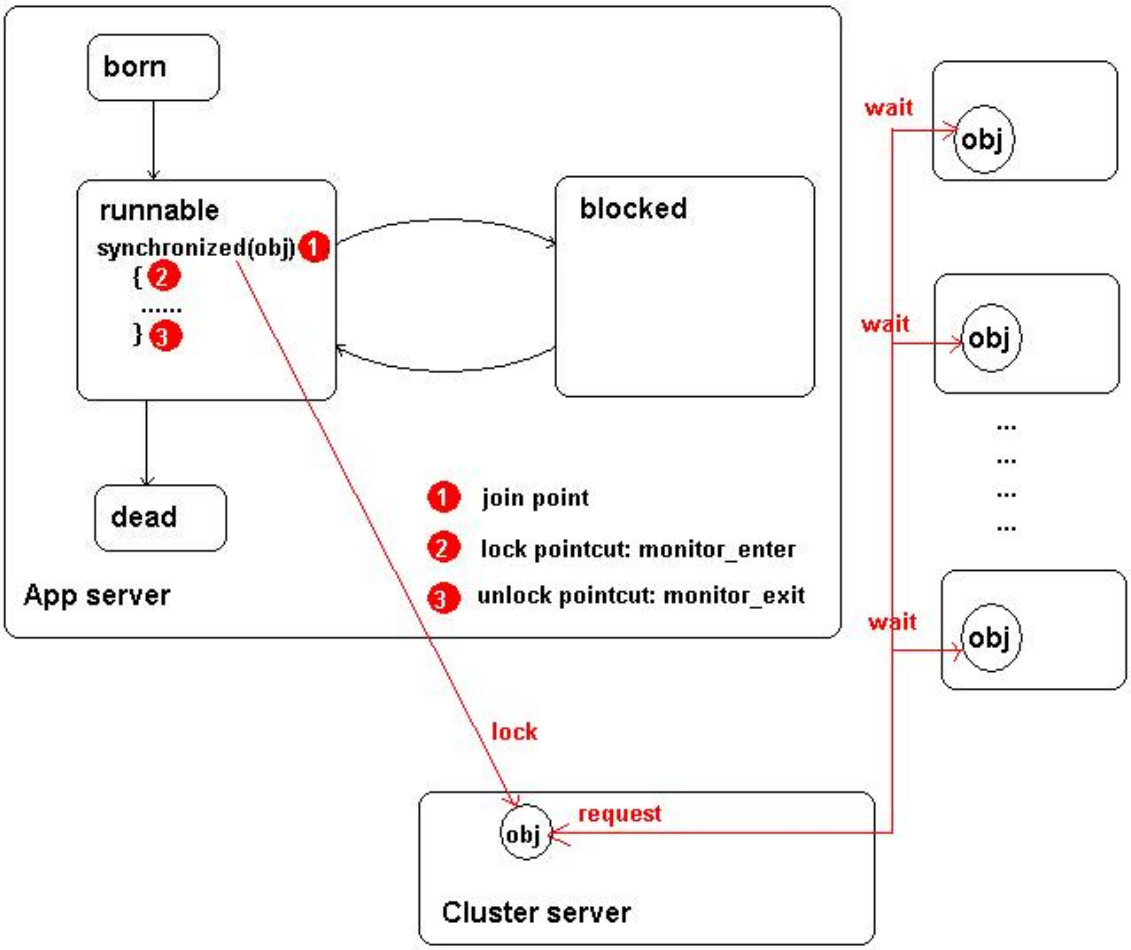

Figure 3.7 Terracotta synchronization mechanism

3.2 Hub and Spoke

Hub and spoke is topology that conceptually refers to a bicycle wheel. From the Figure2.2, Terracotta is using hub-and-spoke architecture. In Terracotta world we call Terracotta client (application server) as L1, Terracotta server as L2. This means it has one L2 server and n L1 application servers. Figure3.8 [10]:

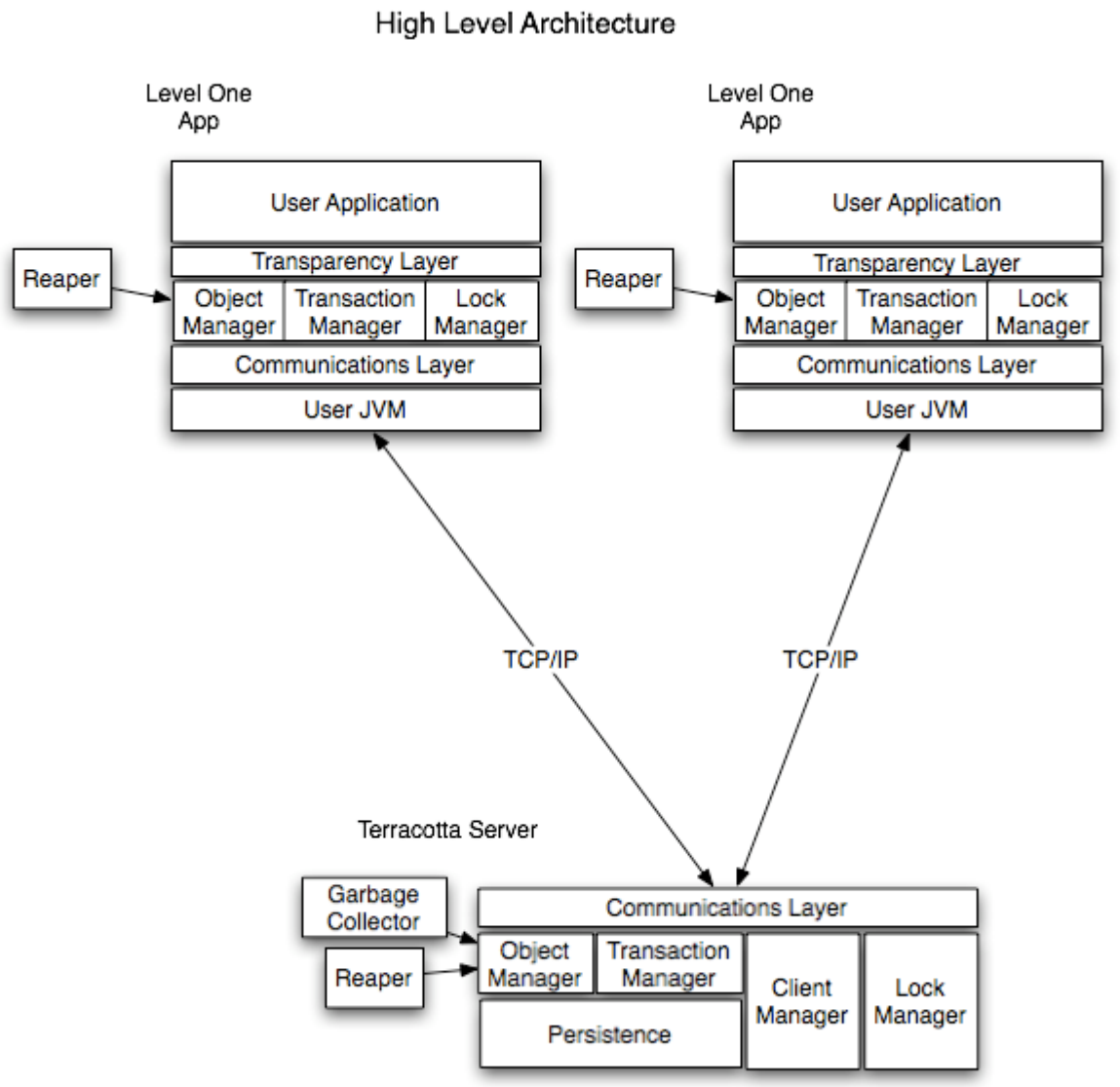

Figure 3.8 architecture overview of Terracotta

As you can see at the bottom of Figure, the clustering server is the core of this architecture. It works like a coordinator across the entire clustering environment and uses lock manager to take care of the clustering lock, like: which thread in which node accesses the critical section (synchronized), which nodes are referencing the shared objects actively and for the shared objects which are not used frequently, they will be paged out to the disk.

More interesting, it servers as a JMV level database, which store all the shared state, scratch data in the persistence as we have already mentioned.

Here comes some common confusion about this architecture as we presented in the following.

3.2.1 Cluster Hub to Avoid SPOF

In fact, Terracotta server can work in an active-passive mode to avoid single point of failure. Basically, in most common situations, there is one Terracotta server running in active mode, and other Terracotta servers act as a hot standby in passive mode in case of failures. For now,

there are ways to cluster Terracotta servers. One is using shared disk, and the second is over network. For more details, go to Configuring a Terracotta Server Cluster [11].

3.2.2 Batched Field-Level Changes

As we have mentioned above Terracotta can propagate fine-grained changes. However, fine grained change replication also means that our application JVM has to communicate with Terracotta server each time when there is a change occurs, right? And if so, the network will become overload.

Actually, what is really happening between application servers and clustering server when we replicate the changes is not like we naively assumed. In short, Terracotta extends the semantics of the locking of java language; it can recognize a synchronized block and take it as a boundary of a transaction. We refer Terracotta transactions as sets of changes that must be applied atomically. So this means that if we changed 3 different fields of an object one after the other, Terracotta distributes one batch with the 3 updates instead committing 3 updates respectively as Figure3.9:

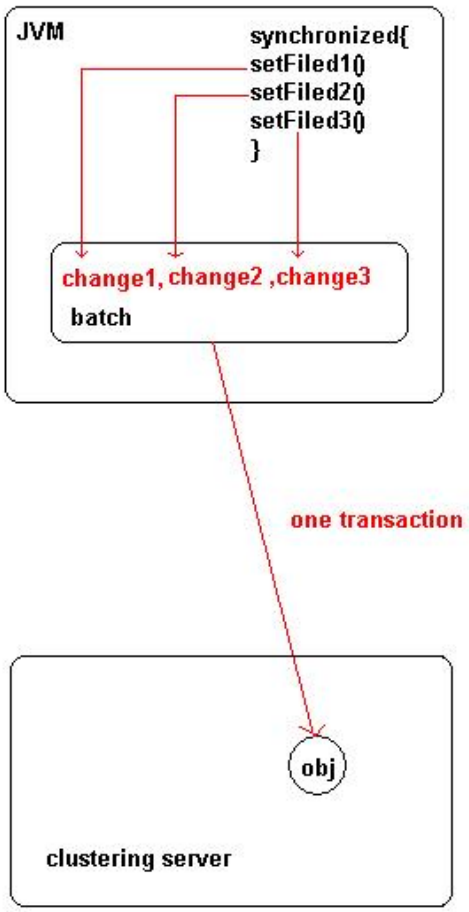

Figure 3.9 batching changes to Terracotta server

So batching up the changes in the local JVM and then sending these to the clustering server once avoid frequently chatting through wires, which also means that higher efficient and lower network overload.

Chapter 4 Scenarios & Discussion

The previous chapter described the theory behind Terracotta JVM level clustering technique, now we move on to the design of different of scenarios; test how Terracotta provides high availability and scalability, and how it will impact the way of our programming and where to use it.

4.1 Pitfall Overview

Multi-tier is a brilliant architectural design pattern in the development of software. The separation of concerns in the code makes the developers can focus on what they really care and parallel the deployment. Struts, Spring, and Hibernate, these open sources are all focus on speeding efficiency of software development or relieving developer' coding burden. They all make software look elegant; however, they also have a significant problem when it comes that applications require scaled out in response to increasing capacity demands. We will discuss these problems as the following.

4.1.1 Loading Problem

This is the most common problem: inefficient read from database.

4.1.1.1 Naïve Loading Problem

As we have discussed previously, nowadays, database has been abused. Figure4.1 illustrates a basic use of database for clustering.

1. Low efficiency in clustering

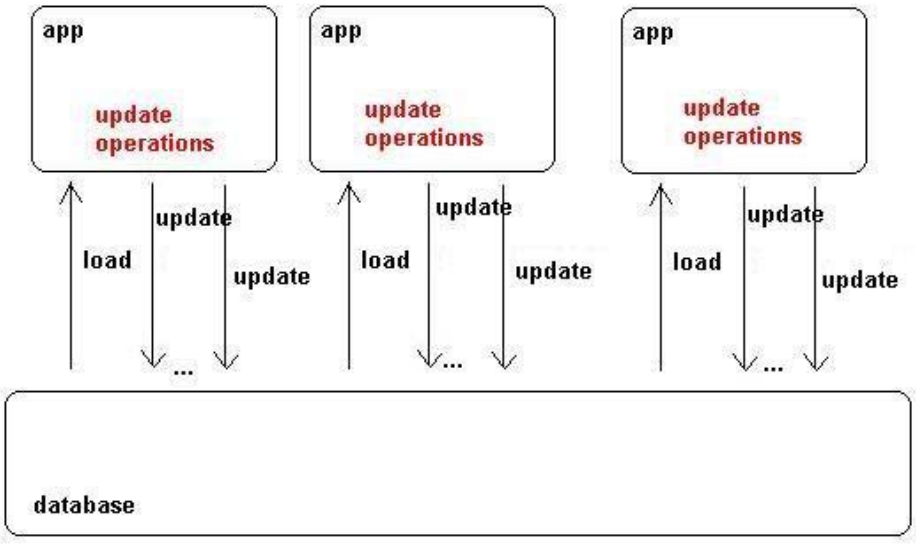

Figure 4.1 naïve loading database solution in distributed environment

Since we don't have an easy-use, reliable mechanism for application state objects clustering, we have to load the objects that we need from database, and commit the changes back to it after each update finished to keep data persistent. In this picture, we clustered three

applications and here comes the problem: each of them has to individually communicate with the database for loading even they are operating the same objects.

