INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Clustering e Scheduling Distribuído de VMs

Distributed Clustering and Scheduling of VMs

João Nuno Pessanha Alcoforado Sampaio de Lemos

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Professor Alberto Manuel Ramos da Cunha
Orientador:	Professor Luís Manuel Antunes Veiga
Vogal:	Professor João Manuel Santos Lourenço

Outubro 2010

Acknowledgements

I would like to thank my advisor, Luis Antunes Veiga, for all his dedication and helpful brainstorming during the development of this thesis, as well as all the reviews and suggestions that allowed me to polish the final report.

I would also like to thank all my colleagues that helped me during my course. Special mention to Bruno Almeida, Bruno Loureiro, Carlos Jacinto, Cláudio Diniz, Daniel Janeiro, Diogo Oliveira, Edgar Rocha, Jean-Pierre Ramos, João Antunes, João Coutinho Neves, João Neves, João Reis, João Ribeiro, Manuel Tiago Pereira and Sérgio Gomes, whom I had the pleasure to work with in many projects during the course. A special mention also goes to teaching assistants Andreia Silva, Filipe Cabecinhas and Nuno Lopes, for all their dedication and help in several courses.

I would also like to thank my parents and brothers, for all the support, caring and patience while I was finishing my course and master thesis.

And last, but certainly not least, I would like to thank my girlfriend, Margarida, for all the love and unconditional support that keeps me going through.

Lisboa, November 25, 2010
João Lemos

To my parents, Nuno and Maria Carlota

Resumo

Ao longo dos últimos anos, aglomerados de computadores feitos de simples estações de trabalho têm-se afirmado como o padrão no que toca à computação de elevado desempenho, dado que a escalabilidade e eficiência de custo desta solução ultrapassa a maioria das soluções que recorrem a computadores dedicados. Se as estações de trabalho num aglomerado puderem trabalhar de forma colectiva e fornecerem a ilusão de que formam um único computador com mais recursos, então teremos o que é referido na literatura como um *Single System Image*.

Neste trabalho, apresentamos o Caft, um "middleware" que se executa por cima do sistema Terracotta e que tem a capacidade de correr uma aplicação Java com multiplas tarefas de forma transparente, distribuindo-as pelos nós do aglomerado e utilizando os recursos computacionais e de memória disponíveis. São utilizadas instrumentações de "bytecode" para adicionar instrucções para correr a aplicação no aglomerado, bem como sincronização extra se necessário. O "middleware" suporta vários modos de funcionamento, de modo a atingir um equilíbrio entre a transparência e a flexibilidade de uso.

O "middleware" foi testado com uma aplicação que calcula os números de Fibonacci, uma aplicação geradora de imagens foto-realistas (Sunflow) e uma aplicação que efectua a multiplicação de uma matriz por um vector. Concluímos que a nossa solução é escalável, na medida que permite que uma aplicação com múltiplas tarefas atinja tempos de execução menores adicionando mais nós ao aglomerado. Concluímos também que é possível aproveitar os recursos de memória adicionais presentes no aglomerado.

Abstract

In recent years, computer clusters made entirely of simple desktop computers are becoming the standard for high-performance computing, as the scalability and cost-efficiency of such solution surpasses most high-end-mainframes. If the workstations in a cluster can work collectively and provide the illusion of being a single workstation with more resources, then we would have what is referred in the literature as a *Single System Image*.

In this work, we present Caft, a middleware that runs on top of the Terracotta system and has the capacity to run simple multi-threaded Java applications in a transparent way, scheduling threads across the several nodes in a Terracotta cluster and taking advantage of the computational and memory resources available in the cluster. We use bytecode instrumentations to add clustering capabilities to the multi-threaded Java application, as well as extra synchronization if needed. The middleware supports several modes, in order achieve a balance between transparency and flexibility.

We tested the middleware with a Fibonacci computing application, an Open Source renderer (Sunflow) and an application that multiplies a matrix by a vector. We concluded that our middleware is scalable, as it allows a multi-threaded application to achieve lower execution times by adding more nodes to a Terracotta cluster. We also concluded that, with our approach, it is possible to take advantage of the extra memory resources available in the cluster.

Palavras Chave
Keywords

Palavras Chave

Java

Computação paralela e distribuída

Imagem de sistema único

Instrumentação de bytecode

Terracotta

Keywords

Java

Parallel and Distributed Computing

Single-System Image

Bytecode Instrumentation

Terracotta

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The designation "Virtual Machine" has been around since the 60s, and it was originally used to describe a software implementation that executes programs like the "real" hardware. In those days, hardware-level virtual machines were popular [40], and several VMMs, like IBM's CP-40, were developed at that time. This allowed IBM to run several single-user operating system instead of a multi-user operating system, such as Unix. However, the VMM solution implied higher overheads and difficult design decisions, such as what should be handled by the VMM and what should be handled by the guest OS (e.g. swapping memory to disk). In a system where both the VMM and the OS had mechanisms for page swapping we could end up with conflicts or a suboptimal decision. As a result, multi-user OSs ended up being widely adopted and the concept of "Virtual Machines" was abandoned in this context. Other virtualizations were developed some years later, such as the P-code, which allowed the wide spread of the Pascal language. In the late 1990s, with Sun's Java Virtual Machine becoming widely used, the need of a virtualization layer between the programming language and a real machine became very clear, offering more portability, less code size, and easier to implement debugging at runtime. Nowadays, the term "Virtual Machine" designates a full taxonomy of different virtualizations, which some authors such as Smith et all [45] try to classify. Despite the variations, we can define a virtual machine as a target architecture for a developer or compilation system, that can have or not correspondence to an existing physical hardware.

In recent years, computer clusters made entirely of simple desktop computers are becoming the standard for high-performance computing, as the scalability and cost-efficiency of such solution surpasses most high-end-mainframes. If the workstations in a cluster can work collectively and provide the illusion of being a single workstation with more resources, then we would have what is referred in the literature as a *Single System Image* [12]. Much research has been done in the area of SSIs, providing sophisticated systems that achieve a single view of resources such as process space (OpenSSI) or filesystem (NFS). One of the initially most promising techniques that has been widely used is Distributed Shared Memory (DSM). By extending the traditional virtual memory architecture we can provide a distributed global address space that allows a cluster composed of different machines to be used as a shared memory system. The first software-based DSMs systems such as TreadMarks [2] simply organized the memory into pages of fixed sized that were split across the machines in the cluster, with a *Release Consistency* [23] algorithm to provide proper synchronization. Unfortunately, programming in accordance with the consistency algorithm proved to be a difficult task and the performance was far from excellence. Modern DSM systems such as Terracotta [48] follow an object-based approach and the memory is organized as an abstract space for storing objects of different sizes, offering a transparent location of objects to the applications.

Those and many other systems are described in more detail in chapter 2, including the algorithms used for guaranteeing consistency and minimizing communication among nodes.

Considering these facts, and the known popularity of programming languages designed for running with a High Level Language VM such as Java or C#, it is worth studying the possibility of extending a VM with clustering support in a SSI fashion. There are three major approaches for achieving this goal:

- **Extend a programming language at source or bytecode level:** allows a simple and straightforward implementation fully compatible with current VMs but it does not provide full transparency to the programmer and existing applications need to be modified or recompiled for using a specific library. In either case, the application source might not be available.
- **Design a cluster-aware VM:** gives full transparency to the programmer but requires the applications to use a specific cluster-aware VM instead of any standard VM, sacrificing the portability of the system.
- **Design a cluster infrastructure capable of running several standard VMs:** gives the best compromise between portability and transparency but it is the hardest one to develop, due to the fact that its layered approach makes it difficult to use the runtime information present at the JVM level. Also, due to the mismatch between the memory models of Java and the underlying DSM [53], non-trivial optimizing techniques need to be employed to enable efficient object sharing among distributed Java threads. Many implementations are incomplete and do not provide a full SSI, as we will observe in chapter 2.

One of the essential mechanisms necessary for providing SSI systems is the scheduling of threads for load balancing across the cluster. To the best of authors knowledge, some work has been done in improving the scheduling of threads for page-based DSM systems in order to avoid *Page-Thrashing* and improve the locality of memory accesses but no modern DSM system can provide the full transparency desired for running already existent applications. The current most popular system that uses the concept of a shared object space is Terracotta, a middleware that promises to deliver performance at any scale for high popular frameworks like Hibernate and Ehcache. Terracotta is used in a high percentage of the companies belonging to Forbes Global 2000, and being an Open Source project, also has very good support from its community of users, with frequent bug reports and documentation updates. All this characteristics make the Terracotta system a favourite to use as a basis for our work.

At present, Terracotta has no concept of global thread scheduler and the programmer of a multi-threaded application needs to be concerned about manually launching multiple instances of the applications, and manual load-balancing. Also, a current Java program that launches several threads using only the Java Thread class cannot take advantage of the Terracotta infrastructure. Considering these limitations, we believe that if we had a middleware that could bridge both Terracotta and multi-threaded Java applications, handling the scheduling of threads and using the existent shared object space to keep data consistent, we could run already existing applications in a distributed environment with almost no extra effort and obtain scalability. Also, developing an application from scratch would get easier, as the programmer could develop the application just like if it were meant to be deployed on a single multi-core machine. This belief holds the main motivation for developing **Caft**, a Cluster Abstraction for Terracotta, that we will introduce in this document.

A global scheduler can also have thread migration support, as the load in the cluster changes overtime and it becomes necessary to rebalance the load, as many authors defend that otherwise the communication

overhead becomes a bottleneck in performance [49]. Also, in a heterogeneous cluster the processors differ in speed and the computational resources available also change during runtime, adding extra complexity for the global scheduler to deal with [50]. Despite these difficulties, recent research in Virtual Machine technology has allowed the concept of *capsule* to appear and entire systems can be migrated within a cluster for user commodity and also for load-balancing. Recent studies by Chen el all. [17] have showed that this approach can be just as efficient as thread migration. Both concepts will also be approached in chapter 2 of this document, and at least considered for future versions of the Caft middleware.

1.1 Objectives

In this work, we aim to develop a prototype of a Java-based SSI system with a global thread scheduler that can provide efficient load-balancing across an entire cluster of computers. The system should be flexible enough to provide good transparency for running an existent application without much changes, and still be powerful enough in order to allow the programmer to optimize it, depending of the application itself. It is also worth studying the possibility of integrating such a system on top of a VMM supporting Virtual Machine Migration or extending the system with thread migration mechanisms for improving the load balancing.

With our middleware, it should be possible, for example, to run an application with four threads, where a pair will run on a dual-core machine and another pair on another dual-core machine, and the application should perform more work in less time, achieving speed-up. Also, if the threads are running in separate machines, each with its local heap and memory capacity, it should be possible to take advantage of the extra memory available in the cluster. These are the main requisites concerning performance and scalability.

Concerning transparency, required source code changes for clustering the application should be kept to the minimum. The scheduling of threads and synchronization should be done at bytecode level, using configuration files or command line parameters that can be changed easily for extra flexibility. However, we believe that this requirement should be relaxed in case the source code is indeed available and the programmer desires to fine tune the application. As such, the programmer should be able to enable or disable some instrumentations, as well as have some mechanisms to choose the data structures that should be shared and the ones that should not.

1.2 Contributions

Considering the objectives defined, we developed Caft as a middleware to be run on top of Terracotta. It can be configured to either run as a **master** or as a **worker**. The former will load and run the desired multi-threaded Java application, while the latter will wait for requests from the master to run threads. The idea is to deploy one master and several workers and be able to obtain scalability by having more CPUs and memory available for running threads and parallelizing the application more than it would be possible with a single node.

For the scheduling of threads, we opted to implement a simple scheduler that keeps track of all the workers available and assigns the next thread in a round-robin fashion. This allowed us to begin with a simple implementation that can scale with multi-threaded Java applications that divide the workload in

equal parts. In the future, a more advanced scheduling can be considered, and in chapter 2 we perform a survey of scheduling algorithms and migration techniques to improve load-balancing.

Also, considering the performance versus transparency trade-of, as well as the availability of source code, we developed Caft with three different modes: Identity, Full SSI and Serialization. Identity mode should be used if we have a multi-threaded Java application that is properly synchronized, or the programmer has access to the source code and can add synchronization with ease. Full SSI mode should be used if we have a multi-threaded Java application that is not properly synchronized, or the programmer has no access to the source code. In both modes, all fields belonging to a Java `Runnable` target that is passed to the Thread class will be shared. Serialization mode allows the programmer to specify the fields that need to be shared using Java annotations, allowing for a more fine grained configuration.

To summarize, we present the contributions of this work in the following list:

- Proposal of an architecture for round-robin scheduling over Terracotta (with the possibility to improve in the future)
- Implementation of the proposed model, available as a prototype
- Implementation of different modes, to add more flexibility
- Identification of the limitations of Terracotta for applications with many data-sharing
- Evaluation of the proposed system

1.3 Document Roadmap

This document is organized as follows. Chapter 2 describes in detail the context of our work, including some SSI systems that share our topics of interest. Chapter 3 describes the architecture of the middleware developed, using Terracotta as an infrastructure for running multi-threaded applications. Chapter 4 describes the implementation of the middleware in further detail, focusing on the bytecode instrumentations that it performs on the Java application. Chapter 5 describes the evaluation method to measure solution adequacy and performance. Chapter 6 summarizes all work done, introduces some ideas for future developments, and draws some conclusions.

Chapter 2

Related work

In this chapter, we are going to focus on solutions developed in the academic world and in the industry for providing a SSI view of a cluster, particularly for providing a global address space. In section 2.1 we describe the Distributed Shared Memory (DSM) approach, as well as the consistency models that support it and the adaptations necessary to make a common application work with a specific consistency model. In section 2.2 we are going to examine systems that integrate a global address space with a software platform that can make a regular application written in Java to become cluster-aware and run seamlessly with minimal programmer intervention. To finalize, in section 2.3 we are going to focus on scheduling algorithms and migration techniques to improve load-balancing.

2.1 Distributed Shared Memory

Distributed Shared Memory Systems have been around for quite some time, and it was one of the first solutions adopted for clustering [39]. By extending the traditional virtual memory architecture, the distributed memory is hidden from the programmers and applications can be developed using the shared memory paradigm instead of other traditional and more error-prone, albeit more performant, parallel computing communication forms such as message-passing. Like in traditional shared memory systems, there is a possibility that two or more processors are working in the same data at the same time, and as soon as one of them updates a value the others are working in an out-of-date copy. To solve this problem, there are a significant number of possible data consistency models that were adopted by DSM implementations [39], which will be described in section 2.1.1. In section 2.1.2 we are going to describe several practical software DSM systems that were developed in the academic world. Finally, in section 2.1.3 we are going to focus on software transactional memory, an alternative and promising concurrency control mechanism analogous to database transactions.

2.1.1 Consistency models

The very first consistency model was Sequential Consistency [32], which is the simplest and most restrictive consistency model. Roughly speaking, sequential consistency requires that all writes be immediately visible to all processors accessing each memory page. This synchronization in every memory access is expensive and in many cases it is stronger than necessary for a distributed application to run correctly.

Therefore, a more relaxed model is needed to minimize the number of messages exchanged and the amount of data in each message, as a high amount of traffic in the network can have a serious impact on the performance of the system.

Release Consistency (RC) [23] was one of the first and most important relaxed consistency models developed for concurrent programming. This model has two synchronization operations: *acquire* and *release*. The former is used by any processor before attempting to make a write to a given object belonging to the global address space, while the latter is used after the writes are done. Therefore, in the RC model, the writes made by a certain processor p1 only need to be seen by all the other processors in the cluster after p1 releases the lock, so all writes from p1 could be queued and put in a single message which is sent to all the nodes.

Lazy Release Consistency (LRC) [31] is an algorithm similar to RC, but instead of globally propagating all changes at the time of a processor release, it postpones the propagation to the time of acquire, guaranteeing that the acquiring processor will receive all changes that "precede" the acquire operation. For example, consider a scenario where processor p1 acquires a lock over an object A, performing a few writes and then releasing it. If a processor p2 attempts to acquire a lock over A, both the lock and the writes will be propagated from p1 to p2 (and only to p2). Similarly, if another processor p3 tries to acquire the lock over A it will receive from p2 all the writes done by p1 and p2 before the p3 acquire of the lock. Therefore, for each acquire operation only one message needs to be sent, and naturally, only the differences between each memory page need to be sent.