2. Low efficiency even in one app

Worse, even we operated with the previous object; we still need to load it again because we closed last database connection. Illustrated by Figure4.2

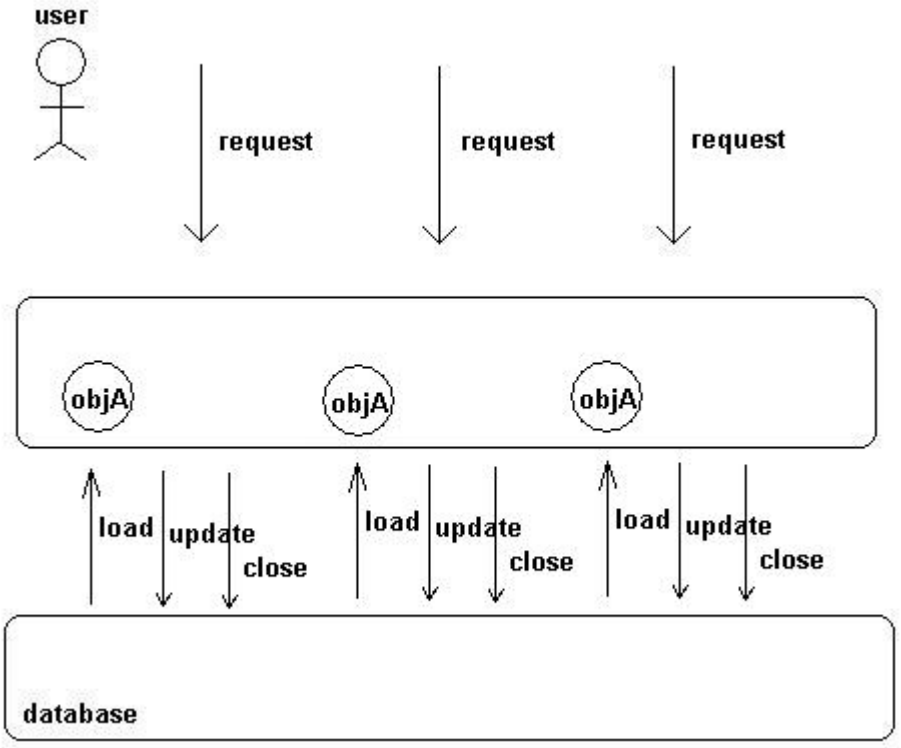

Figure 4.2 naïve loading database solution in single JVM

The real problem is that we don't have application layer's shared states which leads we have to do these unnecessary trips to the database.

4.1.1.2 Loading problem with Cache Solution

Actually, in the real market, as we mentioned in Chapter 1, various sorts of caching products help us solve the above problem.

1. Limitation of the memory
To solve the problem of Figure4.2, we add a cache layer between our application server and database. Figure4.3

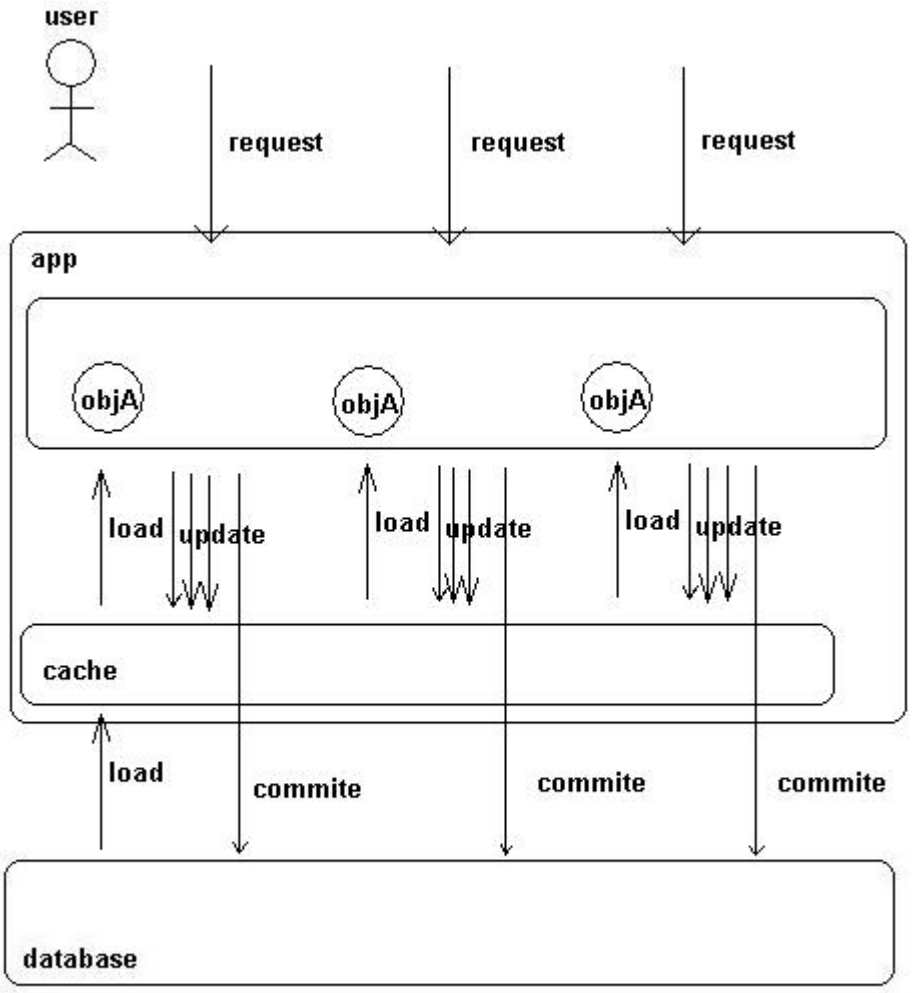

Figure 4.3 add cache as objects shared layer

This all works fine, except only one pitfall, namely, our application layer cache is an in-memory cache in this case. This solution requires enough physical memory to keep the data in the cache; otherwise some of the cache has to be evicted out from cache according certain expiration mechanism like LRU, LFU, etc [12]. If there is no enough memory space for the entire cache, we still need to start to unnecessary trips to the database.

2. Still Overload on the Database as More Nodes Added In

To solve the problem of Figure4.1, we do the same as above by adding a cache layer between application servers and database. Illustrated by Figure.4.4

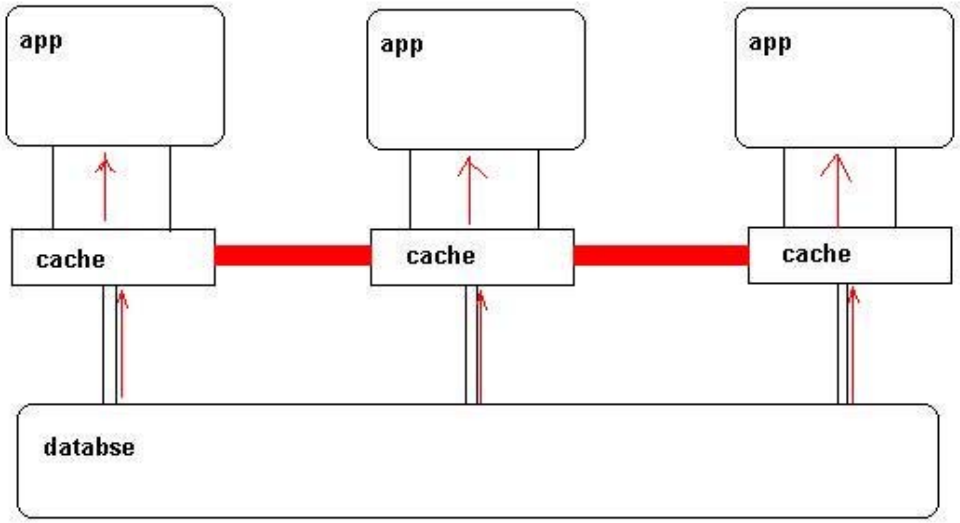

Figure 4.4 add cache layer in the distributed environment

For now, our application servers don't need to get the objects from database for each access. When the objects first have been loaded, the cache layer stores them in for later use. However, each application cache still has to load/maintain the objects from database separately. This means increasing application caches also burden the database regardless of what has been loaded in other application caches. So this solution, we could imagine, would work well under a few nodes environment, but the naïve loading problem will become obvious as adding more nodes.

4.1.2 Replication Problem

When changes occur in the clustered environment, it comes more complicated and inefficient problems. Basically, each application server works on its own cache, which means that the other caches will have no idea when there are changes occurred in one cache. Take a web based application as an example illustrated by Figure4. Since there is no way to inform the other two application caches that object A has already been updated and both of them have the object A in their cache, so they just get it from their local cache instead checking out from the database and have no idea that the object is stale.

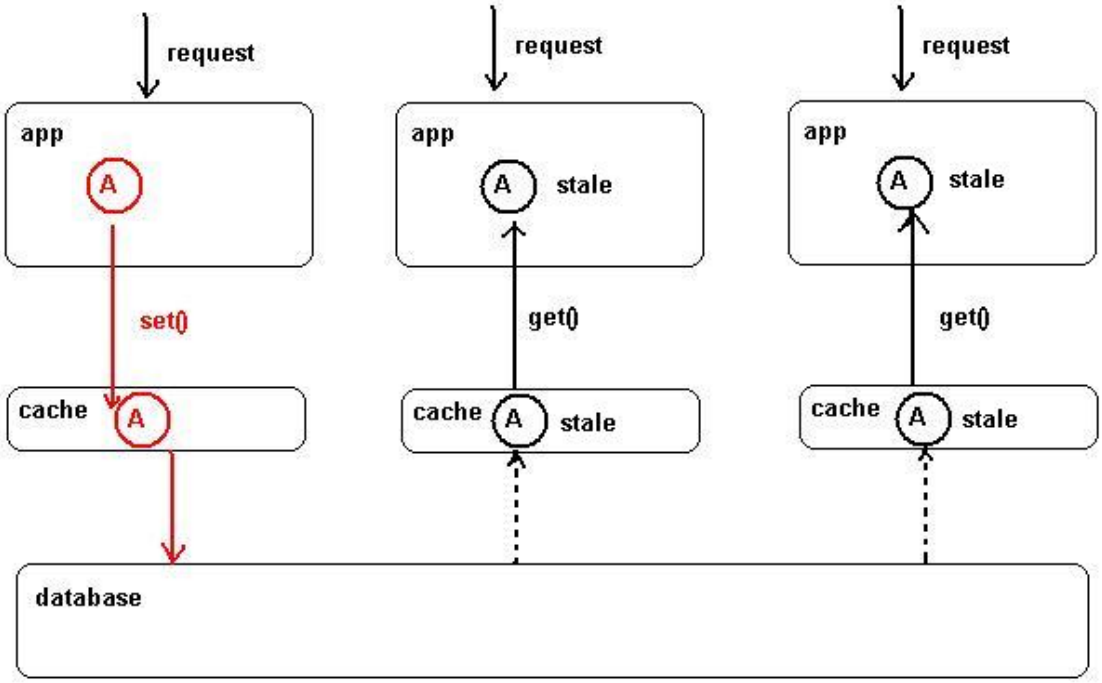

Figure 4.5 add cache layer leads object stale problem

Normally we have two solutions to solve this problem:

1. Sticky Session Solution [14]

It means not that the sessions are sticky. In simple words, sticky session solution is that we force our loadbalancer transferring the requests from the same user to the same application server. Figure4.

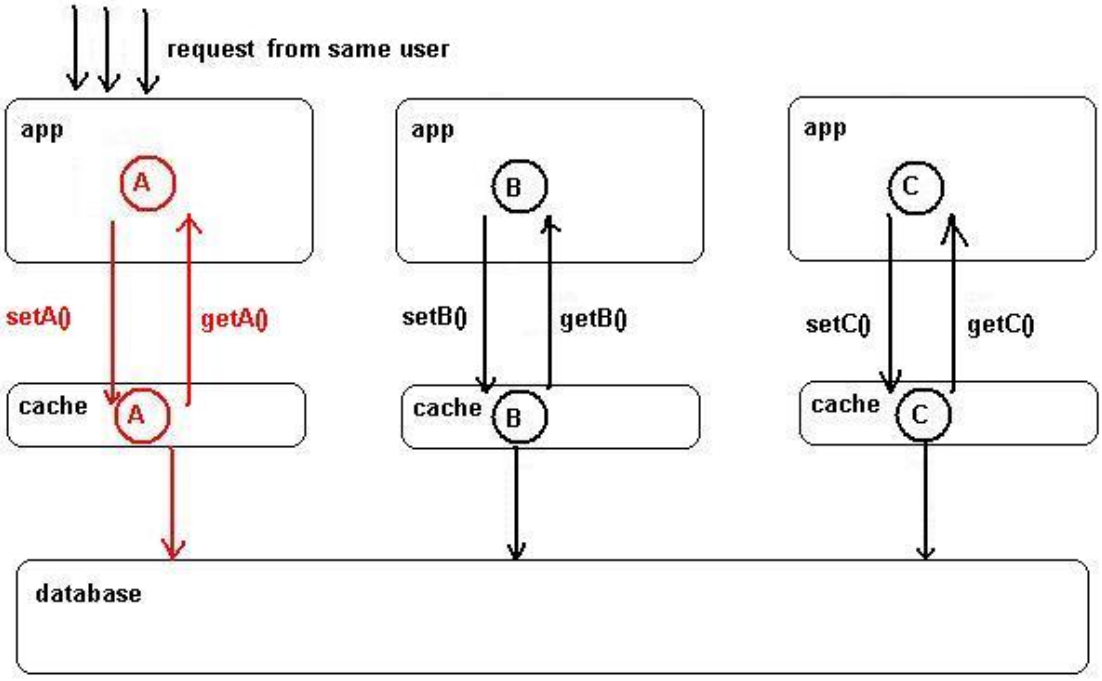

Figure 4.6 sticky session solution

Now, we have all the servers have no stale object state.

2. Distributed Cache Solution
Another commonly used solution is distributed cache, which means to replicate data between the caches to get them consistent. As we have mentioned in chapter 1, to maintain cache consistent we have to so all those unnatural programming like put, update and remove and we also need some sort of notification mechanism. Basically, there are two different ways to fix this problem.

- ✧ **Copy between clustered caches:**
  Cache server will serialize the object and transmit it to other cache servers in the cluster.
- ✧ **Invalidate the data, other cache servers load it from database**
  Cache server will be forced to load the data from database to refresh its cache.

4.1.2.1 Problem of Sticky Session

The problem with sticky session is quite obvious: it doesn't solve our scalability problem radically. The load balancer with sticky session mechanism has to send all the requests to their original server, even that server is heavily loaded and there is another server is idle.

4.1.2.2 Problems of Distributed Cache

Copy mechanism illustrated by Figure4. (Red circle represents fresh object, black circle is stale.)

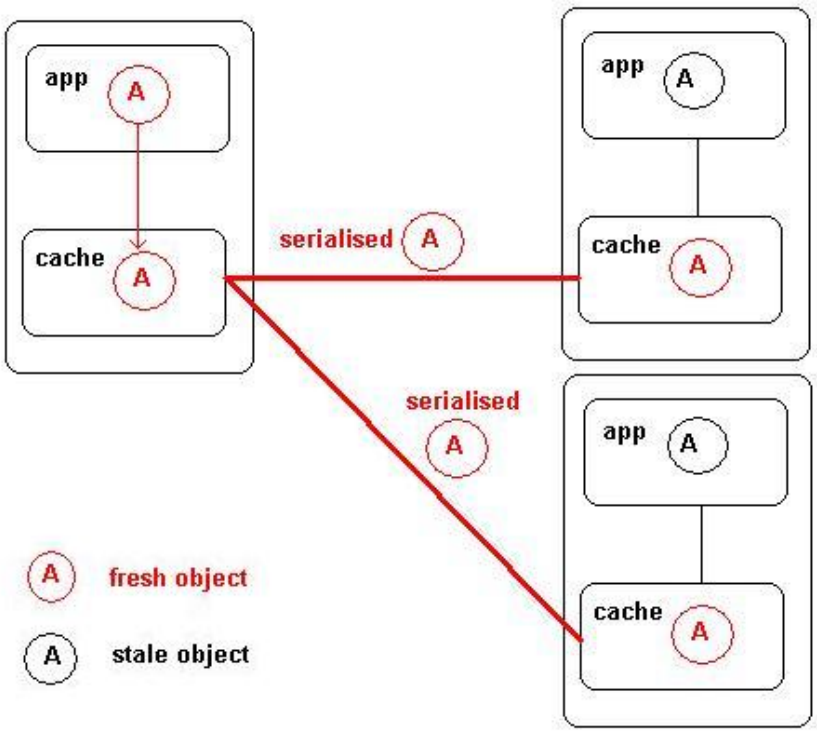

Figure 4.7 network overhead problem based on copy mechanism of distributed cache

As discussed in chapter I, the cache service doesn't understand our business objects, and then we must break them into bytes using serialization and assemble them back, which could cause a serious network load problem as more nodes added in.

Invalidate mechanism illustrated by Figure4.8

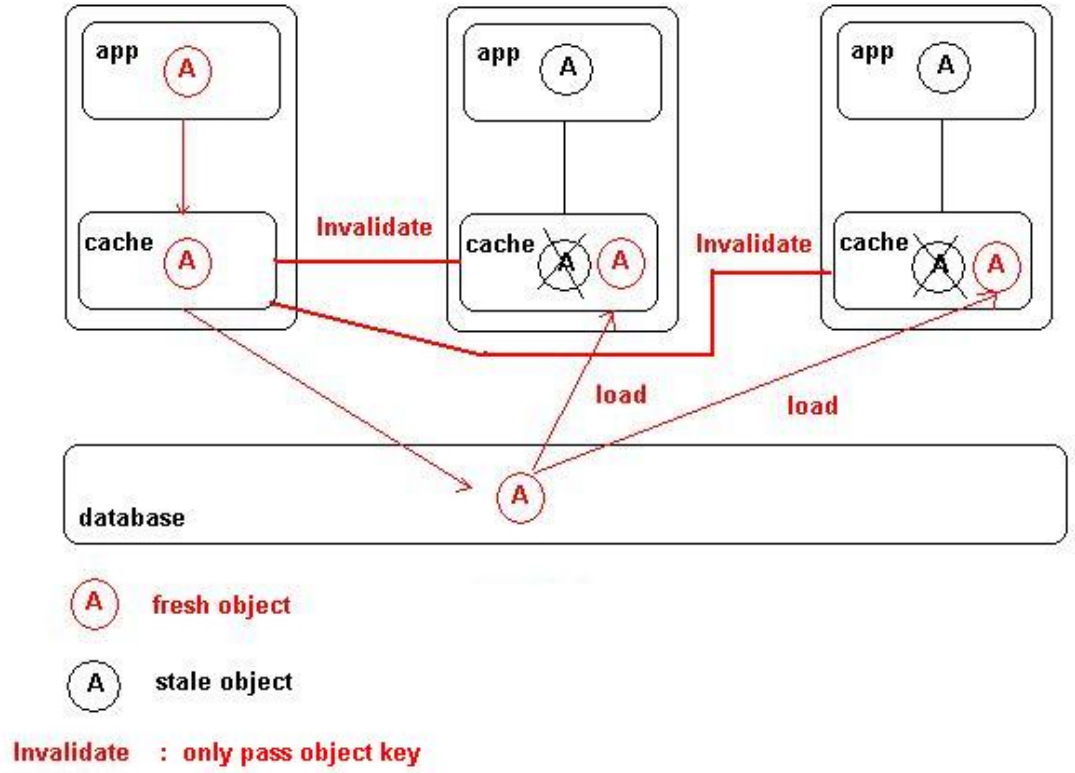

Figure 4.8 database overhead problem based on invalidate mechanism of distributed cache

Using invalidation mechanism (LRU, etc), we could effectively avoid net work problem, because only the object key needs to be transmitted over the network. However, this leads database load problem as mentioned in section 4.1.1. Server 2 and server 3 have to refresh the data from the database.

4.1.2.3 Local JVM Problem of Distributed Cache

Distributed cache layer will take care of all the notification between caches. However, there is another problem about distributed cache, which needs us to fix it by ourselves. After cache servers finish distributed updates, we have our cache data coherent. But we should notice that server 2 and server 3 both have the stale object A in their local heap, which could be previously got from cache. So we have to utilize some sort of callbacks mechanism to register our app, get informed when updates occur. This will result in complex architecture of our code, which turns out hard to maintain. Again it is unnatural. Figure4.9

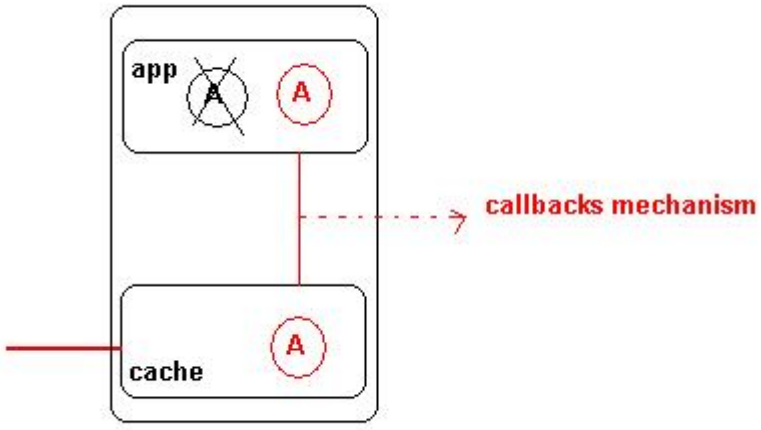

Figure 4.9 local JVM Problem of distributed cache

4.1.2.4 N-1 Notification Problem of Distributed Cache

No matter copy or invalidate mechanism that we use, we all need some kind of notification mechanism of cache mechanism. Even cache server will take care automatically for us or only object key will be sent across net when clustering caches work under invalidate mode, we will still got n-1 notification problem. N-1 notification means that n-1 notifications have to be sent to other nodes in the cluster as one cache instance change occurs. This could potentially generate a large amount of network traffic.

4.2 Terracotta Scenario and discussion

We use this section to present our test results based on Terracotta, and we will see how Terracotta solves the problems as mentioned in section 4.1.

4.2.1 Scenario1 – fix separately loading problem

Now we will plug Terracotta into the application and see how it fixes this problem of clustering with cache as presented in 4.1.1.2. For the simplicity, we hardcode the code and simulate database as a map, all the loading operation will get corresponding business objects from this map. And we use Apache CXF to monitor the visit times to our database (which is in case simulated by a map). We use hashmap as our cache and store the object into the hashmap for later access. We simulate three application servers and three individual caches just like as we mentioned in section 4.1.

Since Terracotta can seamlessly cluster our applications, first let’s look at this test without Terracotta plug-in. Illustrated by Figure4.10

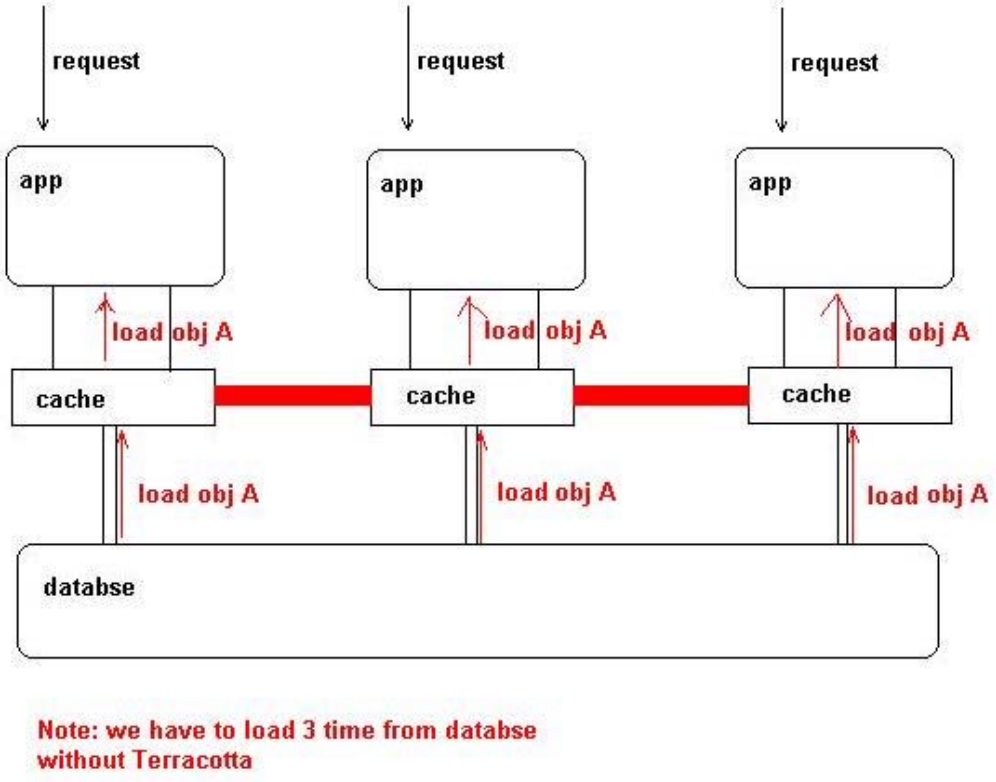

Figure 4.10 distributed cache solution without Terracotta

In the above figure, we have to load three times if we want to get object A from three application servers respectively. The following is the result from eclipse console:

```
Server [Java Application] D:\MyEclipse 6.0\jre\bin\javaw.exe (Apr 3, 2008 10:52:14 PM)
2008-4-3 22:52:17 sun.reflect.NativeMethodAccessorImpl invoke0
信息: Started SelectChannelConnector@0.0.0.0:9000
Server is ready...
Call count = 1
Call count = 2
Call count = 3
```

Figure 4.11 loading times from database based on distributed cache solution without Terracotta

And the lapsed time for this **getObject()** call is showed as following Figure4.7. Here we paste 8 times' of calling this method. We could see each application server loads the object separately from the database and they warm the cache when this **getObject()** is first called. After the object has been loaded into the cache, the accessing time is dramatically reducing.