Entry Consistency (EC) [7] is another memory consistency model. It was first used in Midway, a programming system for distributed shared memory multicomputers. The entry consistency model takes advantage of the relationship between typical synchronization objects that define critical sections, like mutexes or barriers, and the data protected by those objects. Since a critical section defines a region where the data may have been written by another processor, and a synchronization object controls a processor's access to the data and code inside it, the view of the shared memory can become consistent only when the processor enters that same critical section. Performance measurements were promising, as the number of messages decreased a lot comparing to RC. However, comparing with LRC, the results were about the same and the need to have an explicit association between every object and a synchronization variable can be troublesome for the programmer.

Automatic Update Release Consistency (AURC) [26] is yet another release consistency model that uses automatic update to propagate and merge shared memory modifications. Automatic update is a communication mechanism implemented in the SHRIMP multicomputer that forwards local writes to remote memory transparently, which is accomplished by having the network snoop all write traffic on the memory bus and checking if the page written has an automatic memory mapping, that is, if the source process virtual address is mapped with a virtual address from a remote process. If such a mapping exists, all writes to the source page will be automatically propagated to the destination page. Consecutive written addresses are combined into a single packet, in order to reduce the network traffic. This allows for zero CPU overhead in synchronization, as the only thing a processor has to do is to store the write in the memory address as he usually would. As a result, performance is substantially increased comparing to the original LRC, but unfortunately AURC is dependent on specialized hardware support.

Scope Consistency (ScC) [27] was designed as an improvement to the EC model, offering most of the advantages without the explicit binding between variables and synchronization objects. ScC introduces a new concept called *consistency scope* to establish the relationship between data and synchronization events implicitly from the synchronization already present in programs to implement release consistency.

Table 2.1: Consistency models

Model	Time of propagation	Program modifications	Hardware dependent
SC	page write	None	No
RC	lock-release	acquire/release operations	No
LRC	lock-acquire	acquire/release operations	No
EC	critical section entering	object/lock association	No
AURC	page-write	None	Yes
ScC	consistency scope entering	None, if RC consistent	No

A consistency scope consists of all critical sections protected by the same lock.

In conclusion, all consistency models can reduce communication and give some performance improvements, but they are very dependent on the applications synchronization mechanisms and may not work without some tinkering. In the next subsection, we are going to describe some systems that were developed in the academic world as proof of concepts to distributed shared memory and consistency models. The table 2.1 summarizes the consistency models main proprieties.

2.1.2 Software Distributed Shared Memory Systems

Ivy [33] was one of the very first distributed shared memory system prototypes to be implemented and proven to be more simple than the traditional message-passing interface. Read-only pages could reside in more than one node but a page marked for writing could only reside in one node and the mapping-manager would map the writes to the remote node, guaranteeing simple sequential consistency at all times. The nodes were simple single processor machines, which means that no multithreading was considered. Unfortunately, the large size of consistency unit makes the system prone to the *false sharing* problem. The false sharing problem occurs when two or more unrelated objects are written concurrently on the same page, causing the page to "ping-pong" back and forth between the processors.

Munin [14] is a second-generation distributed shared memory system. Compared to Ivy, it was also used with simple single processor machines but it uses a release-consistent memory interface to reduce the overhead, as seen in the previous section. Also, it supports multiple consistency protocols by having the programmer annotate each shared variable to establish the protocol according to the expected access pattern, and then allowing it to change at runtime. Despite the improvements, a more transparent model to the programmer was still needed and Munin still uses a home-based protocol for handling memory pages (e.g. a memory page belongs to a node and needs to be entirely fetched on a page fault).

TreadMarks [2] is another second-generation distributed shared memory system. It uses Lazy Release Consistency to reduce the number of messages used in comparison to Munin and it also supports multiple-writers by creating a twin copy of the virtual memory page and when the modifications need to be sent to another processor the differences between the page and the twin copy are put in a separate data structure to be sent and the twin is discarded. This way, the overall bandwidth is reduced comparing to the Munin

Table 2.2: Software DSMs

System	Consistency Model	Multithreading support
Ivy	SC	No
Munin	multiple	No
TreadMarks	LRC	No
Brazos	ScC	Yes

home-based approach.

Brazos [46] is a third-generation distributed shared memory system, supporting multiple multi-core processor nodes. Brazos uses a software-only implementation of Scope Consistency and a distributed page management system similar to the one in TreadMarks. Comparing to previous generation DSMs, Brazos is multi-threaded and can overlap the computation with the communication latencies associated with many DSM systems. Also, it uses multicast instead of multiple point-to-point messages, reducing the communication necessary and improving the performance.

In conclusion, despite the improvements that were made to adapt the DSM concept to new hardware, all these prototypes imply a different programming approach that is impractical, as most programmers do not want to have that many worries to guarantee that the multi-threaded application that is perfectly fine on one computer works correctly with a given consistency model. This problem gets even worse if the systems support multiple consistency models, and so a more transparent system is needed if we want it to be used for general-purpose applications. The table 2.2 summarizes the Software DSMs studied:

2.1.3 Software Transactional Memory

So far, all systems and consistency models considered are based on a pessimistic lock-based approach with the definition of critical sections to protect data. A new approach called Transactional Memory [25] was developed to try to circumvent the three main issues with lock-based solutions:

- **Priority inversion:** can occur if a low-priority thread gets hold of a lock before a higher priority thread. Because there is a mutual exclusion paradigm, the higher priority thread will have to wait until the lock if free.
- **Convoying:** can occur if a thread holding a lock is preempted by the scheduler by some kind of interrupt (e.g. a page fault) resulting in other threads inability to progress.
- **Deadlock:** can occur if two threads try to get hold of the same data sets and both wait for the other to release it, being both unable to progress.

Besides these three main issues, the lock mechanism is conservative by nature and if there *might* be a conflict in a certain region, only one thread will be allowed in that section, even if at runtime the probability of conflict is not high. Therefore, a new concept of transaction was introduced in memory operations, very similar to the transactions in relational databases. Instead of having locks, all threads are

allowed to execute a critical region at the same time and after finishing the operations a conflict detection algorithm is run. If there are no conflicts, the writes are made permanent into memory, otherwise the atomic operation is rolled back and retried at a later time.

There are two main approaches in implementing STMs: *transaction log* and *locks*. The former is implemented by having a transaction log local to each thread. All writes are done in the transaction log and at the end they are written to the memory after checking that there is no conflict, which means that the rollback operation is trivial but the commit operation implies much larger overhead. The latter can be further divided into two approaches: *commit-time locking* and *encounter-time locking*. The former is implemented by locking all memory locations during commit and marking access time with a global logical clock that is checked in every read/write and if the memory was accessed after the beginning of the transaction the transaction is aborted. Again, this makes the rollback simple but the commit, which should be the most common operation, expensive. The latter just gives exclusive access of the memory positions to a thread and all other threads that try to access the same memory positions simply abort, putting the largest overhead in the rollback operation, which in the STM paradigm should be the less common operation.

Unfortunately, STM also has some disadvantages that makes it still unpractical for very large systems, as the overheads from conflict detection and commit cannot be avoided. Also, some operations cannot be undone by nature (e.g I/O). Some authors have proposed new instructions for the IA-32 ISA to improve performance [41], but no standard processor available in the market adopted them yet. Sun has recently attempted to develop a multi-core processor capable of supporting hardware transactional memory [16], which unfortunately was canceled in November 2009. We believe STM has a lot of potential in the future once there is hardware support for transactions that amortizes the inherent overheads, as STM programming is far less error prone.

2.2 Distributed Virtual Machines

In section 2.1, we saw how the DSM abstraction can provide an SSI view of a cluster and the main concerns in implementing it. The next logical step is to study how we can combine this virtual memory abstraction with a platform that can make a normal application written in a high-level language cluster-aware without modifying the source code. Due to the popularity of Java programming language, not only for commercial applications but also for research due to its open-source nature, most of the systems presented in this section focus clustering of Java applications but similar approaches could be done for other high-level languages, such as C#, etc.

The current techniques used for supporting distributed execution in a cluster can be divided in three major categories. The first set can be classified as *Compiler-based DSMs* and it consists of a combination of a traditional compiler and a DSM system (see section 2.1). By compiling a normal application we can insert special instructions or bytecodes that add clustering support without modifying the source itself. The second set can be classified as *Cluster-aware Virtual Machines* and it includes implementations of Virtual Machines that provide clustering capabilities at middleware level. For instance, cJVM is a cluster-aware Virtual Machine with a global object space. The last set can be classified as *Systems using standard VMs*. In this approach, the applications will run on standard VMs that run on top of a DSM system. Some systems that rely on standard VMs also have static compilers similar to the *Compiler-based DSM* approach, with the major difference being that they transform a Java bytecode application

into a parallel Java bytecode application instead of native code. Other systems, like Terracotta, perform bytecode enhancement at load-time.

2.2.1 Compiler-based DSMs

The need to combine both performance and cluster-aware capabilities have led some authors to develop compilers that put special checks or instructions in the program at compile-time, adding cluster-aware capabilities without modifying the source code. The application can then be run as any native application would, in a virtual or real machine.

Jackal [52] incorporates a DSM system with a local and global GC that provides full transparency relative to the location of threads and objects. Jackal compiler generates an access check for every use of an object field or array element and the source is directly compiled to Intel x86 assembly instructions, giving the maximum performance of execution possible without a JIT. Jackal has no support for thread migration or load balancing.

Hyperion [4] also has a runtime that gives the illusion of a single memory space and it supports the remote creation of threads, which provides a better load-balancing. To keep the objects synchronized, a "master" copy is kept and updated in every write, resulting in a performance bottleneck.

In this approach, classes with native methods cannot be distributed as the already compiled code is not portable. Also, the compilation to native code indicates that these systems will only work in a homogeneous cluster, which is a severe limitation to our *Single System Image* ideal.

2.2.2 Cluster-aware Virtual Machines

Many Cluster-aware Virtual Machines were developed in an attempt to provide a *Single System Image* view of a cluster, especially in Java as it is a very widely used platform for developing object-oriented applications. Java/DSM [54] was one of the very first platforms for heterogeneous computing to be able to handle both the hardware differences and the distributed nature of the system, as the alternative of developing a distributed application with RMI required extra effort from the programmer. Despite the better abstraction, Java/DSM did not explore Java semantics for performing optimizations and the load-balancing was limited since it had no thread migration mechanisms. Also, every node needed to have a copy of every shared object, which meant that all the extra memory added by having more nodes in the cluster was just wasted.

cJVM [5] distributes the application's threads and objects over the cluster without modifying the source code or the bytecodes. Java object access an memory semantics are exploited, allowing optimization mechanisms such as caching of individual fields and thread migration. In the original implementation, there is no JIT support and it only works with an interpreter loop. To keep the objects synchronized, a "master" copy is kept and updated in every write, resulting in a performance bottleneck compared to the original Sun JVM.

Kaffemik [3] followed an approach where every object is allocated in the same virtual memory address in every machine. The biggest advantage is that the address can be used as a unique reference that is valid in every instance of every Kaffemik node. The virtual machines then work together, each containing

a part of the global heap. Unfortunately, Kaffemik had no means of caching or replication which meant that an array access for example could result in several remote memory accesses, reducing performance.

JESSICA2 [56] also provides a global object space (GOS) that gives the illusion of a single heap. Each JVM heap space is divided into two sections, one that contributes to the global heap space and stores master copies of objects and another for object caching for improving performance. An interesting optimization also referred in the article is the possibility that a cached copy of an object can become the "master" copy if accessed many times, which allows locality improvements at runtime by migration of ownership of the objects. To support thread migration and be able to restore the Java thread stack in a different memory space, the stack is captured at bytecode boundary an translated into a platform-independent text format to be restored by the target JVM. JESSICA2 also supports JIT compilation, which is a major improvement relative to the previous systems.

In conclusion, the major advantage of this approach is not having to modify the applications, as all clustering is done at the VM level. Despite the very promising systems described above, all of them have a major disadvantage as they sacrifice one of the most important features of Java: cross-platform compatibility. Also, the already existing JVM facilities such as local garbage collection and JIT compiler are difficult to integrate in this type of systems. Therefore, these special cluster-aware VMs either invest a considerable amount of effort reimplementing such features or they do not implement them at all. It would be interesting if the clustering capabilities could be used with a combination of different virtual machines and in the ideal scenario we would use the standard and better supported Sun's Java Virtual Machine. This approach will be described in the next chapter.

2.2.3 Systems using standard VMs

JavaParty [55] was one of the very first platforms to support the aggregation of several standard Java Virtual Machines and allow the execution of a multi-threaded program in a clustered environment. JavaParty [55] extends the Java language with a new "remote" keyword to indicate that a certain class and its instances should be visible anywhere in the distributed environment, avoiding explicit socket or RMI communication. This implementation does not fulfill the ideal SSI since the programmer has to explicitly point the classes to be clustered and needs to distinguish which invocations are remote and which ones are local because the argument passing conventions are different.

JavaSymphony [22] works under a new concept of *Virtual Architectures* that impose a virtual hierarchy on a distributed system, allowing the programmer to explicitly control locality of data and load balancing. Again, this is far from our ideal solution of having a SSI system as all objects need to be created, mapped and freed explicitly, which defeats the important advantage of built-in garbage collection in the JVM. The entire process can be quite cumbersome and since JavaSymphony does not provide assistance for these steps, the semi-automatic distribution is likely to be error-prone.

Addistant [47] works by transforming the bytecode of the Java application at load time and the developers only have to specify the host where instances of each class are allocated. All the instances of the same class are then allocated in the same node, giving poor load balancing flexibility. Moreover, the population of the cluster (number of nodes) is static and must be known in advance. System classes with native code cannot be migrated as there is no bytecode to instrument at load time. Also, application classes that use system classes with native code generate dependencies that make the former non-migratable.

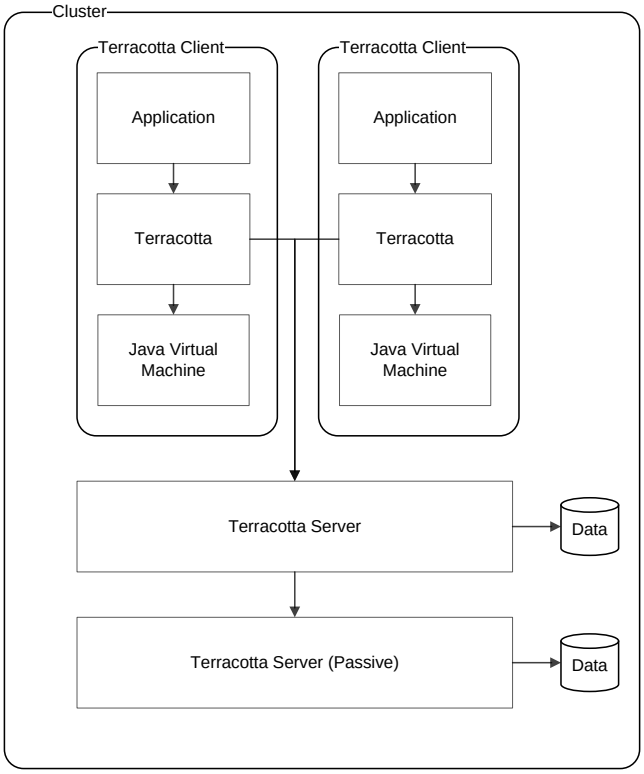

Figure 2.1: Terracotta architecture

J-Orchestra [51] also uses bytecode transformation to replace local method calls for remote method calls and the object references are replaced by proxy references. J-Orchestra can partition a Java program in such a way that any application object can be placed on any machine. Additionally, any object can be migrated to a different node at run-time to improve load-balancing and take advantage of a better locality. J-Orchestra also offers some run-time optimizations such as the lazy creation of distributed objects that do not suffer the overhead of registering until they need to be used. Despite these improvements, the tools provided by J-Orchestra to determine class dependencies and to ensure the correct partition requires non-trivial intervention from the user, still not achieving the SSI ideal. In addition, the bytecode instrumentation technique has the same limitation as Addistant (objects that have or depend on native code cannot be migrated).

JavaSplit [21] is yet another runtime for executing Java applications that uses bytecode instrumentation for adding clustering support. JavaSplit supports the multi-threaded paradigm directly, without introducing unconventional programming constructs. All the bootstrap classes are rewritten with JavaSplit and the final result is a distributed Java application that uses nothing besides its local standard Java Virtual Machine (JVM). Each newly created thread is placed on one of the worker nodes using a load-balancing function and thread migration is not supported.