Visit Sequence	App Server1 Lapsed Time	App Server2 Lapsed Time	App Server3 Lapsed Time
First	3033.801782 millis	3126.375838 millis	2934.970785 millis
2	0.025422 millis	0.025981 millis	0.212877 millis
3	0.024863 millis	0.027099 millis	0.029892 millis

4	0.024304 millis	0.025422 millis	0.022349 millis
5	0.025701 millis	0.027657 millis	0.023746 millis
6	0.024305 millis	0.024305 millis	0.022908 millis
7	0.025422 millis	0.024863 millis	0.023746 millis
8	0.024863 millis	0.024864 millis	0.029334 millis

Figure 4.12 accessing time based on distributed cache solution without Terracotta

Now we plug Terracotta to our application servers, and it will work like Figure4.8

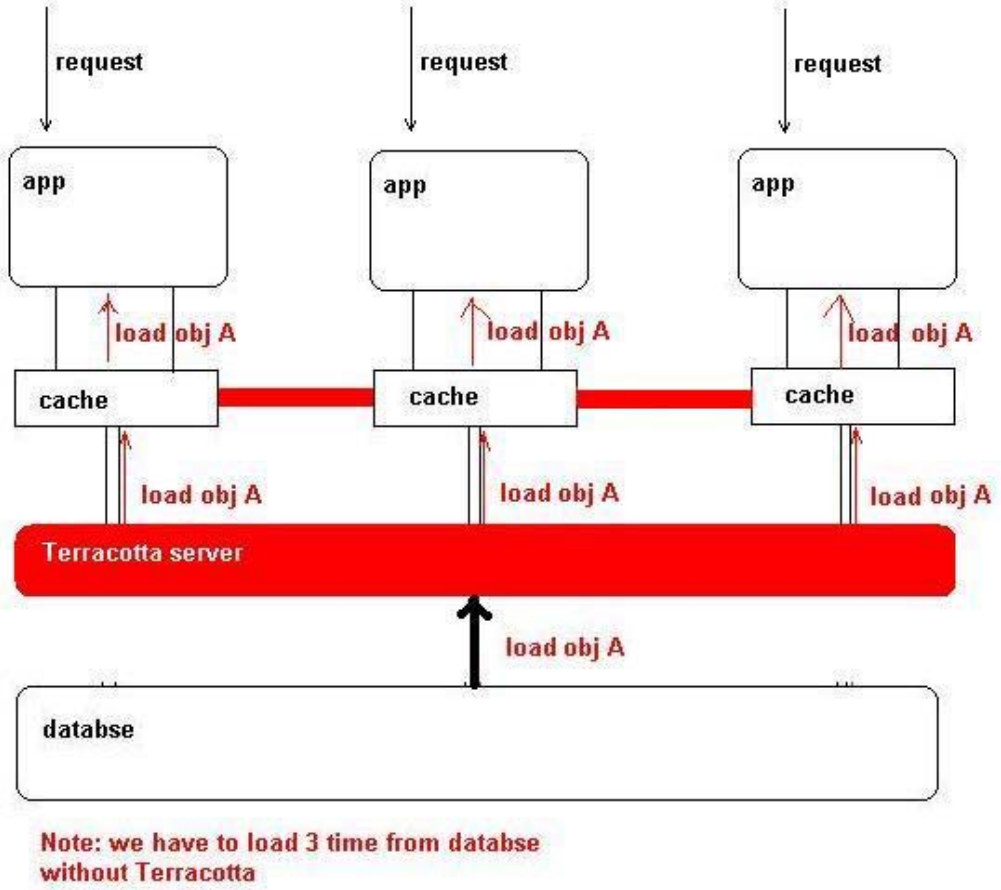

Figure 4.13 distributed cache solution with Terracotta

Terracotta provides a single, consistent cache to solve this overload problem as the above red bar. Except the first time accessing we need to load the object directly from database, the other nodes which need to get the object will not have to warm their cache directly from the database. Instead, they will automatically load it from the Terracotta Server. The following is the result from eclipse console:

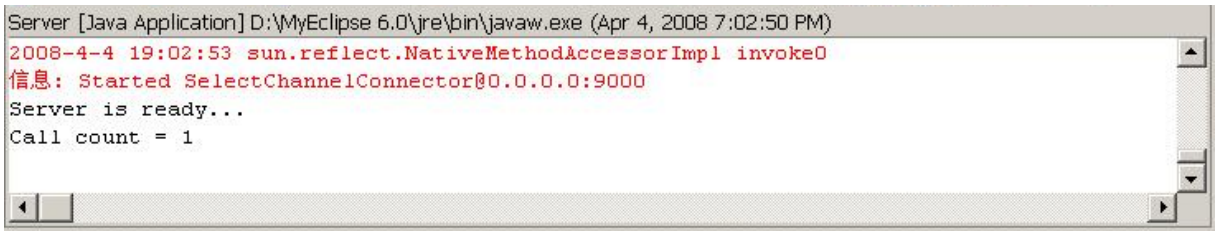

Figure 4.14 loading times from database based on distributed cache solution with Terracotta

We also present the lapsed time for this **getObject()** as following Figure4.15. Here we do the same thing of pasting 8 times' of calling this method. After the object has been loaded into the cache, we still get the accessing time is dramatically reducing. But we would notice that the loading time of server2 and server3 is much less. This is because that Terracotta notices that our hashmap as a cache layer have been clustered, and when we ask for that object it just faulted in (Figure4.16) and took approximate 4 milliseconds. We don't need reload the data from database. And now if we refresh this, the data is already local to its own particular JVM, so we got our warm cache for each server.

Visit Sequence	App Server1 Lapsed Time	App Server2 Lapsed Time	App Server3 Lapsed Time
First	5247.586215 millis	4.312559 millis	4.48353 millis
2	0.177117 millis	0.181587 millis	0.184381 millis
3	0.181587 millis	0.188851 millis	0.179911 millis
4	0.180191 millis	0.184381 millis	0.182705 millis
5	0.20254 millis	0.179353 millis	0.179073 millis
6	0.186336 millis	0.197511 millis	0.18494 millis
7	0.251149 millis	0.201142 millis	0.241651 millis
8	0.189689 millis	0.173206 millis	0.182985 millis

Figure 4.15 accessing time based on distributed cache solution with Terracotta

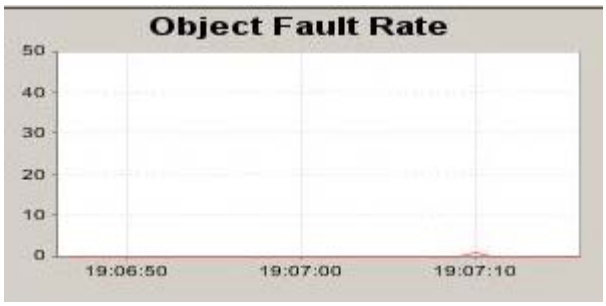

Figure 4.16 object fault in Terracotta

And as more nodes are added for scale-out, the loading amount from database is still one. So we get higher number of application servers and lower load of database.

4.2.2 Scenario2 – fix limitation of physical memory

In this scenario, we do the same as the first scenario; the only different is we use Ehcache. As we have mentioned above, in our cache solution we have the constraints of available memory. We need enough memory to maintain our active data, otherwise even some data were still valid, and we still have to abandon them due to limited memory space. This leads undesirable effect to load them again from database.

Terracotta solves this problem by using its virtual heap so our JVM can spill data out into the Terracotta server. The basic workflow is first client JVMs send data to the cluster server, then Terracotta cluster, in turn, can store data in the disk.

In our code, we set objects live time 3000 milliseconds to simulate object eviction. Figure4.17 and Figure4.18 illustrate that our application server1 load the object from Terracotta server. And then after 3000 milliseconds, application server2 and server3 request the same object. Here we could see all the servers load the object from Terracotta server instead from database.

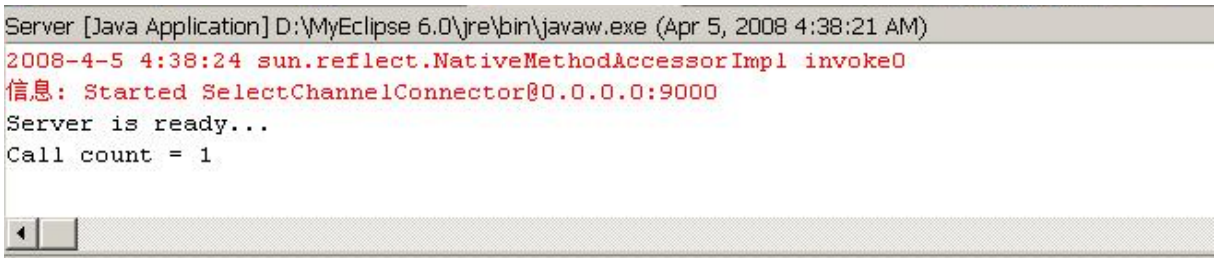

Figure 4.17 load object from Terracotta server instead database

We only load from database once.

Load	App Server1 Lapsed Time	App Server2 Lapsed Time	App Server3 Lapsed Time
First	4909.708915 millis		
After 3 seconds	13.140777 millis	13.430198 millis	11.496992 millis
After 3 seconds	0.178235 millis	0.184381 millis	0.296686 millis
After 3 seconds	0.24612 millis	0.181588 millis	0.23299 millis
After 3 seconds	0.185499 millis	0.184939 millis	0.184381 millis

Figure 4.18 accessing time of loading object from Terracotta server after 3 seconds

After we load the object from database and store the object in the Terracotta server, whenever we need, we get it from the cluster server and put it into our cache. Then we have the warm cache in local. The fault-in process works just like Figure4.10.

4.2.3 Scenario3 – fix n-1 ACK problem

In this scenario, we will see how Terracotta eliminate n-1 problem as we mentioned in section 4.1.2.4. Actually, we just write a simple application running in one JVM and we cluster it using Terracotta. It is illustrated by Figure4.19.

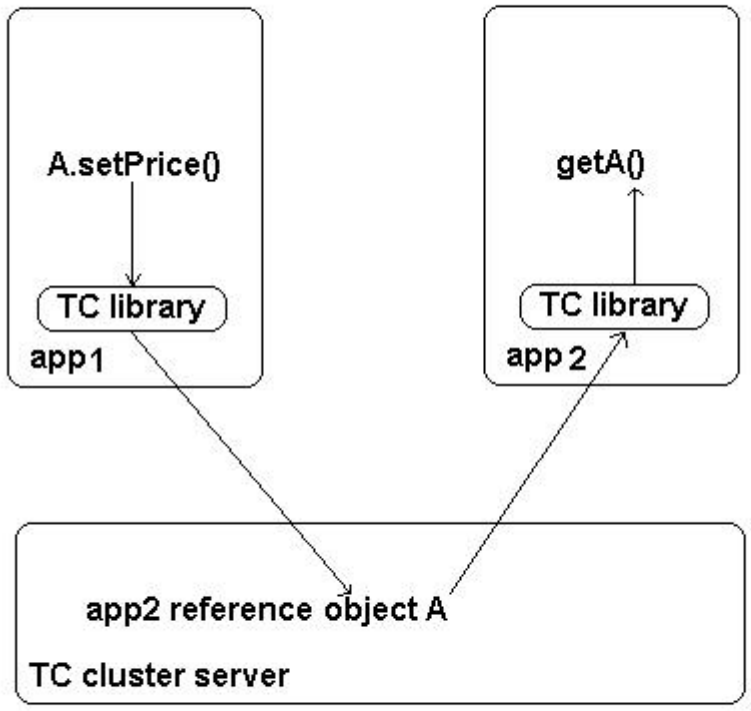

Figure 4.19 fix n-1 ACK problem with Terracotta

We have two application servers, 1 and 2. We have Terracotta server. In application server 1, we change object A, then it talks to Terracotta server to notify the object has changed. Client reads object A on application server2. The interesting part is that we don't implement any types of callbacks mechanism to inform app2 that object A has changed; app2 will still get the fresh new object.

The reason is, as we talked before, Terracotta is a 2-tier hub architecture and clustering server maintains essentially has a map of which reference is living on which application server. With this client-server architecture, we avoid n-1 ACK problem, the clustering server is the only one needed to know the changes and with the map in clustering server the changes are pro-actively pushed to the application server2.

4.2.4 Scenario4 – where to use Terracotta

In this scenario, we will do a deep dive analyzing how to design a good application, how to use Terracotta to implement a good clustering application.