Terracotta [48] is a recent JVM-level clustering product, used in a high percentage of the companies belonging to Forbes Global 2000. Terracotta supports full transparency in a way similar to JavaSplit except that it works within an aspect-oriented programming (AOP) framework. To take advantage of the Terracotta clustering model, an instance of the Java Application needs to be launched in every node and the central Terracotta Server also needs to be setup. The programmer has to configure his Java application to decide which fields in each class remain local and which ones are going to belong to the Distributed

Shared Objects space (DSO), as well as all locking and synchronization concerns. The Terracotta (TC) libraries are loaded by each JVM running the application and are responsible for handling the bytecode instrumentation at load-time for implementing the behavior specified by the programmer. The Terracotta Server implements the Virtual Memory Manager (VMM), which is responsible for holding the global heap and propagating the differences to the JVM clients (objects are cached on disc before the server runs out of memory). Also, the Terracotta Server can itself be clustered for improved scalability. Figure 2.1 illustrates the architecture described. The dashed squares represent a cluster node, either corresponding to a real or virtual machine.

The main features of Terracotta make it an appropriate platform for clustering application servers like Apache Tomcat or JBoss, but it lacks transparency for running multi-threaded applications non-cluster aware. In chapter 3 we are going to propose a Terracotta extension that attempts to make the scheduling of threads in a cluster transparent.

To summarize, the table on the next page illustrates the main features of all systems studied in this section. In section 2.3 we are going to study the existing scheduling algorithms and thread migration techniques in order to have a good theoretical background to choose the best approach for extending Terracotta.

Table 2.3: Distributed Virtual Machines

System	Application changes	Interoperability	Single heap topology	Object caching	Global GC	JIT	System classes clustering support	Load-balancing mechanisms	Thread migration
Jackal	source compilation	same ISA	Fixed compile-time object distribution	No	Yes	N.A	No	None	No
Hyperion	source compilation	same ISA	Fixed compile-time object distribution	No	Yes	N.A	No	Initial thread placement	No
Java/DSM	None	same VM	Shared objects in every node	No	No	No	No	None	No
cJVM	None	same VM	Master/proxy	No	No	No	No	Initial thread placement	Yes
Kaffemik	None	same VM	Master/proxy	No	No	No	No	Initial thread placement	Yes
JESSICA2	None	same VM	Master/proxy + flexible home	Yes	Yes	Yes	Yes	Object ownership migration	Yes
JavaParty	source/ new keyword	any standard JVM	Clustered classes in every node	No	Yes	Yes	No	None	No
JavaSymphony	None	any standard JVM	Explicit object mapping	Yes	No	Yes	No	Explicit object mapping	No
Addistant	bytecode level	any standard JVM	Master/proxy class-based	No	Partial, no cyclic garbage support	Yes	No	None	No
J-Orchestra	bytecode level	any standard JVM	Master/proxy	No	Partial, no cyclic garbage support	Yes	No	Object migration	No
JavaSplit	bytecode level	any standard JVM	Master/proxy + LRC	Yes	Yes	Yes	No	Initial thread placement	No
Terracotta	bytecode level	any standard JVM	Central Terracotta server+proxys	Yes	Yes	Yes	Yes	None	No

2.3 Clustering and thread scheduling

One of the problems considering clustering in distributed systems and software DSMs in particular is the scheduling of threads for maintaining a balanced system with a fair share load that minimizes communication and can make good enough decisions that give an acceptable performance in the long run. In this section, we will discuss a few algorithms and techniques to attempt to reach an ideal scenario where no nodes will be heavily loaded while other nodes are idle or only lightly loaded.

Load-distribution algorithms [43] can be classified in the following categories: static, dynamic and adaptive. Static algorithms are the most straight-forward approach, a new task is simply assigned to a node known a priori via a round-robin policy. An heterogeneous cluster might have an adapted weighted policy, assigning more tasks to the most powerful nodes and less tasks to the less powerful nodes, but no information about the current state of the system is used. Both these approaches have been tested in web clustering architectures [13]. Therefore, static algorithms can potentially make poor assignment decisions. For example, a new thread might be initiated in a node A, which is heavily loaded, while the local node B was idle, simply because node A was next in our fixed scheduling algorithm.

Dynamic algorithms attempt to improve the performance of their static counterparts by exploiting system-state information in runtime before making the decision. Because they must collect, store, and analyse state information they have more overheads and are harder to implement, but this extra overhead is usually compensated. In our idle node example, a dynamic algorithm could check that the node B where the new thread was initiated was idle and decide not to create the task at node A. The algorithm could even consider the state of the receiving node, possibly only assigning the new thread if the node was idle.

Adaptive algorithms are a special case of dynamic algorithms. Besides considering the system load, the system state itself can change the scheduling policies. For example, if a given policy performs better under a heavy-loaded system and another one performs better in a lightly-loaded system, an adaptive algorithm can use the former after a certain threshold of CPU load and the latter when the system reaches a lighter state, adapting itself to different workloads or application suites.

Both dynamic and adaptive algorithms raise an important issue: what is a "heavily-loaded node" and how can we define a metric that will allow us to determine if node A will be a good option for executing the next thread? Some authors like Kuntz [28] have defined the best metric as being the CPU queue length, and no significant performance was gained by using or combining other metrics such as the system call rate and the CPU utilization. In addition to the queue length metric, many authors proposed that a dynamic load-balancing system should have a priori knowledge of the resources needed by of the task in order to choose the best node. Since the node is chosen before the task is executed, the resource usage of a task must be predicted, either based on the past behavior of the task or by providing the load-balancing system with a user estimation. Both approaches are error-prone and can have a very negative impact if used with a wrong estimation.

Choi et all. [19] have proposed a novel metric to minimize the impact of inaccurate predictions. It is known that overlapping CPU bound and I/O bound jobs results in better resource utilization, so the number of tasks considered in the CPU queue length should consider its nature. For example, a node with three CPU bound and two I/O bound should be considered as having five tasks in its queue but only three *effective tasks*, since the CPU bound overlap with the I/O bound. However, there is still a need to classify a task as "CPU bound" or "I/O bound", which is not trivial to do, and the performance

improvements proved to be marginal compared to the historical-based approach.

Considering this, we can now take a deeper look to some scheduling algorithms and define what major considerations should they have to be time and space efficient. Two major requirements have been identified: good locality and low space [36]. The former means that threads that access the same memory pages should be scheduled to the same processor, as long as it is not overloaded, minimizing the overhead of page fetching, while the latter indicates that the memory requirements for the scheduling algorithm should be kept small to scale with the number of threads or processors, as in a cluster both numbers tend to grow overtime.

Work stealing schedulers [9] is a dynamic scheduling solution that provides a good compromise between the above requirements. Each processor keeps its own queue and when it runs out of threads it steals and runs a thread from another processor queue. This way, threads relatively close to each other in the computation graph are often scheduled to the same processor, providing good locality. The space required is at most S_1P, where S_1 is the minimum serial space required. This space bound can still be improved, as we will see in the next paragraph.

Depth-first search schedulers [8] is another dynamic scheduling approach. It works by computing a task graph as the computation goes by. A thread is broken into a new task by detecting certain *breakpoints* that indicate a new series of actions that can be performed in parallel by another processor (e.g. a fork). The tasks are then scheduled to a set of *worker* processors that hold two queues, one for receiving tasks (Q_{in}) and the other to put tasks created (Q_{out}), while the remaining processors are responsible to take tasks from the Q_{out} queues and schedule it to the Q_{in} queue of another processor. It was proven than the asymptotic space bound for this algorithm is $S_1 + O(p.D)$ for nested parallel computations of depth D, which is an improvement over the previous work-stealing approach. However, as the created tasks have a relative high probability of being related with the previous computation, the locality is not as good.

DFDeques [36] is a dynamic scheduling approach that seeks the best of both worlds. Threads are assigned to multiple ready queues that are depth-first ordered, similarly to the depth-first search schedulers seen in the previous paragraph. The ready queues are treated as LIFO stacks similar to the work-stealing schedulers. When a processor runs out of threads to run, it can steal from a ready queue chosen randomly from a set of high-priority queues. The asymptotic space bound is $S_1 + O(K.p.D)$, with K being a runtime parameter which specifies the amount of memory a processor may allocate between consecutive steals. As K is usually small, the space bound is about the same as in pure depth-first schedulers and at the same time we can take advantage of thread locality as threads close to each other in the computation graph will be scheduler to the same ready queue.

These algorithms have been widely studied and were used to introduce scheduling in many parallel programming libraries and applications. Satin [37] is a Java-based grid computing programming library that implements a work stealing approach by allowing a worker node to steal a method invocation from another node. Athapascan-1 [15] is a C++ library for multi-threaded parallel programming that implements a data-flow graph where both computation and data grains are explicit, allowing a depth-first scheduler algorithm to take advantage of the existing structure to schedule new threads. When considering applying one of these algorithms to a DSM system for general-purpose computations there are a few extra considerations that should be taken. We have to deal with heterogeneous nodes with different clocks and resources that may or may not be available at a certain time. This implies that a system should be dynamic and support some kind of migration of tasks to rebalance the load [50]. Also, DSMs have a much higher communication requirements than message-passing and, unlike parallel programming, we cannot predict easily the kind of applications that will run and what could be the

best parallelism possible. In the subsection 2.3.1 we are going to cover implementation issues of thread migration mechanisms and how much we can benefit from them and in subsection 2.3.2 we are going to cover another approach that takes advantage of system Virtual Machines to provide a more coarse-grained but easier to implement migration mechanism.

2.3.1 Thread Migration

In the previous section we studied thread scheduling in distributed shared memory systems in the perspective of initial placement of tasks, and the metrics that can be used for measuring the least loaded node. Besides the initial placement, transparent thread migration has long been used as a load-balancing mechanism to optimize resource usage in distributed environments [20]. Such systems typically use the *raw thread context* (RTC) itself as the interface between two nodes. The RTC consists of the thread virtual memory space, thread execution stack and hardware memory registers. This is the typical platform-dependent format used to represent a thread context and cannot be migrated directly without a few extra considerations. For example, the pointers or references used for data objects can be meaningless in another machine or the thread might be executing a system call that does not have any meaning in another node (e.g I/O calls). Some authors [29] have solved this issues by reserving all stacks in the beginning of the execution and guaranteeing that all of them use the same virtual addresses, and at the same time kernel threads are used to handle all system calls. This way, all threads are portable, considering an homogeneous system, but the number of threads in each node has a fixed limit.

In the particular case of software DSMs there are a few extra considerations that should be taken into account, as increased communication can exceed the benefits of better load-balancing. For example, the original pages that were cached by a thread before migration may not be needed anymore by the source node. In contrast, the destination node will definitely need those pages and the amount of communication needed to keep consistency among the processors implies a considerable overhead. Therefore, threads for migration need to be carefully chosen in such a way that the communication overhead does not exceed the benefit given by the better load-balancing.

One of the simplest solutions is to consider the number of shared pages between pairs of threads and assume that more shared pages implies a bigger communication cost if one of the threads migrates. However, not all data-sharing results in data consistency communication as two threads can simply read the same pages without any of them changing any data. Therefore, data-sharing policies should also consider the type of memory access.

In addition, an efficient thread selection policy needs to consider *global sharing* (i.e. the communication necessary with all processors). Although such a policy will result in a much more informed decision the cost of computing the thread migration cost for each thread increases linearly with the number of processors. Other solutions involve a *partial sharing* policy, which only considers communication cost between the source and the destination node, without regard to global relations, increasing the risk of making a wrong decision. Liang et al. [34] propose a novel thread selection policy called *reduction inter-node sharing cost* (RISC) for page-based DSMs that support release-consistency, which combines the type of memory access with a global sharing policy.

Concerning thread migration in Java systems, Java has a serialization mechanism that can capture an object state and restore it in other node running another virtual machine. However, the Thread class is not serializable and the standard JVM does not provide a mechanism to access a thread stack directly. Therefore, most existing Java solutions rely on the *bytecode-oriented thread context* (BTC) as

interface. The BTC is organized in a sequence of blocks called *frames*, each one associated with a Java method being executed by that thread. Each frame contains the class name, the method signature, and the activation record of the method. The activation record consists of a bytecode program counter (PC), which points to the Java instruction currently being interpreted, a JVM operand stack pointer for the stack that holds the partial results of the method execution, and the local variables of the associated method, encoded in a JVM-independent format. Considering this, and the fact that we still have to deal with threads executing native code in an RTC fashion due to system classes and JITs that compile bytecode at runtime, the following basic approaches were found in the literature [44]:

- **Static byte code instrumentation:** thread migration support is added by pre-processing the already compiled bytecode source and adding statements which backup the thread state in a special backup object. When an application requires a snapshot of a thread state, it just has to use the backup object produced by the code inserted by the pre-processor. The main advantage of this approach is that this way the thread migration can be implemented as a simple extension that manipulates bytecode, without the need to modify the JVM. Unfortunately, the fact that there are more bytecode instructions in the code introduces significant overhead and the thread state restoration requires a partial re-execution of the application. Some implementations of this approach for mobile agents such as AMO [42] also do not consider system classes or classes with native code and cannot migrate code that uses reflection.
- **Extending the JVM and its interpreter:** thread migration support is simply added as an extension to a normal JVM interpreter, as done in systems such as JESSICA [35]. This is accomplished by having a global thread space that spans the entire cluster and a mechanism that can separate the hardware-dependent contexts in native code and the hardware-independent contexts at bytecode level. This way, a thread can migrate with relatively good granularity between each bytecode instruction that is interpreted. However, modifying the JVM interpreter do deal with thread migration adds to the overhead of the already slow interpreter. This approach has been proven to have better performance than the previous static byte code instrumentation [10] but it is still much slower than the creation of a normal thread, which gives the impression that support for JITed code is needed.
- **Using the JVM Debugger Interface (JVMDI):** thread migration support is added by compiling Java applications with extra debugging information that allows access to the thread stack as well as the introduction of thread migration points. Modern debugger interfaces also support JIT compilers, as previous approaches only considered bytecode. However, JVMDI needs huge data structures and incurs large overhead to include the extra general debugging features and the limited optimizations that can be done in a debugging environment.

CEJVM [30] is a master-worker approach that relies on a master node that runs the Java application and delegates threads to worker nodes. It uses the JVMDI to implement thread migration transparently and compatible with any JVM that supports the debugger interface. Performance-wise, the master-worker paradigm only works well with a specific niche of applications and it would be desirable that all nodes be provided with thread migration capabilities in a point-to-point way.

Cho-Li et all [18] define a new approach that consists of integrating the RTC to BTC conversion and the implicit stack capturing and restoration directly inside the JIT. Stack capturing involves using the JIT to instrument native codes and transform them back into the platform-independent bytecode format. This way, the thread scheduler itself can perform on-stack scanning and to derive the BTC format instead

of using a stand-alone process like in the JVMDI approach. For stack restoring, the authors introduce a mechanism called *Dynamic Register Patching* that rebuilds the state of the hardware registers before returning the control to the thread instanced in the new node.

Another issue that we need to address is at which code points should migration be considered as a good option. The simplest approach is to allow migration in any bytecode boundary. However, with all the JVMs running sophisticated JITs there is a high probability that the execution is running native code at the time of migration and it may be very hard and inefficient to simulate the native instructions from the stopped point until the next bytecode boundary for migration. Cho-Li et all [18] define two basic points: the beginning of a Java method invocation and the beginning of a code block pointed by a back edge in the computational graph. The former indicates a new operation that can most likely be done in another node (very small methods that do not typically compensate will be inlined by the compiler and not considered for migration), while the latter represents the beginning of a loop, which is also a good option as it needs a more or less prolonged computation until it finishes. Intra-bytecode migration semantics would be very ambiguous and difficult to implement, so we are not considering it in this report. It is preferable, for example, to turn off the JIT compiler before migration and only enable it after scheduling on another node.

Finally, we have to deal with the type resolution of the operands. As operands in a thread context are pushed in and popped out of the stack at runtime, their types cannot be determined in advance. The simplest solution is to have a separate stack for operand types synchronized with the normal Java stack. This doubles the time in accessing the operand stack, which can be more or less significant depending on the number of possible migration points that we are considering. Also, this approach can be optimized as most types can be verified statically by the Java bytecode verifier [24]

2.3.2 Virtual Machine Migration

The need to provide a cluster to support multiple operating systems, applications, and heterogeneous hardware has led to the development of Virtual Machine Monitors (VMM) or hypervisors that run right on top of the hardware and schedule one or more operating systems across the physical CPUs. The live migration mechanism is less granular than thread migration, as a system VM might have a large number of threads running simultaneously and the migration of an entire system VM to another node requires that the recipient node has indeed more resources to run the system VM. However, a recent performance study made by Chen et all. [17] using a page-based DSM system shows that the virtual machine migration approach can compete with thread migration and it has the advantage of providing a cleaner separation between hardware and software, as well as facilitating fault-tolerance and load-balancing.

IBM have developed the z/VM solution [38], an hypervisor software capable of supporting several thousands of Linux servers running on a single mainframe. z/VM supports full scheduling of user virtual machines according to each user needs by monitoring resource usage and giving a user class from 0 to 3. Higher class users get longer time-slices but lower classes tasks are given a higher priority if the mainframe resources get constrained. Despite the good transparency and scheduling solution, the system only runs on the mainframe zSeries IBM servers, which are not available to the majority of programmers.

Xen [6] is another hypervisor that runs on standard x86 machines, developed in the University of Cambridge. Xen supports many popular operating systems such as Solaris, Linux and Windows. System administrators can migrate Xen Virtual Machines between physical hosts across a LAN without loss of availability

2.4 Summary

In this chapter, we discussed the most important research topics related to our work. We started by presenting an historical background of the Distributed Shared Memory (DSM) concept, along with the consistency models required to make it work. We also gave examples of academic software DSMs and took a closer look at STMs. We concluded that all these prototypes imply a different programming approach that is impractical and, although the concept is interesting, a more transparent approach is needed.

After this discussion of DSMs, we presented a survey on distributed Virtual Machines, divided into three categories: Compiler-based DSMs, Cluster-aware Virtual Machines and Systems using standard VMs. We concluded that the first type is limited to a specific architecture and require the cluster to be homogeneous. The second type also sacrifices cross-platform compatibility by requiring all nodes in a cluster to run the same VM, cross-platform compatibility, and most do not implement already existing JVM facilities. The third type is the one most likely to succeed, due to the Sun JVM being present in the most common architectures and operating systems

To finalize this chapter, we presented several thread scheduling algorithms, along with thread and virtual machine migration techniques. The techniques for thread scheduling in distributed environments covered here can be integrated with a system that already provides a shared object space, giving common programmers the ability to run a regular multi-threaded application in a cluster seamlessly, without worrying about load-balancing.

Chapter 3

Architecture

This chapter describes the middleware **Caft** (Cluster Abstraction for Terracotta), developed during this work to allow Terracotta to run simple Java multi-threaded applications with minimum changes or concerns due to the different environment. We will start by familiarizing the reader with the mechanisms already offered by Terracotta that motivate it to be a very good choice for clustering application servers such as Tomcat or JBoss. After this introduction, we will describe the high-level architecture of the middleware, as well as all compromises assumed. To finalize, we are going to present the packages and classes that compose the middleware, along with a description of their functions and data structures used.

3.1 Terracotta

In this section, we are going to introduce the main concepts of the Terracotta platform. This should give the reader a little background on how to configure Terracotta for clustering some application, which is essential for understanding how our middleware works on top of Terracotta for providing greater transparency for running simple multi-threaded Java applications, the primary use case that motivates this work. The main concepts of Terracotta are presented in the list below:

- **Clients and Servers**: The Terracotta developers adopt the client/server terminology and call the application JVMs that are clustered together *Terracotta clients* or *Terracotta cluster nodes*. These clients run the same application code in each JVM and are clustered together by injecting cluster-aware bytecode into the application Java code at runtime, as the classes are loaded by each JVM. This bytecode injection mechanism is what makes Terracotta transparent to the application.

  Part of the cluster-aware bytecode injected causes each JVM to make a TCP connection to *Terracotta server instances*. In a cluster, a Terracotta server instance handles the storage and retrieval of object data in the shared clustered virtual heap. The server instance can also store this heap data on disk, making it persistent just as if it were part of a database. Terracotta server instances exist as a cohesive array. This works by having one server as a master, holding the entire heap, and another server as a passive mirror, acting in case the master stops responding. In the enterprise version of Terracotta, it is also possible the split the global heap across several servers, each one holding a part of the heap. To illustrate this first basic notion, we present an example of a typical Terracotta cluster in Figure 3.1.

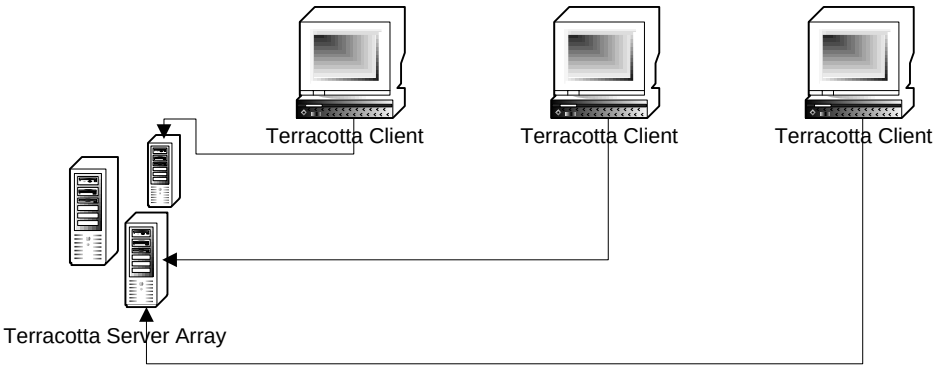

Figure 3.1: A typical Terracotta cluster, composed of Terracotta clients and servers

- **Distributed Shared Objects (DSO)**: In a single JVM, objects in the heap are addressed through references. The Terracotta clustered virtual heap objects are addressed in a similar way, through references to clustered objects which we refer to as *distributed shared objects* or *managed objects* in the Terracotta cluster. To the application, these objects are just like regular objects on the heap of the local JVMs, the Terracotta clients. However, behind the scenes, Terracotta knows that clustered objects need to be handled differently than regular objects.

  When changes are made to a clustered object, Terracotta keeps track of those changes and sends them to all Terracotta server instances. Server instances, in turn, make sure those changes are visible to all the other JVMs in the cluster as necessary. This way, clustered objects are always up-to-date whenever they are accessed, just as they are in a single JVM. Consistency is assured by using the synchronization present in the Java application, which turns into Terracotta transaction boundaries.

  Not every object in the JVM is a clustered object, only those that the developer chooses to share. This works by declaring a field to be *root*, which is going to be described in more detail in the configuration item.

- **Configuration**: As Terracotta clusters JVMs transparently with no explicit API, control over what gets clustered and which operations in the application are sensitive to clustering is performed through the Terracotta configuration. The configuration can be specified using an XML file or using code annotations, depending on the developers personal preference. The three main sections of the Terracotta configuration that must be specified by the developer are: roots, locks, and classes to instrument.

  - **Root**: A root defines a field to be put in the global heap and shared across all JVMs, maintaining object identity. A root is what forms the top of a clustered object graph and allows Terracotta to distinguish which objects are shared and which are not.
  - **Locks**: Access to shared roots need to be locked in Terracotta, in order to guarantee proper data consistency. It is the only allowed way to access shared objects in Terracotta. There are two types of locks:
    - **Auto locks:** allow Terracotta to use already existing synchronization present in the methods that access shared objects, whether it is an advanced data structure such as a ReadWriteLock or just plain old `synchronized` keyword. If no synchronization is present, Terracotta also offers an *auto-synchronized* mode, that behaves just as if the method had the `synchronized` keyword.

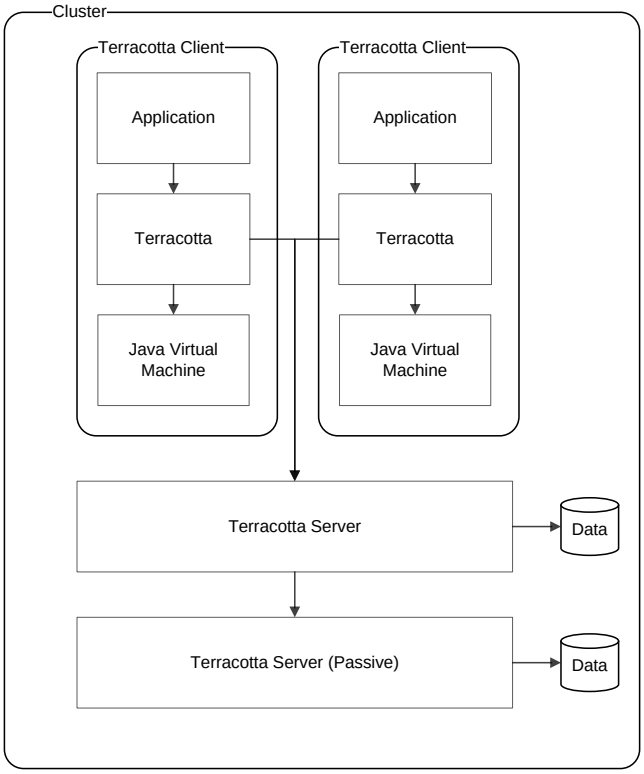

Figure 3.2: Terracotta architecture

  * **Named locks:** allow the definition of a global lock across the cluster, making sure that only one JVM executes the method at a current time. This lock type should only be used as a last resort when there is no access to the source code and the autolock is not enough. Compared to the autolock type, the latter uses the object instance as a lock identifier, thus providing significantly more fine-grained locking, lower lock contention, and thus higher performance.

- **Instrumented Classes:** Classes that access shared roots, or are shared themselves in the global heap, need to be instrumented at bytecode level to guarantee that Terracotta applies modifications and adds proper locking. Each class instrumented adds a bit of overhead, even if the instrumentation was not needed, so the developer should instrument only the classes that are really necessary.

For better understanding, we present in Figure 3.2 a cluster with two Terracotta clients, each running the application on top of Terracotta with a standard Java Virtual Machine. The Terracotta clients propagate modifications in shared objects to a Terracotta Server, which can propagate to another Server that serves as backup. In addition, shared objects can be made persistent.

By defining these concepts, and running the application in the Terracotta infrastructure, it is possible to cluster data structures and allow for good scalability. However, the current version of Terracotta holds the following limitations:

- Threads created never leave the home node. It is possible to adapt the Master/Worker paradigm with a Terracotta add-on but it implies that the programmer needs to use a special distributed

executor service, which has a different interface than the Thread class and may imply a large refactor at source code level.

- Adapting an existing application implies that the programmer needs to add synchronization where needed, which in case of a large application can be troublesome. As the *auto-synchronized* offers synchronization only at method calls, it might not be fine grained enough and can lead to an incorrect semantic in the application execution, resulting in deadlocks.

In the next section, we are going to introduce the high-level architecture of the middleware developed, in order to minimize the limitations above.

3.2 Caft - a middleware that extends Terracotta

The Caft middleware has two major components: **worker** and **master**. The former runs a Thread Service that provides the interface for instantiating new threads, as well as the operations provided by the Java Thread Class (whose methods can be regarded as an implicit interface), while the latter runs the main class of a runnable Jar containing a multi-threaded application, spawning threads in worker machines as necessary. It is assumed that the Jar is available on both the master and the workers.

Both master and workers need to share the thread fields whose identity must be preserved across the cluster and its changes propagated. The master opens the Jar passed as argument, detects the class defined as the main entry point and runs the main method using the Java reflection API. The master uses a custom Classloader, also present in the worker, that applies the instrumentations necessary to make the Thread calls cluster aware, and/or adding synchronization. Bytecode instrumentations are made using the ASM framework [11], allowing us to add methods and changing calls without much overhead. For the master and worker communication, we use simple RMI calls supported by the Spring framework [1] to ease development and configuration.

To simplify the implementation, the coordinator component that decides which node gets to execute the next thread is integrated as a singleton in both components. The data structures that compose the state of the coordinator, such as which nodes are available and their loads, are maintained as roots in Terracotta's Distributed Share Objects (DSO) space. This approach also avoids the need to have an extra node that serves as a coordinator and the persistence of its state is guaranteed by the Terracotta Server.

To better illustrate our design, we present the Terracotta architecture in Figure 3.3 with the Caft middleware, running a worker in one of the Terracotta clients and a master in another. The middleware runs on top of Terracotta, loading the application and performing bytecode instrumentations at load time. If configured for running a worker, Caft will start an RMI service using the Spring framework, keeping the Java application in its own class path to ensure everything works when it receives a Runnable target to execute in it. If configured to run a master, it will simply run the application, just as already described.

Considering that we need to have a trade-off between transparency and performance, as less transparency should allow for better customization and tuning, we developed Caft with three different modes. The mode to be used is passed as an argument to both master and workers, and they should not be mixed. The modes supported are presented in the list below:

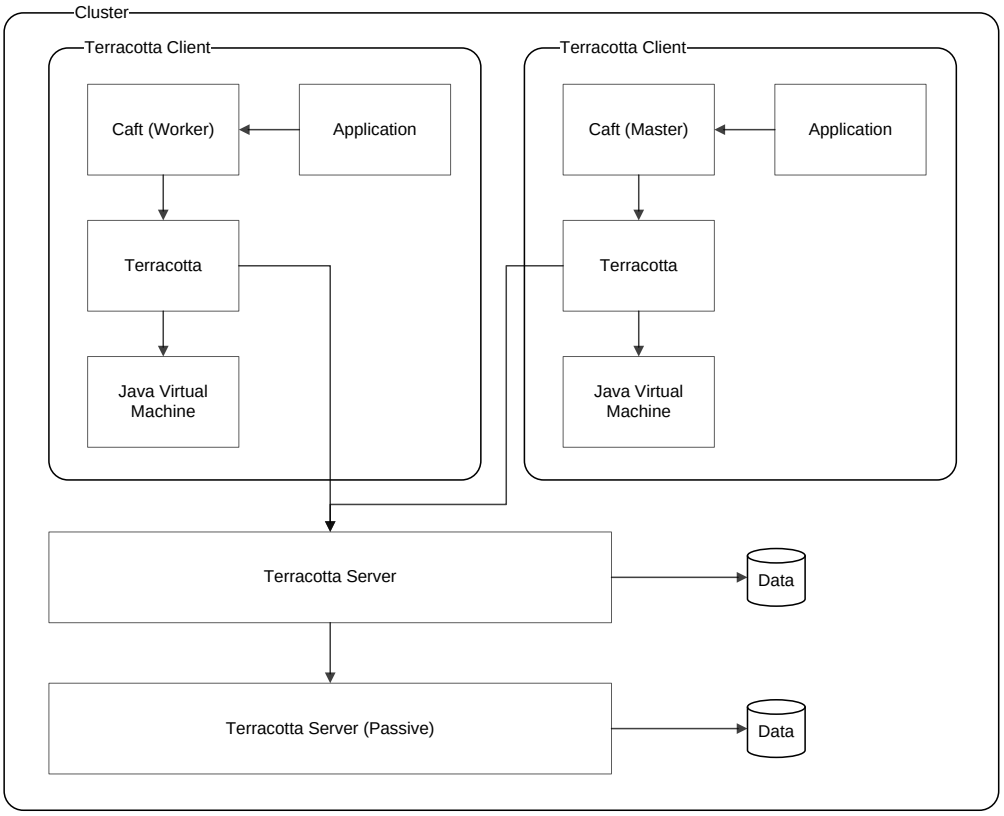

Figure 3.3: Terracotta architecture running Caft

- **Identity:** Identity mode assumes that the application is properly synchronized using the Java synchronized keyword or more advanced structures like `ReentrantReadWriteLock`, or at least, that the user has access to the source code and can add synchronization with more or less work. All thread fields are shared in the Terracotta DSO to ensure that the writes are propagated and all methods are annotated with the `AutolockWrite` Terracotta annotation, so that each synchronized block can be converted into a Terracotta transaction. It should be noticed that without synchronization present the `AutolockWrite` annotation does not have any extra side effect, meaning that concurrency should not be affected.

- **Full SSI (Single System Image):** Full SSI mode assumes that the application lacks proper synchronization for usage with Terracotta, or the source code is not available. Full SSI behaves just like Identity mode but with extra instrumentations that add getters and setters to each field, with proper synchronization, and it also synchronizes array writes. This adds an extra overhead that will depend a lot on the kind of application. For example, an application that manipulates large arrays using loops will have one transaction per write, resulting in many transactions and communication with the Terracotta Server Array. It could be more efficient to simply put a synchronized block around the entire loop and generate only one transaction. However, one large transaction can also take too much memory and imply a large overhead, so a compromise in the middle is usually the best option.