We can start by looking at a simple baseline JDBC application, which is where we usually start at if we were just doing simple operations against the database. Next step would be to add hibernate. And next if we want to increase our performance, this is done by enable the

second level cache, using something like Ehcache, OScache, Jboss Tree cache underneath hibernate. And finally there is a last model which is recommended by the hibernate documentation, which is to detach the pojos entirely from the application and reattach them later after updates have been made. We will analyze the pros and cons of each of these design patterns and where to use Terracotta to provide the best performance in clustering case

So the first level is our baseline JDBC application. We like to illustrate a typical application as Figure4.20. This typical application has a user request flow, a conversation, which is happening between the use and the application. We just picked up a random number: there are 3 user requests during this conversation, 2 pojo-updates per request.

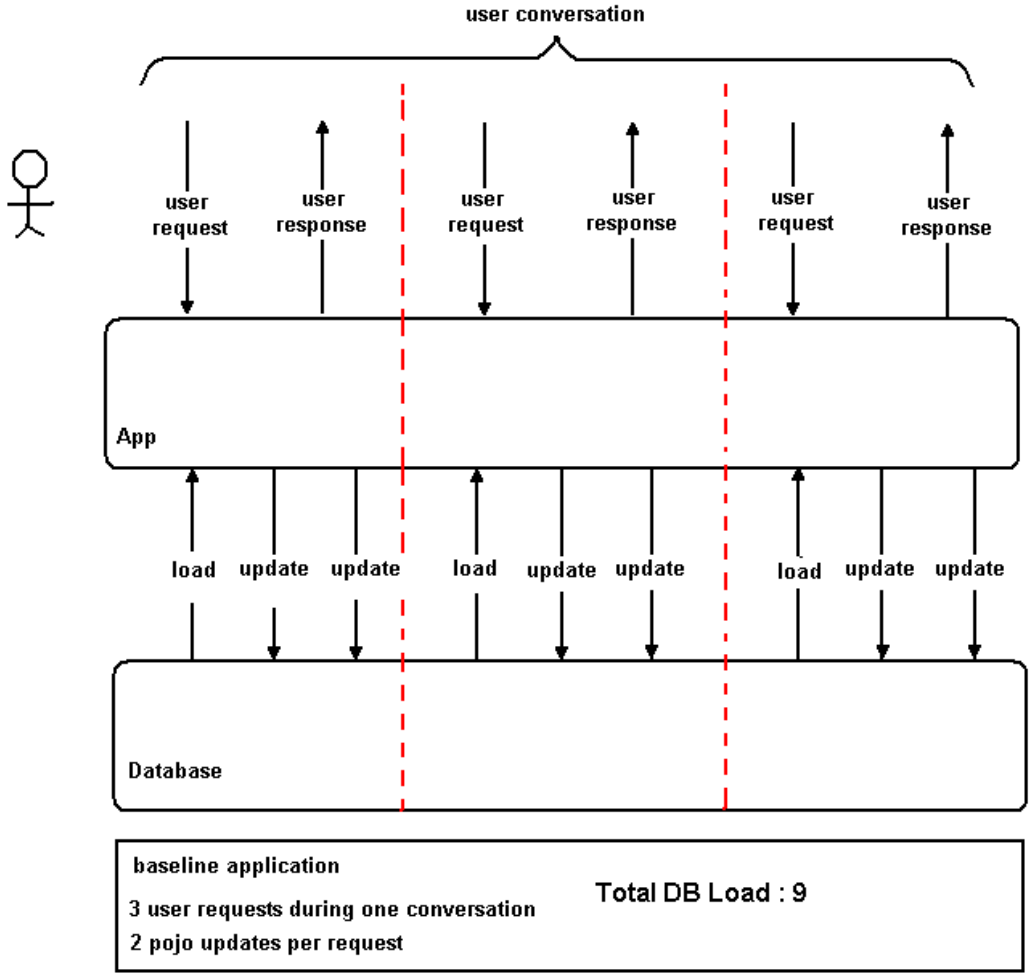

Figure 4.20 baseline application

Here we can see that each operation, it will be directly operating against the database. We count from left to right, there are 9 totally operations to the database.

The next step is to go ahead and add hibernate. This is a great option, because adding hibernate as flexibility to this application, so we have the ability to model our relational data using objects, using pojos. It gives us a very nice model for being able to write all of our operations to the database using the object model.

The upside of this solution is that now we talk in terms of beginning a transaction and ending a transaction. This is a typical hibernate best practice: we open a session, begin a transaction,

modify or create pojos in that session, then we will write them into the database. During the time in which I modify those pojos, we are going to be writing directly to the heap store that the pojos represent illustrated by Figure4.21

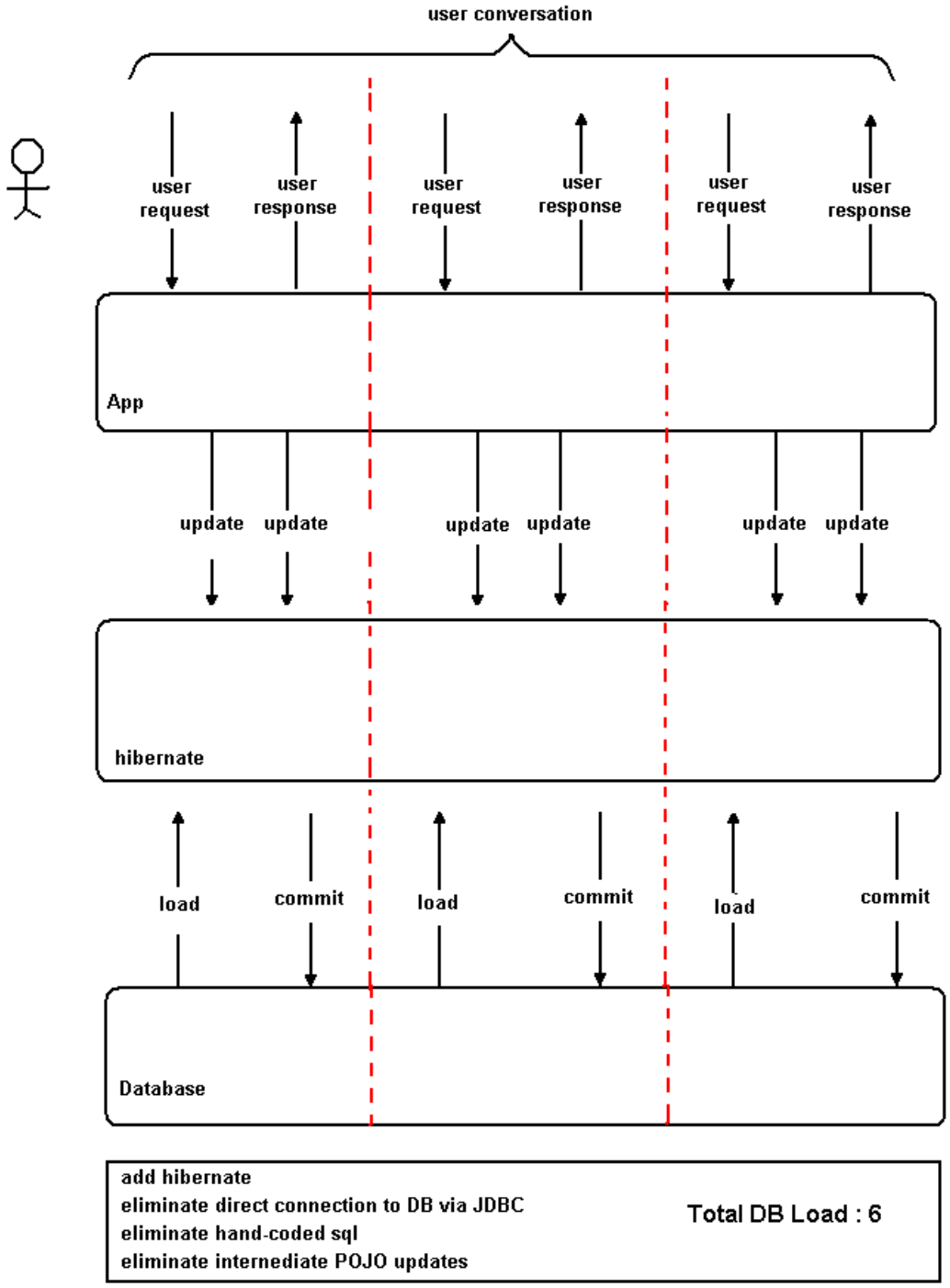

Figure 4.21 add hibernate

So now we no longer write those updates to the database, we actually writing those updates directly to the pojo store, which is the level that is depicted by hibernate.

The difference here is that now our application instead of writing those updates to the database is writing the updates into the hibernate pojo store which is batching those updates up in memory and only committing them when we tell hibernate to commit the entire transaction.

So what this means is like if we count each of these transactions. Now we see instead of 9 operations to the database, now we have 6 total transactions to our database. So we have increased our flexibility and decreased our database load.

The next step is the third level. This level is to enable second level cache, which is illustrated by Figure4.22

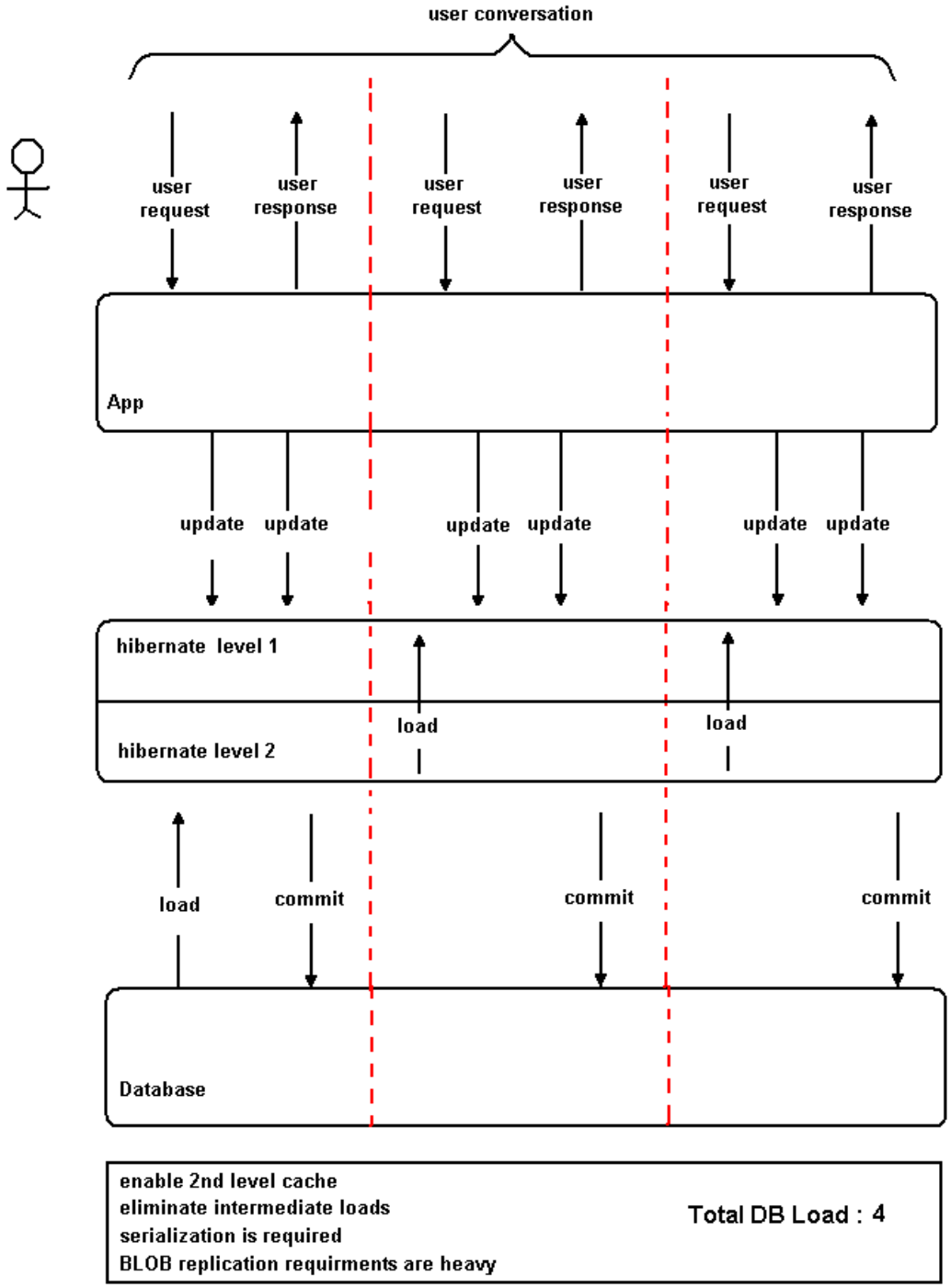

Figure 4.22 enable 2nd level cache

This level is to enable second level cache that will enable us to eliminate all the intermediate loads from database.