- **Serialization:** Serialization mode allows the user to decide which fields of the Runnable class to be run in a Thread are meant to be clustered and have identity preserved, and the rest are simply serialized and copied via RMI, allowing for local thread variables that do not really need synchronization. This mode should be carefully used and is the least transparent of all, but it is

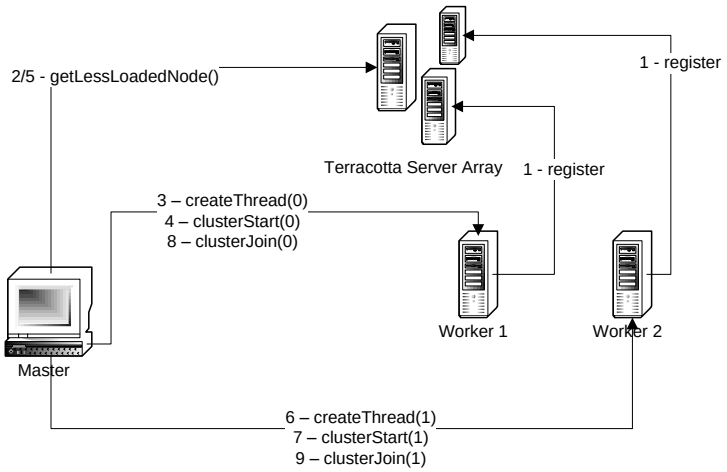

Figure 3.4: Terracotta deployment scenario with Caft

the one with the biggest potential for increased performance as we shall see in chapter 5.

As an example, consider the following deployment scenario, illustrated by Figure 3.4, with two worker machines that will receive Java Runnable targets and use them to create local threads, one master that will run the application, and a Terracotta Server Array holding the DSO.

In this example, the master is running a multi-threaded application that launches two threads and waits for them to finish. The worker machines register themselves with the Coordinator in step one, whose state is shared in the Terracotta Server Array in order to be accessible by the other machines. The master checks the coordinator state in step two, which determines that **Worker 1** is the less loaded node (at this point, could be either of them as both never had any thread assigned). In step three, the master sends the thread ID to that worker and makes the Runnable target available to it, either copying it via RMI or putting it in the Terracotta DSO, depending on the mode used. The worker creates a local instance of a Java Thread using the Runnable target, which is started also by a remote call of the master in step four. The master will then attempt to create another thread, which after consulting the Coordinator in step five it returns the node **Worker 2** as being the most appropriate, as the master already assigned a thread to Worker 1. A thread is created and started in Worker 2 in steps six and seven, analogous to the first thread created in Worker 1. After this, the master joins both threads, illustrated by steps eight and nine, making an RMI call to the workers which will execute a local join in the Java Thread object corresponding to the thread. As we are using the DSO in Terracotta, the master will see every relevant change in the objects passed as a Runnable target to both threads.

This concludes the section which describes Caft architecture and its high-level components. In the next section, we are going to describe the code modules in which the middleware was decomposed, together with its classes and data structures for better understanding.

3.3 Caft module decomposition and structure

This section describes in further detail the module decomposition and structure of the Caft middleware. We are going to describe the packages and classes that implement the several modules of the middleware, as well as relevant data structures that compose the middleware state.

3.3.1 Package list

This subsection presents the packages that compose the middleware, dividing the implementation in several modules for better understanding and organization. During development, we adopted the convention that every package belonging to the middleware should be a subpackage of `org.terracotta.caft`. As such, the `org.terracotta.caft.common` package holds classes that are used by both the master and worker components. The package `org.terracotta.caft.common.asm` contains the bytecode class and method transformations that are used in the middleware, for thread creation and implementing the several modes described in the previous section. The `org.terracotta.caft.coordinator` package holds the Coordinator class, a singleton present in both the master and worker components. Finally, the `org.terracotta.caft.master` and `org.terracotta.caft.worker` packages hold the classes that compose the master and worker components. The following list summarizes all packages of the middleware:

- org.terracotta.caft.common
  - org.terracotta.caft.common.asm
    - org.terracotta.caft.common.asm.caftroot
    - org.terracotta.caft.common.asm.fullssi
    - org.terracotta.caft.common.asm.thread
- org.terracotta.caft.coordinator
- org.terracotta.caft.master
- org.terracotta.caft.worker
  - org.terracotta.caft.worker.service

3.3.2 Common package

The `org.terracotta.caft.common` package holds the classes shared by both the master and worker components. This includes the stub `ClusterThread` class, as well as `CaftClassLoader`, the custom Classloader used to apply the bytecode instrumentations at load time. The `ClasspathHacker` is used to add the application jar to the classpath of both the master and worker, to ensure that every class is visible when loading using RMI.

In the ClusterThread class, we share a map on the Terracotta DSO which is accessible by both the master and the workers, named **runnableTargets**. The runnableTargets is a Java `ConcurrentHashMap` using long thread IDs as keys and `Runnable` targets as values. This map is used by both Identity and FullSSI modes for storing and sharing the thread context to be used on the worker nodes to create a new thread.

To finalize, diagram 3.5 illustrates the classes belonging to this package.

Asm package

The `org.terracotta.caft.common.asm` package holds all the ASM bytecode instrumentations that are made by the middleware. For better organization and understanding, we split the instrumetantions into

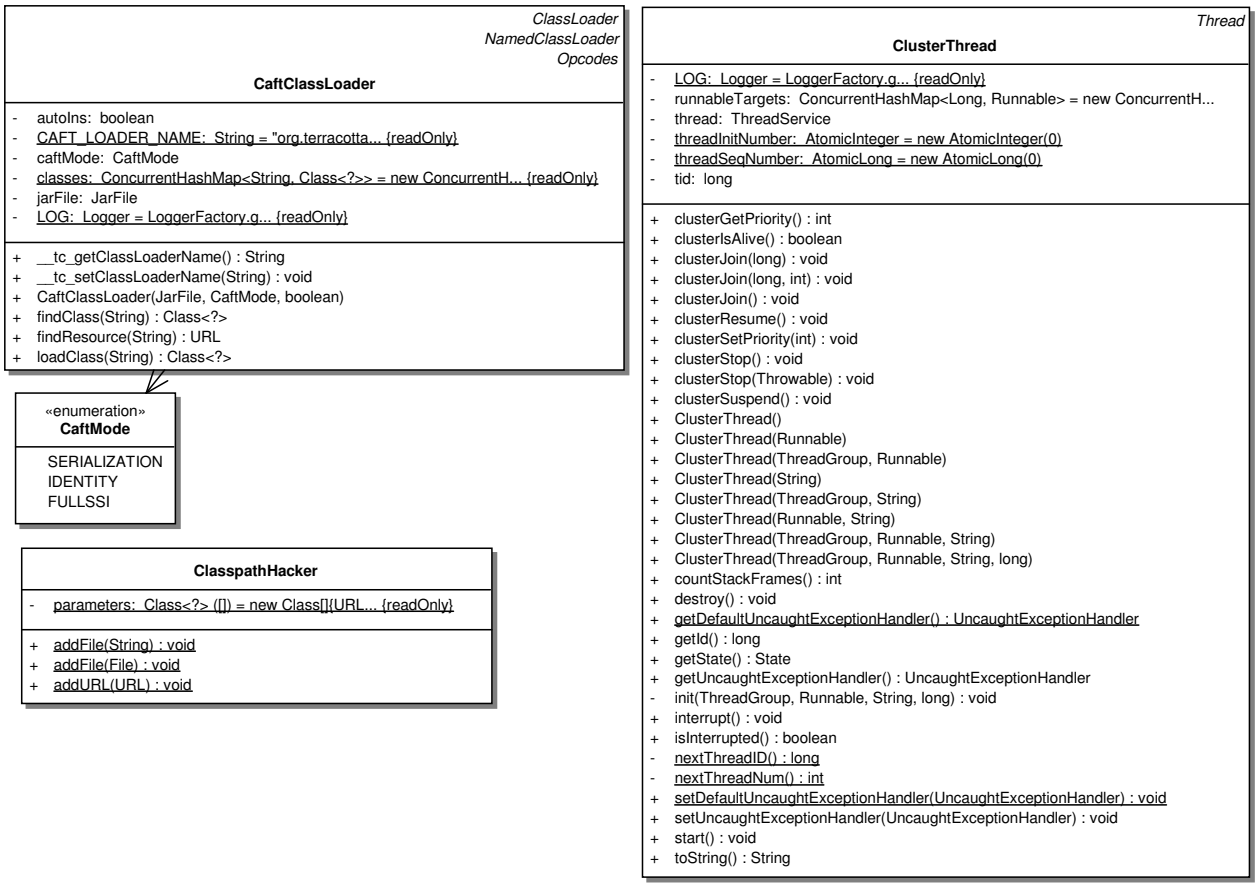

Figure 3.5: Common package class diagram

three subpackages: `org.terracotta.caft.common.asm.caftroot`, `org.terracotta.caft.common.asm.fullssi` and `org.terracotta.caft.common.asm.thread`. The `org.terracotta.caft.common.asm.caftroot` subpackage holds classes only relevant to Serialiaztion mode and the management of fields annotated with the CaftRoot annotation, while the `org.terracotta.caft.common.asm.fullssi` holds classes only relevant to the Full SSI mode. The `org.terracotta.caft.common.asm.thread` package holds the instrumentations that implement all the indirections necessary to use the new ClusterThread class present in the common package instead of the regular Java Thread class.

- **Caftroot package**:

  The `org.terracotta.caft.common.asm.caftroot` package contains the `CaftRootMap` and the `CaftRootAdapter` classes. The former holds the data structures and methods for implementing the sharing of fields annotated with the `CaftRoot` annotation, while the latter is an ASM method instrumentation for replacing access to CaftRoot fields with the static methods defined in the `CaftRootMap` class. The UML diagram of this package is presented in Figure 3.6.

- **Fullssi package**:

  The `org.terracotta.caft.common.asm.fullssi` package contains classes relevant to the Full SSI mode of Caft. The package contains the following classes: `FieldInfo`, `GetterAdder`, `SetterAdder`, `StaticGetterAdder`, `StaticSetterAdder`, `StaticArraySetter` and `GetterSetterAdapter`. The FieldInfo class is a simple container that holds field information relevant to the other classes. The GetterAdder and SetterAdder are ASM class adapters that add getters and setters for each non-static field, while the StaticGetterAdder and StaticSetterAdder add getters and setters for each

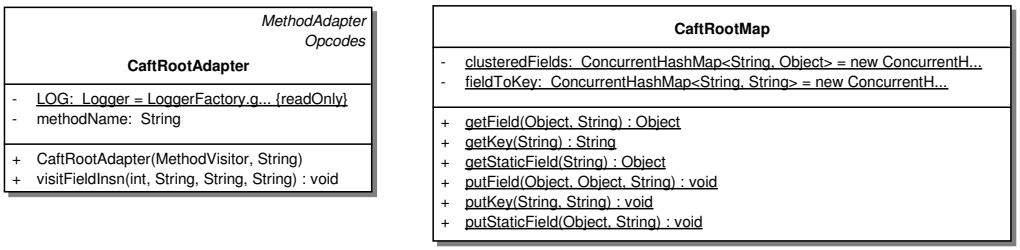

Figure 3.6: Caftroot package class diagram

static field. It should be noticed that each getter or setter is properly synchronized and annotated with Terracotta annotations for turning the synchronization into a Terracotta transaction. The StaticArraySetter is a collection of static methods that put a value of a certain primitive or reference type in a specific array position, wrapped in a synchronized block and annotated with the Terracotta `AutolockWrite` annotation. Finally, the GetterSetterAdapter is an ASM method adapter that replaces direct field access with the corresponding getter and setter calls, and also replaces array writes with static method calls of the StaticArraySetter class. The UML diagram of this package is presented in Figure 3.7.

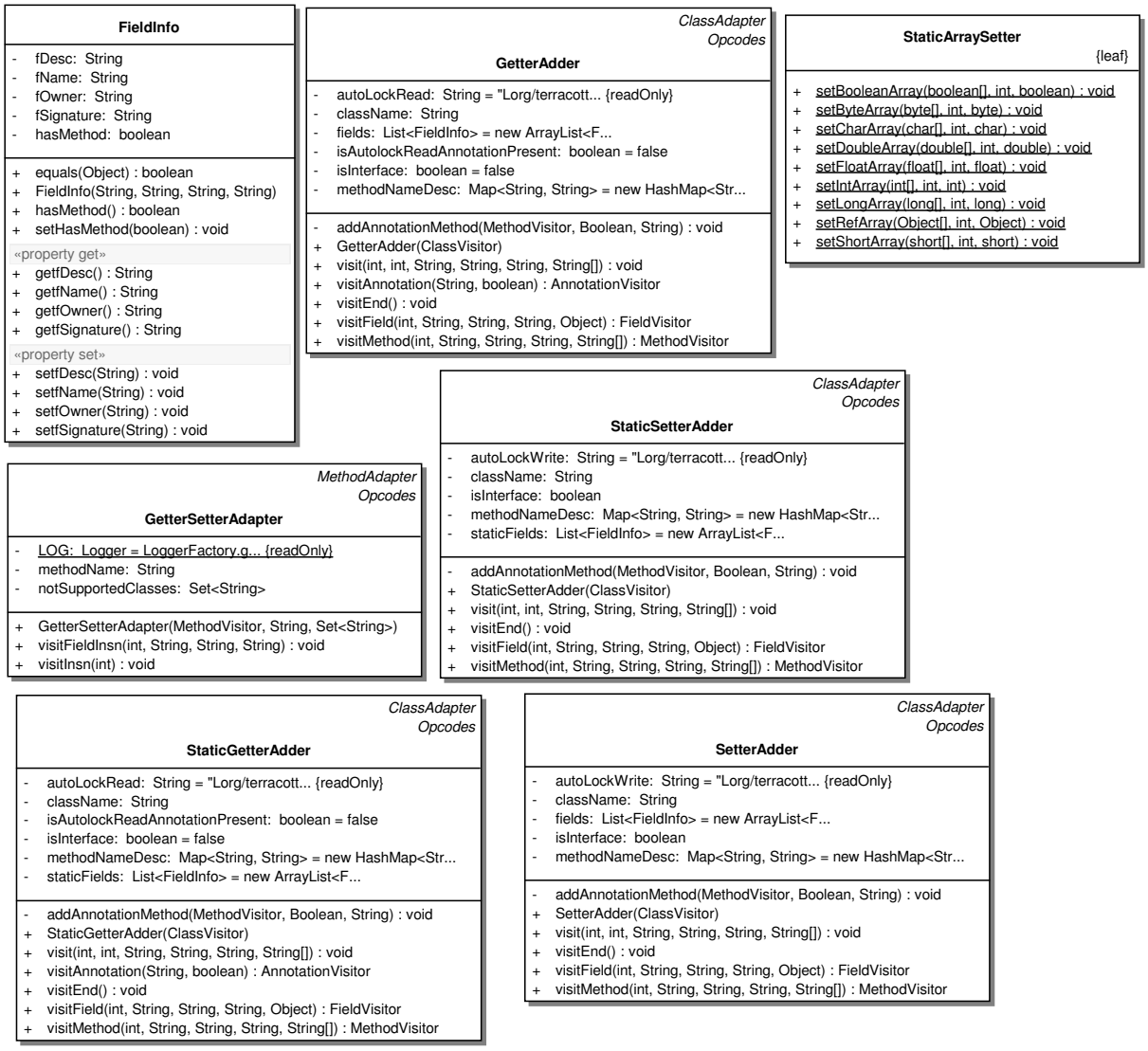

Figure 3.7: Fullssi package class diagram

- **Thread package**:

  The `org.terracotta.caft.common.asm.thread` package contains the `AddClusterThreadAdapter` and the `ThreadClassAdapter` classes. The former is an ASM method instrumentation that replaces the Java Thread class instantiation and method calls with instantiation and calls of the ClusterThread class, in the common package. The latter is an ASM class adapter that applies the AddClusterThreadAdapter instrumentation and adds Terracotta annotations for proper synchronization, depending on the mode chosen. The UML diagram of this package is presented in Figure 3.8.

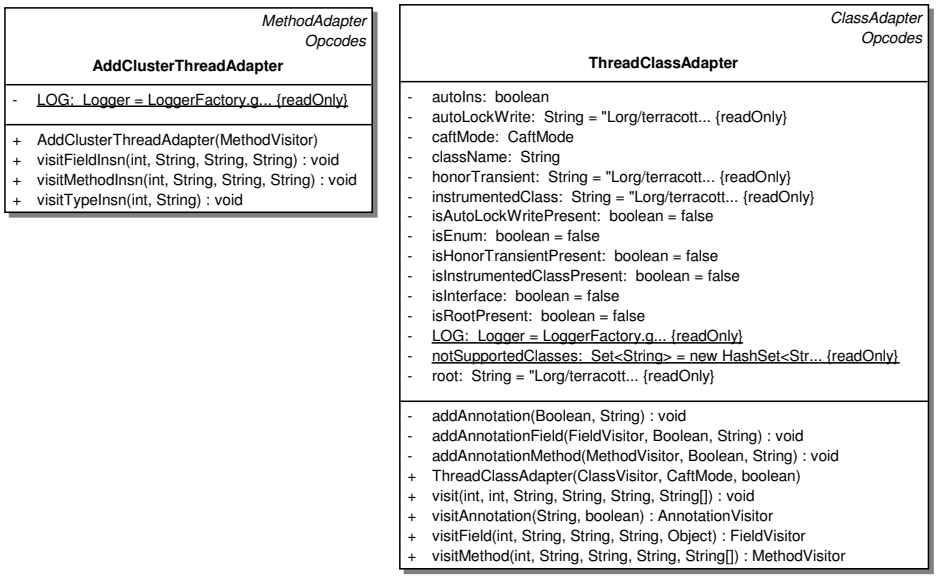

Figure 3.8: Thread package class diagram

3.3.3 Coordinator package

The `org.terracotta.caft.coordinator` package holds the `Coordinator` class, accessible by both the master and the worker components. The coordinator shares several data structures in the Terracotta DSO for keeping its state accessible to all nodes in the cluster. The data structures that compose the coordinator state are as follows:

- **tidNodes**: The tidNodes is a Java `ConcurrentHashMap` using Long thread IDs as keys and Strings corresponding the worker nodes address as values. This allows the coordinator to keep track of the worker nodes where a thread with a certain id was instantiated.

- **nodesLoad**: The nodesLoad is Java `ConcurrentHashMap` using String worker nodes addresses as keys and an Integer correspoding to the current number of threads in that node as value. This allows the coordinator to keep track of the number of threads assigned to each node.

- **loadNodes**: The nodesLoad is Java `ConcurrentHashMap` using Integers corresponding the a number of threads as keys and a Set of Strings corresponding to nodes as values. This map acts as the reversal of the previous map, allowing the coordinator to keep track of all the nodes that have a certain number of threads already assigned.

To finalize, the diagram in Figure 3.9 illustrates the classes belonging to the Coordinator package, using the Singleton pattern.

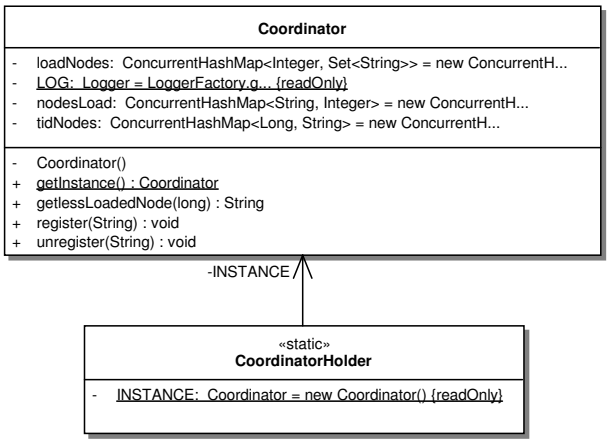

Figure 3.9: Coordinator package class diagram

3.3.4 Master package

The `master` package contains the `StartMaster` class, which holds the main method that loads the application jar and starts the application. We use the args4j framework to easily parse command line options for running the middleware. The options available are presented in the following list:

- **–mode**: chooses the Caft mode in which the master will run: SERIALIZATION, IDENTITY or FULLSSI
- **–jar**: chooses the .jar file containing the Java application to be run on Caft. The .jar should be executable.
- **–autoIns**: instructs the middleware to add Terracotta `InstrumentedClass` annotation to every class loaded, instead of having to configure it using the .xml file.
- **–args**: passes an arbitrary number of arguments, corresponding to the arguments to be passed when executing the .jar main method chosen.

To finalize, the UML diagram of this package is presented in Figure 3.10.

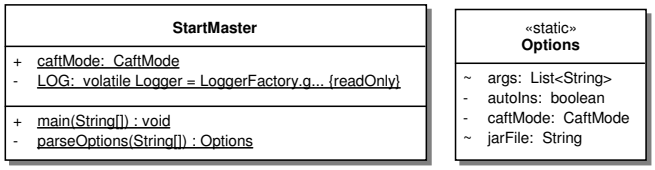

Figure 3.10: Master package class diagram

3.3.5 Worker package

The `org.terracotta.caft.worker` package contains the `StartWorker` class, which holds the main method that starts the RMI ThreadService and waits for requests from the master and the `ThreadServiceImpl`

class that implements the `ThreadService` interface. The ThreadServiceImpl class holds a Java `HashMap` named **threadPool**. The threadPool map uses `long` thread ids as keys and instances of the Java `Thread` class as values. This is a local map present in every worker and is used for storing and retrieving the concrete Thread instance for performing an operation requested by the master, such as to start a thread, wait for it to terminate, and so on.

The ThreadServiceImpl class also holds a `ConcurrentHashMap` named runnableTargets, which is considered by Caft to be the same root as the runnableTargets present in the ClusterThread class in the common package, allowing the sharing of Runnable objects in the Terracotta DSO. These same objects will be the ones used for creating local Thread objects in each worker node.

To parse the command line options for the worker, we use the args4j framemork, similar to the master. The options available are presented in the following list:

- **–mode**: chooses the Caft mode in which the worker will run: SERIALIZATION, IDENTITY or FULLSSI
- **–jar**: chooses the .jar file containing the Java application to be run on Caft. The .jar should be executable.
- **–hostname**: hostname of the machine in which the worker will run and host the Thread Service.
- **–port**: port of the machine in which the worker will wait for connections from the master.
- **–autoIns**: instructs the middleware to add Terracotta `InstrumentedClass` annotation to every class loaded, instead of having to configure it using the .xml file.

To finalize, the UML diagram of this package is presented in Figure 3.11.

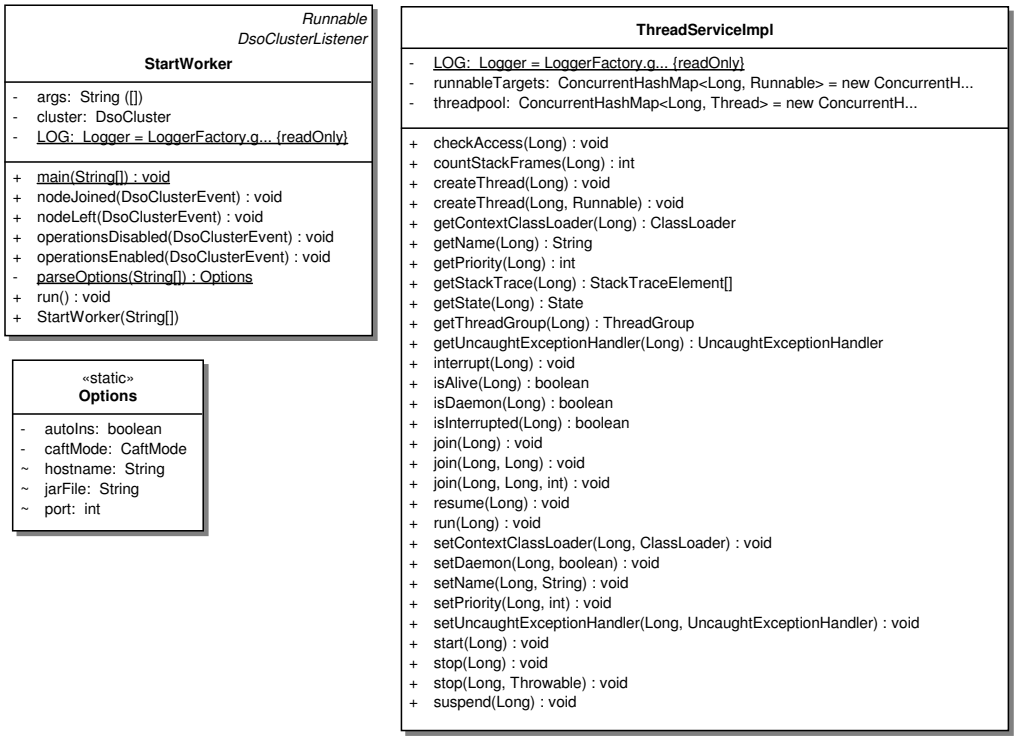

Figure 3.11: Worker package class diagram

Service package

The `org.terracotta.caft.worker.service` package contains the `ThreadService` interface, implemented by the ThreadServiceImpl class in the `org.terracotta.caft.worker` package. We also provide an equivalent `RemoteThreadService`, in case we wanted to use another remoting technology that requires an interface that extends Java `Remote` using the Spring framework. The UML diagram of this package is presented in Figure 3.12.

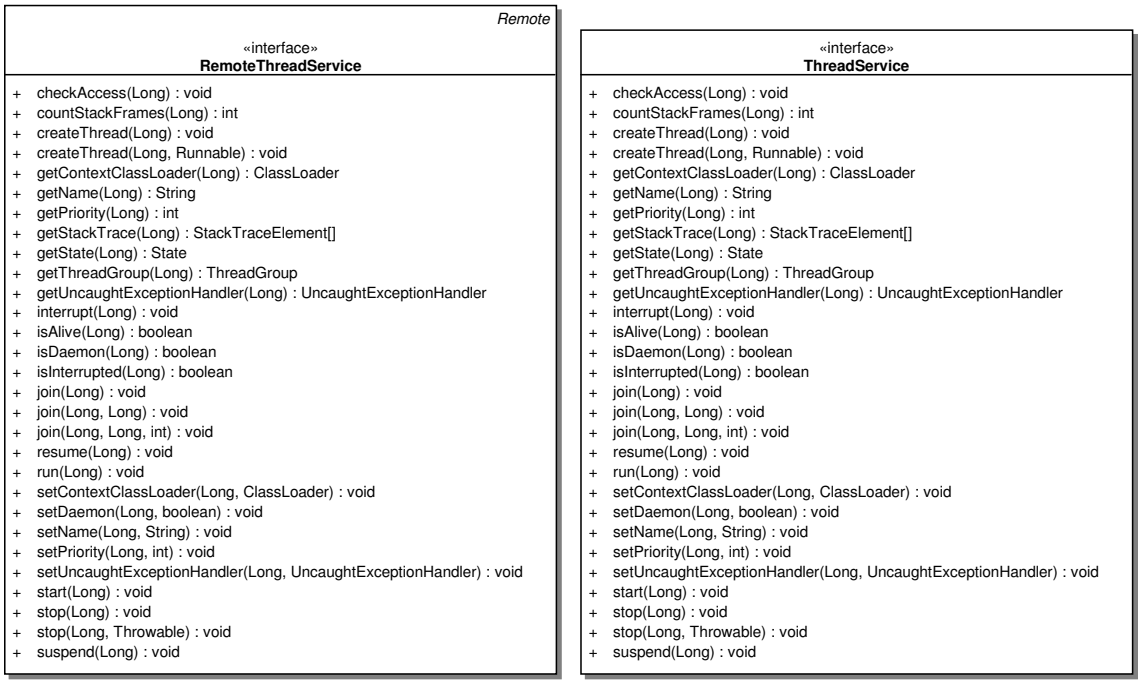

Figure 3.12: Worker service package class diagram

3.4 Summary

This chapter describes the global architecture of the Caft middleware, implemented during the course of this work. We presented a high-level view of the Terracotta middleware, necessary to understand the main concepts of Terracotta. We also described the high-level view of Caft middleware, along with an example of what happens when a Thread is created in a master and how it will converge to a real thread in a remote worker node. We then described the module structure and the several packages that compose it, along with the classes in each one. In the next chapter, we are going to describe implementation details and bytecode instrumentations in larger detail, ending the middleware description.

Chapter 4

Implementation

This chapter describes the main bytecode instrumentations and compromises done in the implementation of the **Caft** middleware. We will start by describing in detail the instrumentations implemented for allowing a thread to be created and started in other nodes. We will then proceed to describe the instrumentations that attempt to automatize the synchronization needed for an existing application, used by the Full SSI mode of the middleware. Finally, we will focus on the instrumentations used by the Serialization mode that offers a more fine grained way to cluster thread fields.

4.1 Thread instrumentations

This section describes the instrumentations developed for clustering threads. We instrument the application classes using the ASM framework [11]. As mentioned in the architecture chapter, we developed a method adapter named `AddClusterThreadAdapter` for implementing the indirections necessary for replacing Java Thread instantiations and method calls with our special ClusterThread class. We also developed a class adapter named `ThreadClassAdapter` that applies method adapters and adds annotations, depending on the mode chosen by the user.

4.1.1 AddClusterThreadAdapter

The `AddClusterThreadAdapter` instrumentation replaces Java type opcodes that have the Java Thread type as argument with equal opcodes with the ClusterThread type. This includes the `new` opcode for creating simple instances of objects, `a newarray` opcode for creating arrays with a reference type, and the `checkcast` opcode for performing type casts. The instrumentation also replaces the `getfield` and `getstatic` opcodes type with ClusterThread instead of Thread. As the ClusterThread class extends the original Java Thread class, type compatibility is guaranteed. It should be noticed that threads that are created extending the Thread class always remain in the home node, as a Thread object is not portable for Terracotta. This may be seen as a limitation, but on other hand, this approach allows for both clustered and local threads to coexist.

For the method calls, some of the methods belonging to the Thread class are final, and therefore cannot be overridden. To circumvent this, we renamed the final methods and replaced Thread method

calls with the renamed method. For example, if we have an `invokevirtual` opcode that invokes the final "join" method of the Thread class, we invoke the "clusterJoin" method instead.

To illustrate this method adapter, we now present a series of bytecodes corresponding to a Java Thread being created, started, and joined.

```
1  new java/lang/Thread //creates a new object instance of the Thread class
       and puts it on the stack
2  dup //duplicates the last object reference on stack
3  aload 1 //Push the object reference corresponding to the Runnable target to
        be passed to the constructor
4  invokespecial java/lang/Thread.<init>(Ljava/lang/Runnable;)V //invokes the
       constructor of the Thread class that receives a Runnable target as
       parameter
5  astore 2 //Stores the new object on variable 2 of the stack
6
7  aload 2 //Push the object reference correspoing to the instante of the
       Thread class
8  invokevirtual java/lang/Thread.start()V //invokes the start method of the
       Thread class
9
10 aload 2 //Push the object reference correspoing to the instante of the
       Thread class
11 invokevirtual java/lang/Thread.join()V //invokes the join method of the
       Thread class
```

After applying the instrumentation, the bytecodes would be replaced by the following sequence:

```
1  new org/terracotta/caft/common/ClusterThread //creates a new object
       instance of the ClusterThread class and puts it on the stack
2  dup //duplicates the last object reference on stack
3  aload 1 //loads the Runnable target to be passed to the constructor
4  invokespecial org/terracotta/caft/common/ClusterThread.<init>(Ljava/lang/
       Runnable;)V //invokes the constructor of the ClusterThread class that
       receives a Runnable target as parameter
5  astore 2 //Stores the new object on variable 2 of the stack
6
7  aload 2 //Push the object reference corresponding to the instance of the
       ClusterThread class
8  invokevirtual org/terracotta/caft/common/ClusterThread.start()V //invokes
       the start method of the ClusterThread class
9
10 aload 2 //Push the object reference corresponding to the instance of the
       ClusterThread class
11 invokevirtual org/terracotta/caft/common/ClusterThread.clusterJoin()V //
       invokes the clusterJoin method of the ClusterThread class
```

In this case, the start method was overridden in ClusterThread. This way, the only thing that needs

to be changed in the `invokevirtual` bytecode of the start method invocation is the class passed. As the join method is final, we need to invoke a special "clusterJoin" method instead. The ClusterThread class supports all constructors and methods present in the original Java Thread class, guaranteeing compatibility with existing applications.

This concludes the description of the `AddClusterThreadAdapter` method adapter. This instrumentation is used by the `ThreadClassAdapter` class adapter, and should be applied to every method of the multi-threaded Java application.

4.1.2 ThreadClassAdapter

The `ThreadClassAdapter` instrumentation is the class adapter responsible for adding Terracotta annotations and applying instrumentations, depending on the mode chosen. This instrumentation applies the `AddClusterThreadAdapter` to all methods of the Java multi-threaded application that is going to run in the middleware. It also checks if the class fields are supported by Terracotta, and if not, marks them as transient. Not-supported fields include classes corresponding to a local resource, such as a Process, a FileInputStream or a Socket. With this mechanism, the programmer only has to configure that specific class to not share transient fields when the class is shared in the Terracotta DSO.