Here is how it works: when our user conversation begins, we still have to do the same load into memory that we did before and now we perform our updates into our first level cache. And as we end our transaction, we commit down to the database. However, we commit as well into the hibernate second level cache. So the objects exit in the second level cache. Then

when our user returns, we don't have to do these load operations as before. It is done directly in the hibernate second level cache, which means is that we are reading those objects out of the memory store instead of reaching down into the database to instantiate the data. So overall we have now eliminated the load phase from our user conversation. We have a very nice looking of database interactions here: we now take our 6 database transactions and reduce them down to 4.

However, here come problems. First of all, if we were going to store database objects into the second level cache, they would have to be serializable. If we need to cluster our second level cache, the second level doesn't store objects as actual data, they actually are stored as serialized data and that means there are no opportunities for a solution to intercept field level changes and replicate field level changes. What this means is that the cache is storing very large objects that simply serialized as a sequence of bytes. And to replicate that cache, we have to replicate the entire sequence of bytes and we have no idea what those bytes represent, which will be significant throughput costs illustrated by Figure4.23.

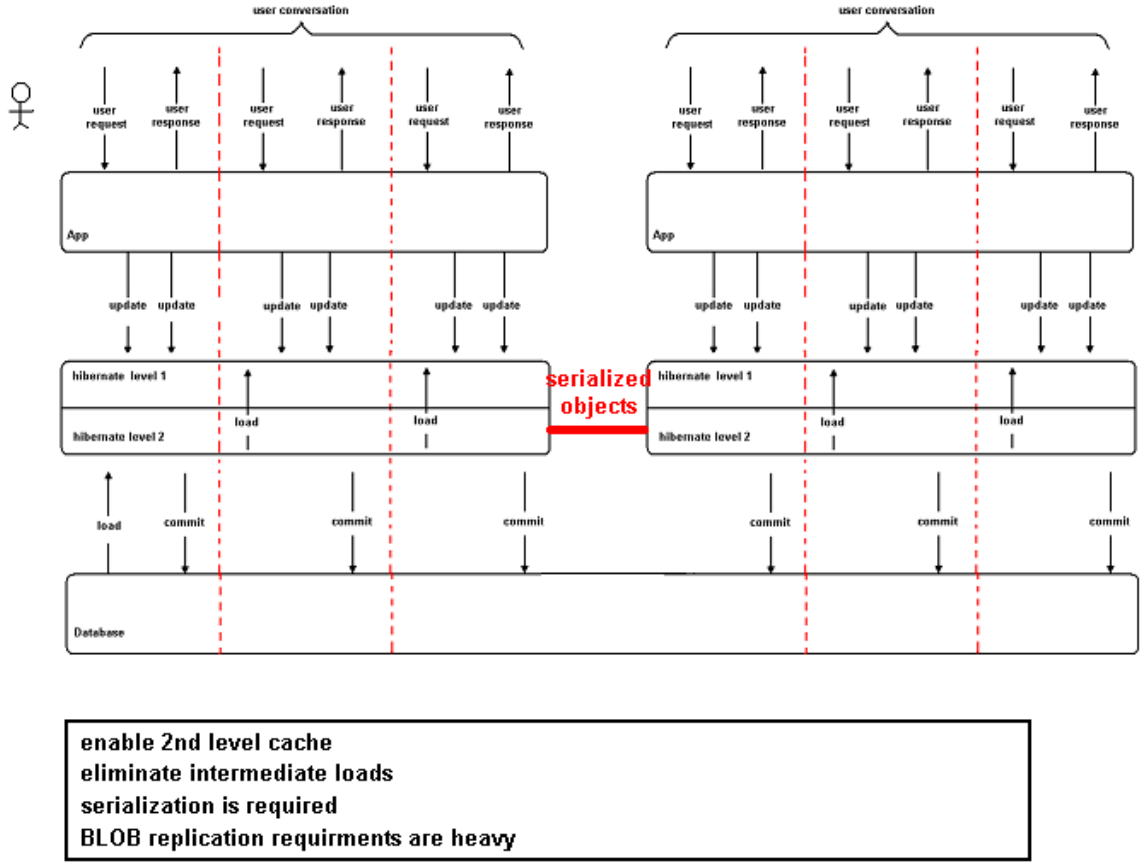

Figure 4.23 problem based on enable 2nd level cache solution in distributed environment

Basically, there are two different solutions available with Terracotta to cluster our application.

The first solution:

This one is if we have an application that enable the second level cache and we like to simply scale it out. We can simply take the Terracotta product and install it to our application, install the hibernate plug-in, install the Ehcache plug-in, enable the Ehcache clustering, then we will get a coherent cache across our cluster, which is very easy to integrate.

We create an on-line forum as a benchmark test application for a company named Devoteam. In this application, we enable the second level cache and query cache and use 4 Tomcat servers with Apache as a load balancer, as is illustrated by Figure4.24.

Note: please see Appendix A for more details on the benchmark test.

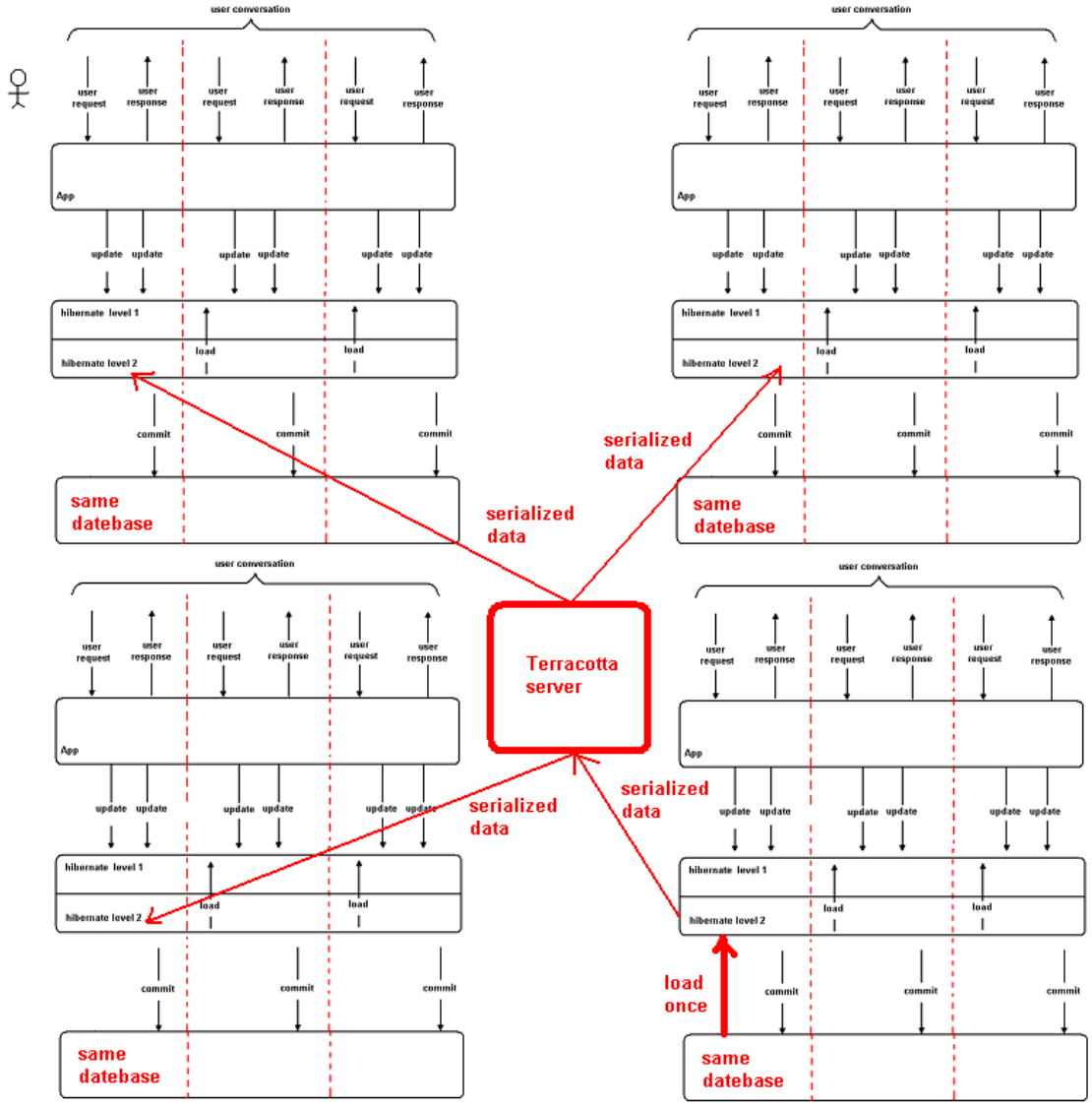

Figure 4.24 enable 2nd level cache with Terracotta

Here we can see each of these application servers got their cache clustered and instead of loading 4 times from database, now we only load once.

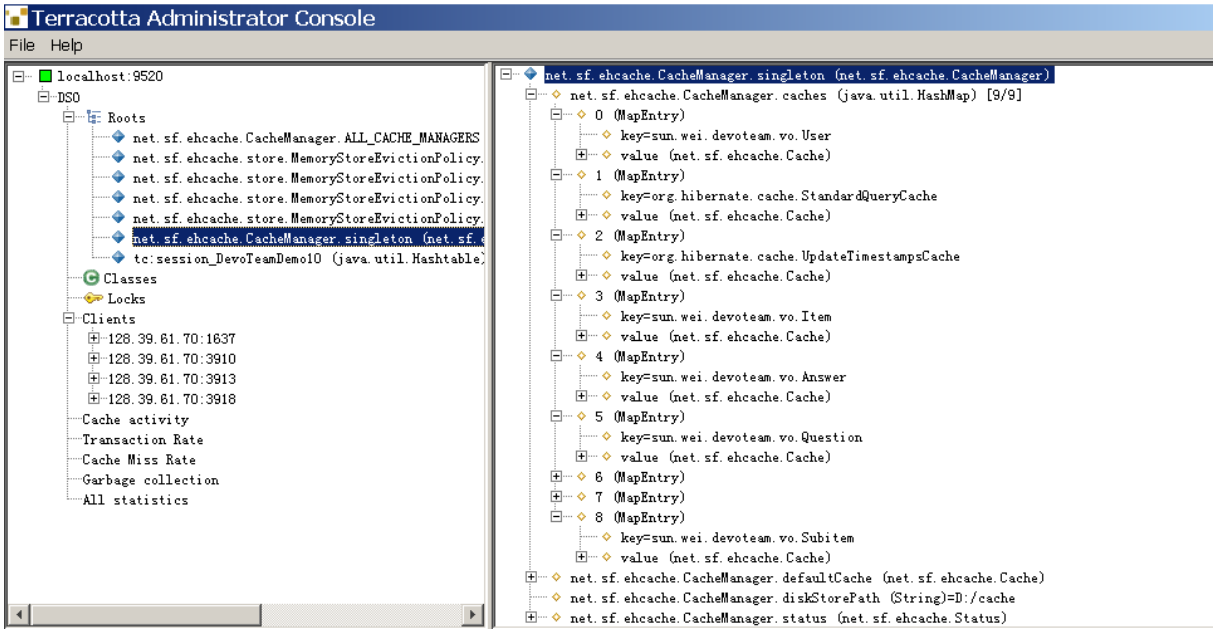

Figure 4.25 results of Terracotta administer console

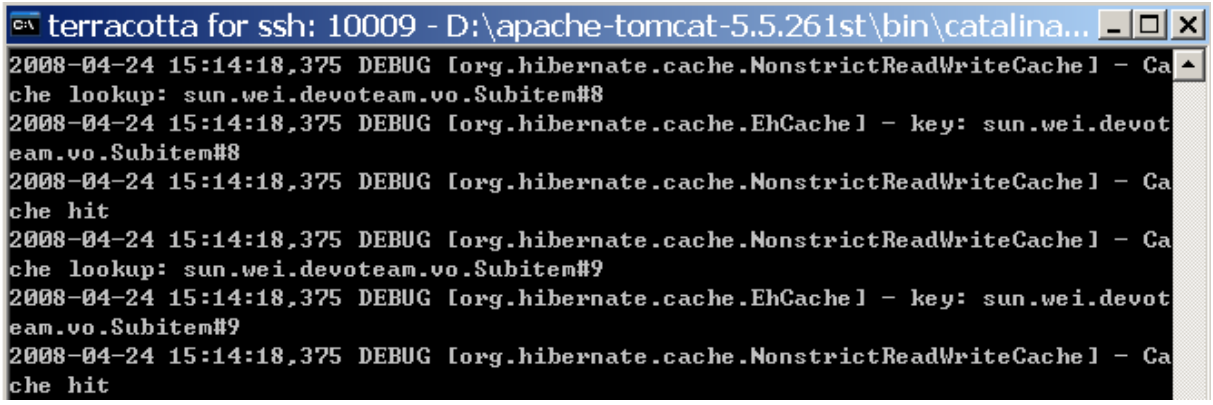

Figure 4.26 Tomcat 1

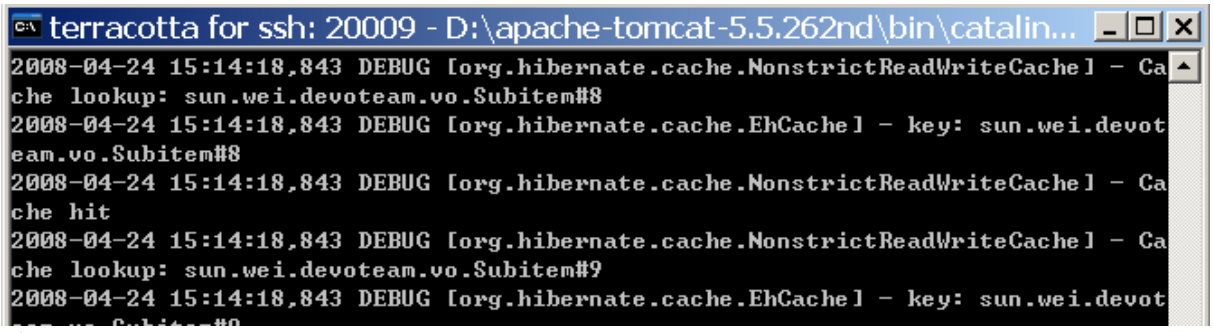

Figure 4.27 Tomcat 2

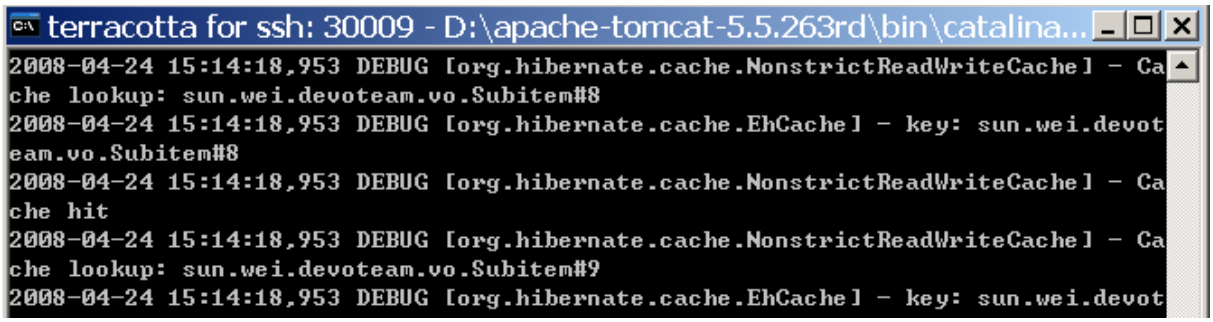

Figure 4.28 Tomcat 3

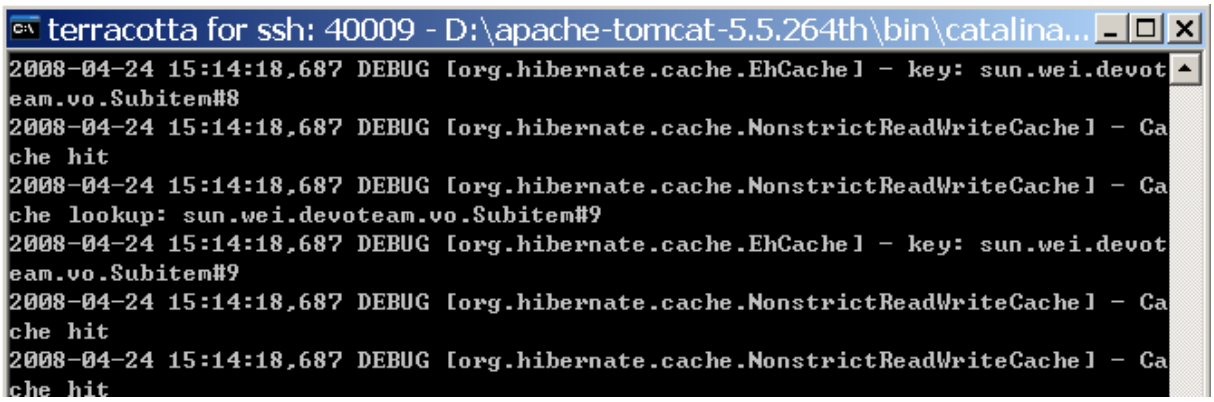

Figure 4.29 Tomcat 4

We use JMeter to test the throughputs of our cluster application.
The test case is simple as follows:
Request the index page. During this process, all the data that are needed to dynamically formalize the index page will be grabbed from database.

The following is the aggregate report for 1 application server. (thread number:100, ramp-up period:2, loop:10)

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
http://localh...	1000	1954	1000	6625	0	13687	7.80%	10.8/sec	4.2
http://localh...	1000	5564	4235	12328	62	29688	28.00%	10.7/sec	492.2
总体	2000	3759	2922	9313	0	29688	17.90%	21.3/sec	496.2

Figure 4.30 JMeter aggregate report for 1 application server

The following is the aggregate report for 2 application servers. (thread number:100, ramp-up period:2, loop:10)

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
http://localh...	1000	1608	125	2079	0	43250	0.90%	12.5/sec	4.9
http://localh...	1000	4022	3344	6563	63	45672	3.50%	12.5/sec	660.8
总体	2000	2815	1422	5578	0	45672	2.20%	24.9/sec	665.4

Figure 4.31 JMeter aggregate report for 2 application servers

The following is the aggregate report for 3 application servers. (thread number:100, ramp-up period:2, loop:10)

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
http://localh...	1000	1506	438	1109	0	43000	0.80%	13.8/sec	5.5
http://localh...	1000	3663	3125	5328	63	46750	0.10%	13.8/sec	745.1
总体	2000	2584	1156	4750	0	46750	0.45%	27.6/sec	750.7

Figure 4.32 JMeter aggregate report for 3 application servers

The following is the aggregate report for 4 application servers. (thread number:100, ramp-up period:2, loop:10)

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
http://localh...	1000	1017	375	641	0	44406	2.00%	14.0/sec	5.8
http://localh...	1000	4309	2922	6672	110	51687	0.40%	14.0/sec	752.2
总体	2000	2663	953	5453	0	51687	1.20%	27.9/sec	758.0

Figure 4.33 JMeter aggregate report for 4 application servers

Refer to figure above, we can see even we cluster our application successfully but we did not achieve the scale out linearly that Terracotta promised and we also got some errors.

We applied to the experts of Terracotta. This is exactly what they said: “If you want to test for linear scale, you should be testing with more than machine. Ideally you should be testing with one machine dedicated to each tomcat instance, terracotta server, JMeter and load balancer. If you can not do that, you should at least run terracotta server on a different machine. When you benchmark clustering results, you will see the real comparisons when tomcat instances are on distributed nodes, otherwise results will be skewed and can not be trusted.”

Unfortunately, we were actually running all of our applications, Apache load balancer, Terracotta server, and JMeter on the same machine. So we figure out that this may be we have not get the good test results.

Anyway, we still get our application clustered. This is the fast and easy way to cluster our application. However it doesn’t take the full advantage of Terracotta solution. As we have discuss previously, the objects in the cache are represented as BLOBs and Terracotta simply treat those as bytes to ship around the network, which works fine but it just don’t take the full advantage of Terracotta provides at the pojo cache layer.

The last level is that we can completely detach our pojos from hibernate. During our conversation with our user, we can store them in a session, in either an httpSession or any other types of contexts like a cache that is outside of the hibernate level. So now we can retrieve them, operate on them and reattach them at the end of the conversation and commit the entire set as a single sequence to the database. You can see from Figure4.34.

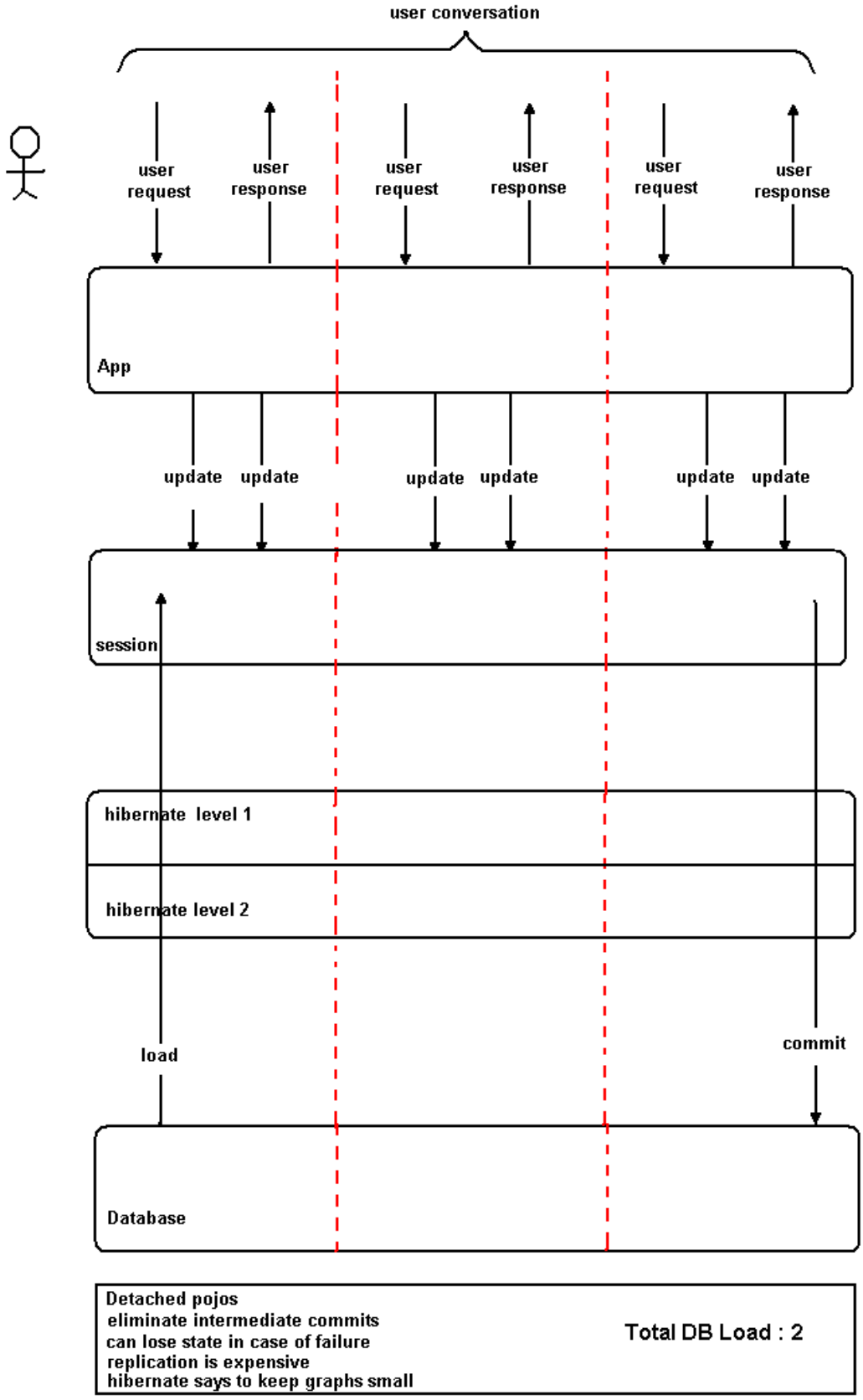

Figure 4.34 detached pojos

Because we have stored the objects locally in the httpSession, we do one load to instantiate our objects and every time our user returns, we are able to simply pull them out of the httpSession, update them in the session. At the end, when we finished with our entire conversation, we just commit our entire sequence to the database. So for now, we have really got the ultimate in database performance.

As the above Figure shows, we are loading our objects, operating on them in memory while we need them in memory and then we commit them at the very end. So all the data that is happening between the beginning and the committing are scratch data as we mentioned previously. Basically, we don't want to put these scratch data into our database. What we are doing here is putting them in the httpSession layer and committing only the important, ultimate data.

However, we still got some problems. First of all, what happens if our application server fails? Our user could lose his conversation. In the shopping cart example, maybe we can offer to lose it or maybe we can't. And if our user is interacting with us, giving our important information, we really don't want to lose it. We want our user to have a non-interrupted experience while they just check out and never knew there was a failure on the server side. Obviously, we need to cluster our application for high availability and high performance.

The problem above boils down that the replication at httpSession layer is expensive. Most application servers and almost every book we read including the hibernate documentation itself says the only way we can realistically put data into the httpSession is we keep our objects small. It is not realistic to put large object graphs in the httpSession, because we can't rely on the httpSession clustering level to replicate those in an efficient manner.

The second solution

The second solution that we can use is to cluster our pojos at the session layer and reattach them in the session. This is the best scale out solution using Terracotta, which basically takes advantage of pojos at the hibernate layer and pojos at the Terracotta layer. It can do fine-grained replication compared to the first solution and reduce the network traffic efficiently, which illustrated by Figure4.35.