In Identity mode, the `ThreadClassAdapter` also adds the Terracotta `AutolockWrite` annotation, in order to take advantage of the local synchronization to add a Terracotta transaction in every method. Without synchronization present, this annotation has no extra side-effect. In Full SSI mode, the `ThreadClassAdapter` also applies the `GetterSetterAdapter` instrumentation for adding synchronization at its lowest level, on field access and array writes, which will be described in Section 4.2.2 in more detail. To finalize, in Serialization mode, this instrumentation applies the `CaftRootAdapter` instead, which will be described in Section 4.3. The `CaftRootAdapter` instruments access to specific fields annotated by the programmer, and does not apply any Terracotta annotation, as it is not needed due to the fact that the remaining fields will be serialized. Also, to avoid the extra work of having the programmer make each class implement the Serializable interface, which is only really needed using this mode, this instrumentation adds the Serializable interface to every class automatically.

As the middleware supports a mode for instrumenting every class loaded, this instrumentation also applies the Terracotta `InstrumentedClass` and `HonorTransient` annotations to every class loaded, if requested. The former annotation tells Terracotta to instrument the class for sharing in the Terracotta DSO, while the latter tells Terracotta to not cluster fields marked as transient.

This concludes the description of the `ThreadClassAdapter`. It is the bytecode instrumentation that, together with the `AddClusterThreadAdapter`, allows the running of threads in remote worker nodes, with proper synchronization. It is also responsible for applying the instrumentations specific to the three modes supported by the middleware.

4.2 FullSSI instrumentations

In this section, we are going to describe the bytecode instrumentations that add the extra synchronization needed by the Caft Full SSI mode. These instrumentations add getters and setters for every field, with proper synchronization and Terracotta locking annotations. After this, a method instrumentation is

applied to every method by the `ThreadClassAdapter`, replacing field access with getters and setters calls and array writes with synchronized writes. Both instrumentations will be described in the next subsections, and its impact will be measured in the evaluation chapter.

4.2.1 Getter and Setter adapters - adding getters and setters for every field

For adding getters, we implemented an ASM class adapter transformation that adds a getter for each non-static field. Each getter has the Java `synchronized` method modifier and is annotated with the Terracotta `AutolockRead` annotation to allow for concurrent reads of the field, but still within the context of a Terracotta transaction. The example below illustrates the bytecode generated for the getter of a simple String field. It should be noticed that each field type is checked to generate the correct return instruction:

```
1 //Getter template
2     public synchronized getfield()Ljava/lang/String;
3   @Lorg/terracotta/modules/annotations/AutolockRead;()
4     aload 0 //Push the object reference(this) at index 0 of the local
          variable table.
5     getfield some/class.field : Ljava/lang/String; //Pop the object
          reference(this) and push the object reference for ''field''
6     areturn //Pops the top value and pushes it on the operand stack of the
          invoking method (instruction generated varies with field type)
```

We use the `synchronized` method modifier to simplify the implementation and minimize the bytecode generated. This is equivalent to add a synchronized block in the body of the method, but without the need to generate extra bytecode for `monitorenter` and `monitorexit`, as well as aditional code for handling exceptions. For adding setters, we add a method modifier in a similar way to the getter adder, with the corresponding AutolockWrite annotation:

```
1 //Setter template
2   public synchronized setfield(Ljava/lang/String;)V
3   @Lorg/terracotta/modules/annotations/AutolockWrite;()
4     aload 0 //Push the object reference(this) at index 0 of the local
          variable table.
5     aload 1 //Push the object reference corresponding to the String
          argument of the method at index 1 of the local variable table
6     putfield some/class.field : Ljava/lang/String; //Pops two values from
          the stack and stores the top value into the field named "field" of
          the instance of ''some/class'' on the stack.
7     return //Returns to the operand stack of the invoking method
```

Finally, for static fields, we proceed in a similar way, but in this case there is no object reference (this) to push to the stack. The examples below illustrate the bytecode generated for the getter and setter methods of a static String:

```
1 //Getter for a static field template
2  public synchronized getfield()Ljava/lang/String;
```

```
3  @Lorg/terracotta/modules/annotations/AutolockRead;()
4      getstatic some/class.field : Ljava/lang/String; //Push the object
           reference for ''field'' at ''some/class''
5      areturn //Pops the top value and pushes it on the operand stack of the
           invoking method (instruction generated varies with field type)
```

```
1  //Setter for a static field template
2    public synchronized setfield(Ljava/lang/String;)V
3    @Lorg/terracotta/modules/annotations/AutolockWrite;()
4      aload 1 //Push the object reference corresponding to the String
           argument of the method at index 1 of the local variable table
5      putstatic some/class.field : Ljava/lang/String; //Pops one value from
           the stack and stores in on the static field named ''field'' of ''
           some/class''
6      return //Returns to the operand stack of the invoking method
```

These instrumentations allow us to synchronize writes and reads of clustered fields, in code without synchronization initially present. In the next subsection, we will describe the method adapter used for replacing field access bytecodes with method calls.

4.2.2 Method adapter - replacing field access with instrumented methods

To use the getters and setters generated using the instrumentations described in the previous section, we developed a method adapter that replaces direct field accesses with method calls. In a regular Java application, a `getfield` bytecode is used as follows:

```
1  aload 0 //Push the object reference(this) at index 0 of the local variable
       table.
2  getfield some/class.field : Ljava/lang/String; //Pop the object reference(
       this) and push the object reference for ''field''
```

In this particular case, we have the object instance on the stack, and the `getfield` instruction will consume it, leaving the field value of that particular instance on stack. Considering this, we just have to replace the `getfield` instruction with an `invokevirtual` that will call the appropriate getter, consuming exactly the same argument and leaving the field value on stack.

```
1  aload 0 //Push the object reference(this) at index 0 of the local variable
       table.
2  invokevirtual some/class.getfield()Ljava/lang/String; //Pops the object
       reference from stack and invokes the getter method
```

In the `putfield` bytecode case, we have the object instance and the new value to be put on the instance field. The `putfield` bytecode will consume both arguments and store the new field value on the object instance. Considering this, we just have to replace the `putfield` instruction with an `invokevirtual` that will call the appropriate setter, consuming exactly the same arguments and leaving the stack state consistent.

```
1 aload 0 //Push the object reference(this) at index 0 of the local variable
      table.
2 aload 1 //Push the object reference to store on field at index 1 of the
      local variable table.
3 putfield some/class.field : Ljava/lang/String; //Pops two values from the
      stack and stores the top value into the field named ''field'' of the
      instance of ''some/class'' on the stack.
4
5 Replaced by:
6
7 aload 0 //Push the object reference(this) at index 0 of the local variable
      table.
8 aload 1 //Push the object reference to store on field at index 1 of the
      local variable table.
9 invokevirtual some/class.setfield(Ljava/lang/String;)V //Pops two values
      from stack and invokes the setter method
```

For static field access, there is no object instance on stack to be considered. The `putstatic` instruction only consumes the value to be put on the field. As the getters and setters methods generated for this kind of fields are also static, all we need to do is to invoke them using the `invokestatic` bytecode. We illustrate this in the examples below:

```
1 getstatic some/class.field : Ljava/lang/String; //Push the object reference
      for ''field'' at ''some/class''
2
3 Replaced by:
4
5 invokestatic some/class.getfield()Ljava/lang/String; //Invokes the static
      getter method and pushes the field value onto the stack
```

```
1 aload 1 //Push the object reference to store on field at index 1 of the
      local variable table.
2 putstatic some/class.field : Ljava/lang/String; //Pops the top value from
      the stack and stores in on the static field named ''field'' of ''some/
      class''
3
4 Replaced by:
5
6 aload 1 //Push the object reference to store on field at index 1 of the
      local variable table.
7 invokestatic some/class.setfield(Ljava/lang/String;)V //Pops the top value
      from the stack and invokes the setter method
```

It should be noticed that we only add getters and setters on instrumented application code. As such, the method adapter was designed to not replace field access on system classes (e.g. belonging to the java.* package).

4.2.3 Array access - synchronizing array stores

In array accesses, writes using array store instructions (`iastore`, `fastore`, `dastore`, `aastore`, `bastore`, `castore`, `sastore`), also need synchronization at some point if the array is shared by Terracotta. Prior to invoking an array store instruction, we have on stack the array reference, an `int` referring to the position on the array where the value will be stored, and the value itself. Considering this scenario, we developed a new class with static methods that consumes exactly the same arguments and performs the array store inside a synchronized block. The array store instruction is then replaced by an invocation of the method corresponding to the data type. Below, we have an example for storing an `int` inside an array of ints:

```
1 iastore //Pops the top three values from stack, corresponding to the array,
      position and value and stores the value in the specified position of
      the array
```

Replaced by:

```
1 invokestatic org/terracotta/clusterthread/asm/StaticArraySetter.setIntArray
      ([III)V //invokes the static method for storing an int in an array of
      ints
```

It should be noticed that this synchronization is very fine grained, and inside a for loop that writes to an array, it generates a significant overhead that could be avoided by synchronizing outside the loop. However, detecting such cases is a non trivial problem that falls outside the scope of this work. Also, writing to a very large array in a single transaction can take large amounts of memory as Terracotta does not automatically fragment transactions. As the Terracotta platform develops and integrates batching algorithms, this kind of problems will become less of an issue.

This concludes the description of the instrumentations used by the Full SSI mode. These instrumentations allow the middleware to run a non-synchronized application by replacing direct field accesses with invocations to special getters and setters, with the `synchronized` modifier and annotated with the corresponding Terracotta annotations. In the next section, we are going to focus on the instrumentations used by the Serialization mode.

4.3 Serialization mode - Caft Root mapping

So far, both the Identity and Full SSI mode rely exclusively on the Terracotta DSO, sharing every field belonging to a Runnable target and guaranteeing object identity for the entire thread context. However, every field that is shared holds a communication cost, and in some cases we could simply copy the data and read it locally, without need for further synchronization with the master node for the program to work correctly. This assumption is the main motivation to add an extra mode, that relies on plain Java serialization for passing a Runnable target to a worker node.

In this mode, we use ASM to add the Java Serializable interface, along with the Serialization UID if it does not exist already, in order to avoid the scenario where the user or programmer has to manually change the source code to add a new interface that was not needed before. This allows us to instantiate threads on remote workers with Runnable targets that are serialized by RMI. However, to guarantee correctness in some applications, we need to provide a way to preserve object identity between fields of

a Runnable target and other fields that remain in the home node. It should be noticed that the original Terracotta Root annotation for fields does not work in this case, as the changes in the Runnable object in the worker node are done in a serialized copy, which is considered by Terracotta as just another different object, and as such the synchronization is not done.

To solve this problem, we introduce a new Java annotation "CaftRoot". This annotation should be applied to all pairs of fields whose identity should be the same across the cluster. Pairs of fields are created by assigning the same string key in the key annotation parameter. We illustrate this concept with a code example below, taken directly from the Sunflow modification applied to Serialization mode. In this example, we show the `BucketThread` class and the `BucketRenderer` class. A bucket is a Sunflow concept corresponding to the set of pixels to be generated by a thread. The results will be presented in the Evaluation chapter 5.

```
1  public class BucketThread implements Runnable {
2          private int threadID;
3
4          @CaftRoot(key="display")
5          private Display display;
6          @CaftRoot(key="bucketCounter")
7          Integer bucketCounter;
8          @CaftRoot(key="bucketCoords")
9          int[] bucketCoords;
10         private BucketRenderer instance;
11
12         public BucketThread(int threadID, BucketRenderer instance, Display
               display, Integer bucketCounter, int[] bucketCoords) {
13                 this.threadID = threadID;
14                 this.instance = instance;
15                 this.bucketCounter = bucketCounter;
16                 this.bucketCoords = bucketCoords;
17         }
18
19         ...
20
21         @AutolockWrite
22         public void run() {
23                 //render bucket...
24         }
25
26         ...
27 }
```

```
1  public class BucketRenderer implements ImageSampler {
2          Scene scene;
3          @CaftRoot(key="display")
4          Display display;
5          // resolution
6          int imageWidth;
```

```
7           int imageHeight;
8           // bucketing
9           String bucketOrderName;
10          BucketOrder bucketOrder;
11          int bucketSize;
12          @CaftRoot(key="bucketCounter")
13          Integer bucketCounter;
14          @CaftRoot(key="bucketCoords")
15
16          ...
17
18          @AutolockWrite
19          public void render(Display display) {
20                  ...
21
22                  Thread[] renderThreads = new Thread[scene.getThreads()];
23                  for (int i = 0; i < renderThreads.length; i++) {
24                          renderThreads[i] = new Thread(new BucketThread(i,
                                this, display, bucketCounter, bucketCoords));
25                          renderThreads[i].setPriority(scene.
                                getThreadPriority());
26                          renderThreads[i].start();
27                  }
28          }
29
30          ...
```

In this example, we map the `Display` object corresponding to the shared data structure used by all threads to store the rendering calculations, as well as the array used to determine the next bucket to render and an `Integer` counter to keep track of the number of buckets processed. When a new thread is instantiated for computing a new bucket, the instance of the `BucketThread` class will be serialized and sent to a worker node. However, the Caft middleware will use the Terracotta DSO to hold the fields annotated with our special `CaftRoot` annotation. This annotation has the same semantics as the original Terracotta `Root` annotation, meaning that any fields annotated with it will be shared among all cluster nodes. In a scenario with a normal Java application running in one node, it would be the same as declaring the field to be static. The difference lies in the need to specify a key for each pair of fields, to indicate to our middleware that they should be considered the same in both the class that implements the Runnable target and the one that uses it, passing fields to its constructor. This way, the programmer gets the ability to choose pairs of fields to preserve identity, while the remaining fields will be copied and no synchronization will be done between them.

Regarding the original Sunflow application, the `BucketThread` class manipulated all three fields of the `BucketRender` class directly, so our program remains correct. It is the programmer responsibility to ensure that having just one instance shared among all cluster nodes of a field makes sense in the application, just like if it was using Terracotta without our middleware. If not, the field itself could be replaced by a map, which would store the several instances of the field.

Table 4.1: CaftRoot mapping

Map	Key	Value
fieldToKey	BucketRendererdisplay BucketThreaddisplay	display
	BucketRendererbucketCounter BucketThreadbucketCounter	bucketCounter
	BucketRendererbucketCoords BucketThreadbucketCoords	bucketCoords
clusteredFields	display	Display instance
	bucketCounter	Integer instance
	bucketCoords	int[] instance

In practice, we implement the access to fields annotated with the `CaftRoot` annotation by having two concurrent hash maps, one that that associates a string composed by the concatenation of the class name plus the field name with the key specified by the user, and another that associates this key with the concrete object instance. Both maps will belong to the Terracotta DSO to be accessible in every node. The table 4.1 summarizes the mapping that will be done for this example.

4.3.1 Caft Root Adapter

Considering the mapping done in the previous subsection, all that is left to do is to find a way to fill the "fieldToKey" map, and add instrumentations that intercepts field access and get the values from the "clusteredFields" map, leaving the stack in a correct state. The "fieldToKey" map is filled at class load-time, using our custom class loader and the Java reflection API to detect which fields have the mapping put down by the programmer. The field access instrumentations are done by a special method adapter that is applied to every method in the application, which after checking if there is a key for a class and field name pair replaces the coded accesses as follows:

```
1 //Getfield
2 ldc key
3 invokestatic org/terracotta/clusterthread/caftroot/CaftRootMap getField(
    Ljava/lang/Object;Ljava/lang/String;)Ljava/lang/Object;");
4 checkcast fieldType
```

For the `getfield` bytecode, it should be reminded that at this point we have the object instance on stack, so we simply generate code that pushes the key and invokes a static method that will consume both arguments and get the value present in the "clusteredFields" map. After this, we also generate a `checkcast` bytecode to ensure that the value put on stack is the same type of the field. The field type information is available via the ASM framework.

```
1 //Putfield
2 ldc key
3 invokestatic org/terracotta/clusterthread/caftroot/CaftRootMap putField(
    Ljava/lang/Object;Ljava/lang/Object;Ljava/lang/String;)V
```

For the `putfield` bytecode, we also need to push the key onto the stack, so it can be passed to our static method that will put the new field value into the "clusteredFields" map. This static method needs to take three arguments, in order to consume the object instance, the value to be put on the specified field, and the string we put, and leave the stack in a coherent state.