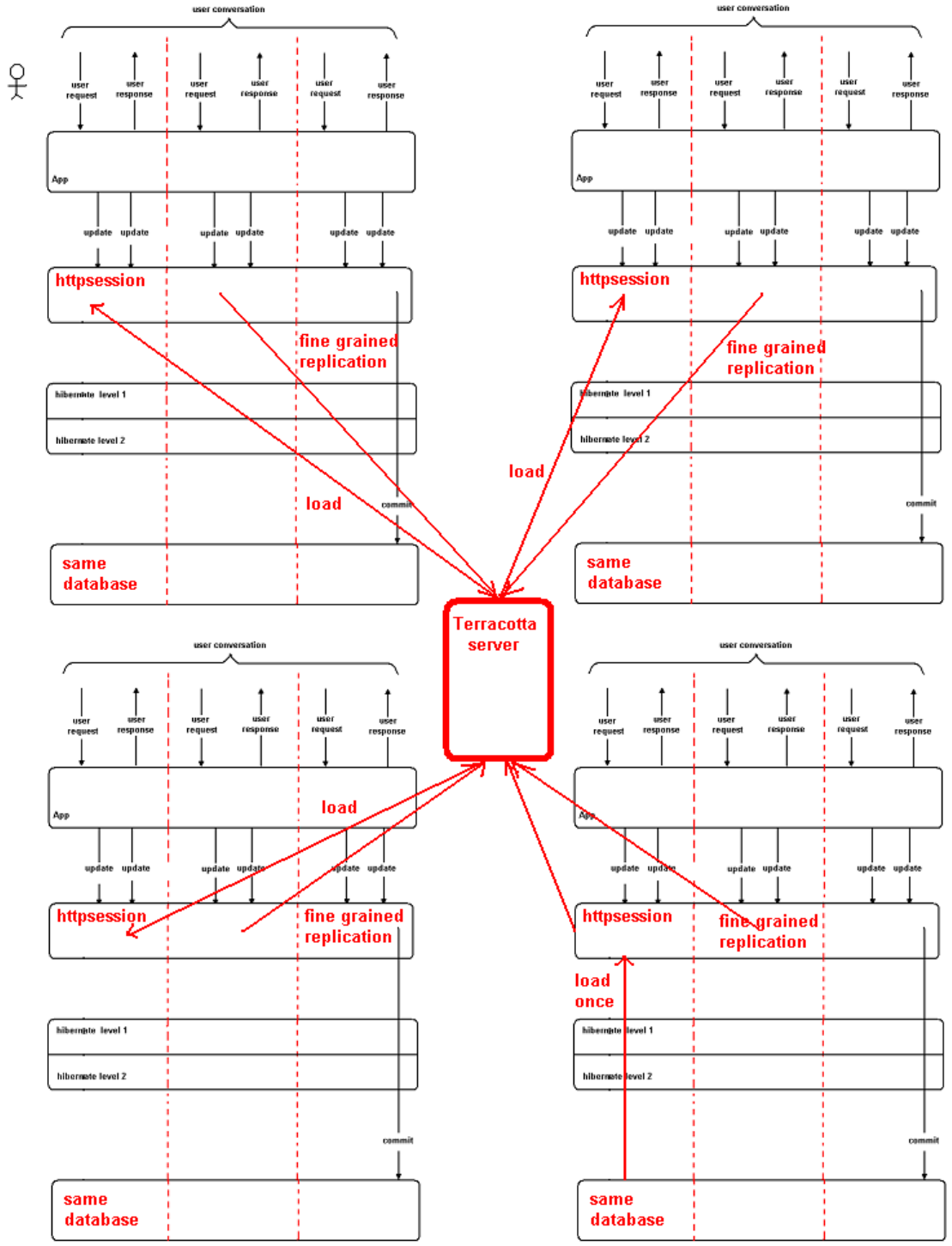

Figure 4.35 detached pojos with Terracotta

So this detached pojo solution with Terracotta gives us an extremely powerful manner to implement high available, scale out distributed applications.

Here we use JPetStore as a benchmark test. And the test case is simple as following:
1. Log in to the JPetStore application
2. Add some kind of items to the shopping cart
3. Repeat modifying the quantity of the items for 1000 iterations

Note: please see Appendix A for more details on the benchmark test.

However, as we anticipated, we successfully cluster our JPetStore application for high availability but we didn't get the linear performance as test based one single machine can not be trusted. We have to at least run Terracotta server on a different machine. As the limitation of the hardware, we didn't finish this test. The entire code and configuration files are attached for future tests.

So in this chapter, we introduce how to eliminate the abuse of database and give a best use case for Terracotta.

Chapter 5 Conclusion and Future work

5.1 Sum up

In this master thesis, we focused on how Terracotta relieves the burden of developers with regard to maintenance of a scale-out application and complexity of writing a distributed application. All the complexity of coding a distributed system is to coordinate the shared data and all the scale-out problems come from database abuse as depicted in Figure5.1.

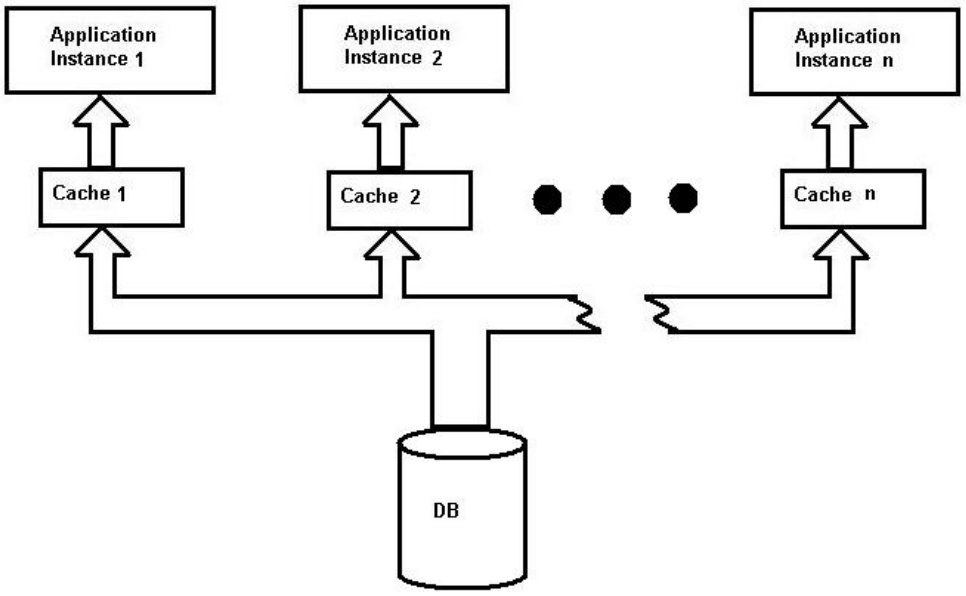

Figure 5.1 typical distributed solution

The typical solution is as following:

The Typical Solution:

1.check if k exists in the cache
2.If not, load v from the service (a database for example) using k
3.Put the canonical value (k,v) into the cache

```
public interface ItemService<K,V> {
        public V get(K k);
}

public class CachingService<K,V> implements ItemService<K,V> {
   public V get(K k)
   {
      CacheElement<V> element = cacheStore.getValue(k);
      if (element != null) { return element.get(); }

            element = getService.get(k); //read from database(dummy code)
```

```
    return cacheStore.putIfNotExists(k, element).get();
}
```

5.1.1 Complexity problem: read inefficient from database leads to complexity

From the above we can see, each application server must load and maintain its cache from database separately. What we want to do is to coordinate the loading action of each cache so that only one of them loads the object, put it into the cache, and then notifies the other pending cache that the object is available. It sounds easy. Actually, this is where that we have to use RMI, JMS, JGroups, Soap, EJB or the proprietary API to implement this sounds-easy inform-mechanism. This is where distributed applications become more and more complex.

5.1.2 Terracotta solution: Transparent, inter-JVM thread coordination

So what we need to do with Terracotta is just to simply rely on java synchronization as following:

```
public synchronized V get(K k) {
}
```

This will ensure only one cache instance reads the value. And that is all.

5.1.3 Implicit problem: limitation of heap space which leads undesirable effect on database

In the above typical distributed environment, the application layer caches are in-memory caches, which means to that they require enough memory space to hold the active data. In the ideal situation, a cache instance would remain all the data we need until they become invalid, however, if there isn't enough memory to keep all the active data, the cache will automatically purge some data out based on certain invalidation mechanism like LRU etc, even though the data is still valid. This will lead to unnecessary trips to the database to get the data, in other words, undesirable trips will burden the database.

5.1.4 Terracotta Solution: virtual heap of Terracotta

When the local heap reaches a threshold, terracotta starts a cache manager which will cleans the data that we don't need based on certain algorithm of LFU and LRU. If the data is needed again by our application, it will be fetched from the Terracotta server.

We don't have to code anything extra. Once the data is added in TC configuration file for sharing, the management of this data is totally taken care transparently by terracotta. This means once we need the data, it will be faulted in from Terracotta server automatically instead going down to the database.

5.1.5 scale-out problem: writes inefficient and Database abuse (based on ORMapping) leads to hard to scale-out

In the ideal situation, we expect that the application object pattern will be totally the same with relational data pattern in the database. However, the actual situation is we have the impedance mismatch[14] between object data and relational data. And Object-Relational frameworks like hibernate provide object durability in the application tier; it is tempting the application developers to use it to store objects that fill in the application data model rather than the business data model.

These objects are typically used by the application to store intermediate results before a final result that is eventually stored in the database.

As an example, a shopping cart object may be used to store a user's intended purchase items prior to placing an order. The data we only need in our business may only be the final placed order, but application developers may be tempted to store shopping cart data in the database to make the application to be failover—that is, we want our users to have a non-interrupted experience while they just check-out and never knew there was a failure on the server side. Without the persistence provided by the database, all the data in the shopping cart would disappear.

Such architecture may provide scalability on the surface; it in fact is the killer of the database and forces the database to scale.

5.1.6 Terracotta solution: virtual heap of Terracotta

We can imagine Terracotta works as a network attached memory, which provides the durability in object format.
So this eliminates the impedance mismatch between the object data in the application layer and the relational data in the database.
It also provides persistence which means that we don't need to put intermediate data into database.

5.1.7 what is Terracotta

Now we can define what Terracotta is:

1. Network-attached, virtual, persistent heap
2. Transparent, inter-JVM thread coordination

5.2 Conclusion

In this thesis, we delve into Terracotta compared to other open source clustering techniques. We deeply analyzed the pros and cons of various clustering techniques and summarize that Terracotta relieves the burden of database, eliminate the complexity and high maintenance of applications as scale-out. And finally we demonstrate a best use case for Terracotta.

5.3 Future work

Although we came up with all the testing code and environment, in terms of performance we don't think our results can be trusted. It seems that we just tested our computer not Terracotta, so it would be interesting to do all of the tests again with supports of appropriate hardware.

ABBREVIATIONS

JVM **Java Virtual Machine**

DSO **Distributed Shared Objects**

POJO **Plain Old Java Object**

HA **High Availability**

HC **High scalability**

HP **High Performance**

API **Application Program Interface**

JMS **Java Messaging Service**

RMI **Remote Method Invocation**

RPC **Remote Procedure Calls**

HTTP **Hypertext Transfer Protocol**

NAM **Network Attached Memory**

O/RMapping **Mapping Objects to Relational Databases**

RAM **Random Access Memory**

ASM **An all purpose Java bytecode manipulation and analysis framework**

TCP/IP **Transmission Control Protocol/Internet Protocol**

AOP **Aspect-Oriented Programming**

JSR **Java Specification Request**

SPoF **Single Point of Failure**

LRU **Least Recently Used**

LFU **Least Frequently Used**

JDBC **Java Database Connector**

ACK **Acknowledgment**

BLOB **Binary Large Object**

SOAP **Simple Object Access Protocol**

EJB **Enterprise Java Beans**

Appendix A

Test application
Scenario 1: Cache application

Scenario 2: Cache application

Scenario 3: Cache application

Scenario 4: Devoteam application and Jpetstore application

JPetstore application included in the Spring Distribution, version 2.5
Download from http://www.springframework.org/download.

Test tool:
Apache JMeter.

Download from http://jakarta.apache.org/jmeter.

Hardware:
1 x AMD (2.4 GHZ w/4 MB KB cache – 2 cores total)
2 GB Memory
Window XPsp2

Appendix B

[1] on-line wikipedia en.wikipedia.org
[2] T. Lindholm and F. Yellin. The Java Virtual Machine Speci_cation. Addison-Wesley, 1997.
[3] Terracotta white paper
[4] http://www.terracotta.org/confluence/display/orgsite/What+Is+Terracotta
[5] http://www.terracotta.org/confluence/display/orgsite/How+Terracotta+Works
[6] http://jcp.org/aboutJava/communityprocess/final/jsr133/index.html
http://www.cs.umd.edu/~jmanson/java/popl05.pdf
[7] http://www.terracotta.org/confluence/display/docs1/Concept+and+Architecture+Guide
[8] AspectJ
[9] http://www.aosd.net/2007/program/industry/I1-ClusteringJVMUsingAOP.pdf
[10] http://www.terracotta.org/confluence/display/devdocs/Architecture
[11]
http://docs.terracotta.org/confluence/display/docs1/Configuring+a+Terracotta+Server+Cluster
[12] LRU, LFU on-line wikipedia en.wikipedia.org
[13] ehcache http://ehcache.sourceforge.net/
[14] impedance mismatch on-line wikipedia en.wikipedia.org