For the static versions, the instrumentations are similar to the ones described for non-static fields, except for the fact that the static method signature takes one less argument, as there is no object instance on stack in this case:

```
1 //Getstatic
2 ldc key
3 invokestatic org/terracotta/clusterthread/caftroot/CaftRootMap
      getStaticField(Ljava/lang/String;)Ljava/lang/Object;);
4 checkcast fieldType
```

```
1 //Putstatic
2 ldc key
3 invokestatic org/terracotta/clusterthread/caftroot/CaftRootMap
      putStaticField(Ljava/lang/Object;Ljava/lang/String;)V);
```

This concludes the overview of "Caft" Serialization mode implementation. It should also be noticed that the programmer needs to add the proper synchronization for the fields that are shared in the Terracotta DSO, while the serialized fields will not need any.

4.4 Summary

In this chapter, we described the implementation of our middleware, focusing on the bytecode instrumentations developed. We started by describing the basic instrumentations for scheduling threads to worker nodes, followed by the ones used by the Full SSI mode for adding extra synchronization. We finalized with the description of the Serialization mode, with a concrete example and the detailed explanation of its instrumentations and data structures used.

Chapter 5

Evaluation

In this chapter we are going to describe the methodology used for evaluating the prototype, and its results. We used up to three machines in a cluster, with Intel(R) Core(TM)2 Quad processors (with four cores each) and 8GB of RAM, running Linux Ubuntu 9.04, with Java version 1.6.0_16, Terracotta Open Source edition, version 3.3.0, and three multi-threaded Java applications that have the potential to scale well with multiple processors, taking advantage of the extra resources available in terms of computational power and memory (Fibonacci, Sunflow renderer and Matrix by vector multiplication). We are also concerned with the transparency of our approach, and how much is the impact of our bytecode instrumentations.

In short, we evaluated:

1. Correct operation of the application in the clustered environment, while taking advantage of all processors in the worker machines.
2. The extent of possible modifications required to application source code, middleware and API.
3. The increase in bytecode size, due to instrumentations (besides Terracotta own instrumentations)
4. Speed-up
5. Memory usage

5.1 Fibonacci

For testing purposes, we developed a simple application that computes Fibonacci numbers using Binet's Fibonacci number formula. The idea was to test the middleware first with an application that did not have much data to be shared (in this case, only an array of BigIntegers has to belong to the global heap). This scenario is very CPU intensive with trivial I/O, so it was expected that it would scale well with more processors. Our application takes the maximum number of Fibonacci to compute, along with the number of threads, and splits the workload by having each thread compute a number of Fibonacci numbers corresponding to the maximum given divided by the number of threads. As the computation of a Fibonacci sequence number gets more demanding with larger numbers, the split is done in an interleaved way, assigning the Fibonacci of zero to the first thread, the Fibonacci of one to the second thread, and so on. Each thread will write its computations to a private array, which will be read by the home node.

In this section, we are going to describe the changes that need to be done at source code level to make the application work in each mode, as well as the impact in the bytecode size due to the extra instrumentations applied. We measured the execution time obtained by running the application in a cluster supported by our middleware, with two or three worker nodes. For comparison purposes, we also measured the execution time in Terracotta, and in a standard JVM.

5.1.1 Source code changes

Concerning the several modes of our middleware, in Full SSI mode we simply edited the `tc-config.xml` file of our middleware to add the classes necessary to be instrumented by Terracotta. This can be done easily by instrumenting every class first, and then checking the **Terracotta Developer Console** to see the classes that really need instrumentation (as in, have some instances that are shared in the DSO). For Identity mode, we needed to add some synchronization, illustrated by the code examples below:

```
1  public class FibonacciThread implements Runnable {
2          private int min, max, nthreads;
3          private BigInteger[] res;
4
5          public synchronized BigInteger[] getRes() {
6                  return res;
7          }
8
9          ...
10
11         @Override
12         public void run() {
13                 int j = 0;
14                 synchronized(res) {
15                         for(int i=min; i < max; i += nthreads) {
16                                 res[j] = fib(i);
17                                 j++;
18                         }
19                 }
20         }
21
22         ...
```

In this case, we needed to add a synchronized block in the `run` method, to allow the subsequent writes inside the for loop to be done in the context of a Terracotta transaction. It should be noticed that the extra synchronized block does not affect concurrency, as each thread has its own private array. Also, the array results will be read later by another class in the home node, so we added the `synchronized` keyword to the `getRes` method in order to avoid dirty reads and guarantee that the correct values will be read from the Terracotta Server Array. For the Serialization mode, we used a `ConcurrentHashMap` shared by all threads and mapped by the `CaftRoot`, in order to store the private arrays of each thread. Since each thread has its own private array to store the results, we can add a synchronized block outside the `for` loop and define only one transaction for storing the results, just like we did for Identity mode. We illustrate this in the code examples below:

```
1  public class FibonacciThread implements Runnable {
2          private int min, max, nthreads;
3          @CaftRoot(key="res")
4          private ConcurrentHashMap<Integer, BigInteger[]>  results;
5
6          @AutolockRead
7          public synchronized BigInteger[] getRes() {
8                  return results.get(min);
9          }
10
11         ...
12
13         @Override
14         @AutolockWrite
15         public void run() {
16                 int j = 0;
17                 BigInteger[] res = results.get(min);
18                 synchronized(res) {
19                         for(int i=min; i < max; i += nthreads) {
20                                 res[j] = fib(i);
21                                 j++;
22                         }
23                 }
24         }
25
26         ...
27 }
```

```
1  public class Fibonacci {
2
3          @CaftRoot(key="res")
4          private static ConcurrentHashMap<Integer, BigInteger[]>  results;
5
6          ...
7
8  }
```

This concludes the subsection regarding the necessary code changes made to the Fibonacci application to make it work correctly in our middleware, concerning the several modes. As we can observe, the modifications necessary for Identity mode are quiet trivial and easy to understand to a programmer that has a notion of the concepts behind Terracotta and how it clusters data. For Serialization mode, we needed to add a map to keep the same semantics of the application, which makes this approach the least transparent and probably harder to understand. Nevertheless, the modifications needed are made using only simple Java concepts, and are certainly easier to add than it would be to develop an equivalent distributed application. In the next subsection, we are going to focus on the bytecode size impact of the custom instrumentations added for each mode.

5.1.2 Bytecode size

For the bytecode size measurements, we ran the application in our middleware in a single node, running both the master and the worker components, and used our custom Classloader to keep track of the bytecode size of each class, before and after applying our instrumentations in each mode. Since each mode needs different changes in the source code (or none), as described in the previous section, the original bytecode size before applying the bytecode instrumentations will be different, depending on the mode chosen. Also, we have taken all the measurements in the master node, which is the one that will load all the application classes needed. The results are shown in the table 5.1.

Table 5.1: Fibonacci - Bytecode size

Mode	Original (bytes)	After instr. (bytes)	Overhead ratio
Serialization	7419	8313	1.12
Identity	6275	6702	1.06
Full SSI	6253	8174	1.31

As we can observe, the mode with the largest overhead is the Full SSI mode, followed by the Serialization and Identity. Considering the bytecode instrumentations defined in chapter 4, this is expected, as the Full SSI mode adds more methods to each class. The Serialization mode shares a `ConcurrentHashMap` as demonstrated in the previous section, so it is expected to have the largest bytecode in the end, while Identity mode gets to have the least impact. However, we should keep in mind that in this case we have very few classes, and the extra annotations from Identity mode do not make much difference. In the next sub-section, we are going to focus on the speed-up improvements, which is the main motivation for running this application in a clustered environment.

5.1.3 Execution time

For the execution time measurements, we configured our application to compute the first 1200 numbers of the Fibonacci sequence, with a number of threads directly proportional to the number of processors available. In short, we ran the application with no more than one thread per processor and measured the time taken by each mode with two, four, eight and twelve threads. Also, we tested our application using only the Terracotta middleware, to have a general idea of how the usage of the original Terracotta platform impacts the performance. Considering that the Terracotta developers refer that the bytecode instrumentations impact performance, even when there is no data being shared, we considered two different scenarios for the tests: **Terracotta Inst. only** and **Terraocotta Inst + Sharing**. The former tested the application with only the Terracotta bytecode instrumentations activated, while the latter also shared the same data structures shared in the Identity and Full SSI modes. Finally, we tested our application in a standard local JVM, for comparison purposes with our distributed solution. As a distributed solution implies more communication overhead, the local solution should be always better as long as we can run one thread per core. The results are presented in Figure 5.1.

As we can observe in the graph, the overhead introduced by Terracotta is not much, as we only share a relatively small array in each thread for storing the Fibonacci numbers, along with some auxiliary

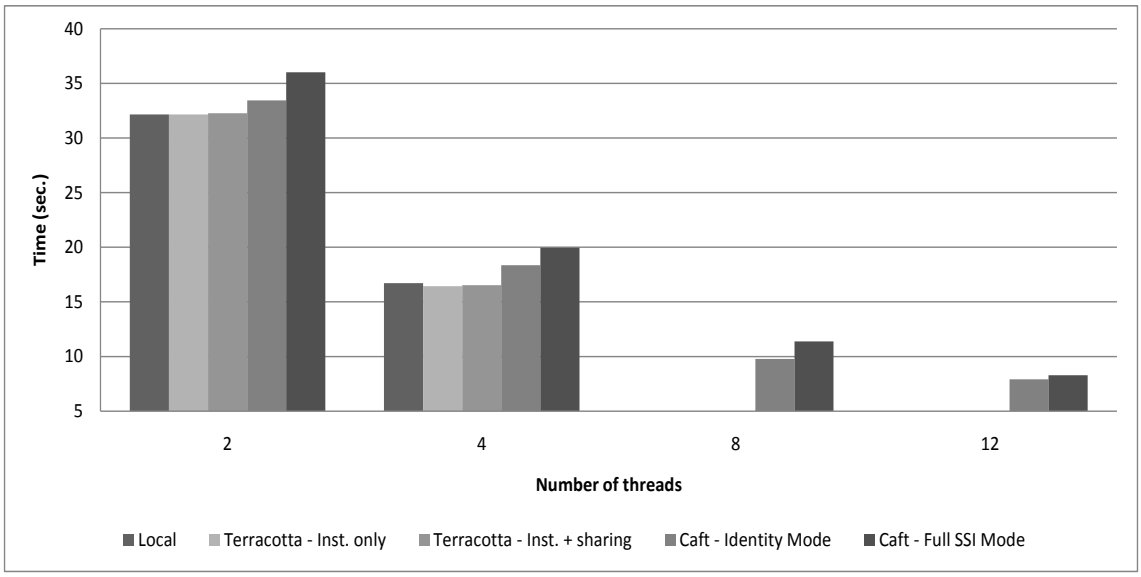

Figure 5.1: Fibonacci - Identity and Full SSI modes

variables. Also, the number of classes in the application is very small. By adding our middleware, we introduce an extra overhead which is not very significant, even when running it in Full SSI mode and as such, it is possible to obtain smaller execution times by adding more nodes to the Terracotta cluster.

For Serialization mode, we made the necessary code changes as described in section 5.1.1. The test scenario was similar to the previous one, measuring the execution time in a local JVM, in our middleware, in Terracotta with only the Terracotta instrumentations enabled, in Terracotta with instrumentations and sharing of data equivalent to the one needed for the Serialization mode to work in Caft. The results are presented in Figure 5.2.

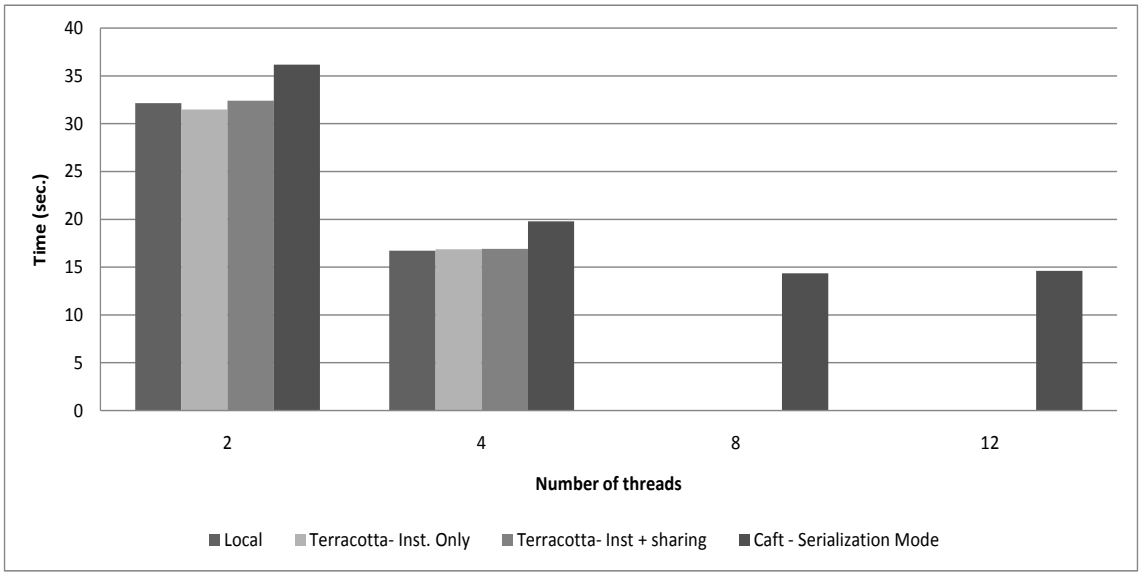

Figure 5.2: Fibonacci - Serialization mode

As we can observe in the graph, the overhead introduced by Terracotta is very similar to the one in the previous case, despite the source code being slightly different as we described in section 5.1.1. The overhead introduced by Serialization mode ends up being larger, but it is still able to achieve lower execution times that a single node in a local JVM. To finalize, we also tested the application with only

one thread in a standard JVM, in order to calculate the speed-up achieved by using a standard JVM, Terracotta, and our middleware. The results are presented in Table 5.2.

Table 5.2: Fibonacci - Speed-up comparing to Local JVM

Mode Speedup	2 threads	4 threads	8 threads	12 threads
Local	1.97	3.79	-	-
Serialization	1.75	3.20	4.42	4.34
Identity	1.89	3.45	6.49	8.02
Full SSI	1.76	3.18	5.57	7.66

In conclusion, the Identity mode scales very well, followed by the Full SSI and as last, Serialization. This is expected, as the Full SSI adds more synchronization and the serialization of the `Runnable` target implies a larger overhead than simply sharing it with Terracotta. In this case, the Serialization mode ends up sharing almost the same structures as Identity mode, and the more fine grained approach does not compensate. In the next section, we are going to test our middleware with Sunflow, an Open Source Java multi-threaded renderer.

5.2 Sunflow

Sunflow is an Open Source rendering system for photo-realistic image synthesis. It supports rendering of scenes to popular image formats such as PNG, TGA and HDR. The scenes can be specified using Java or a special scene graph language, typical of other similar applications such as POV-Ray. It should be noticed that the Swing GUI used normally in Sunflow is not supported by Terracotta, so we will perform all renderings using the command line mode instead.

5.2.1 Source code changes

As with the previous Fibonacci application, in Full SSI mode we simply edited the `tc-config.xml` file of our application to add the classes necessary to be instrumented by Terracotta. This can be done easily by instrumenting every class first, and then checking the **Terracotta Developer Console** to see the classes that really need instrumentation (as in, have some instances that are shared in the DSO). This technique facilitates the task even more, as the number of classes in Sunflow is much larger than in our previous Fibonacci application. For Identity mode, we simply ran the application and hoped that the synchronization present would suffice. In the end, we just needed to add an extra synchronized block in the `Scene` class, illustrated by the code example below:

```
1 public class Scene {
2
3         ...
4
5         public void render(Options options, ImageSampler sampler, Display
              display) {
```

```
6
7                       ...
8
9               synchronized(this) {
10                      bakingPrimitives = null;
11                      bakingAccel = null;
12          }
13
14          ...
15      }
16  }
```

For the Serialization mode, the code changes necessary were described in section 4.3 as a pratical example. Despite being a much more complex application, the changes necessary for the several modes are fairly easy to understand and apply. As already mentioned, in Serialization mode, the decision of what needs to be shared and what does not requires some knowledge of how the application works. In the next subsection, we are going to focus on the bytecode size impact of the custom instrumentations added for each mode.

5.2.2 Bytecode size

As with the previous Fibonacci application, we ran the application in our middleware in a single node, running both the master and the worker components, and used our custom Classloader to keep track of the bytecode size of each class, before and after applying our instrumentations in each node. The results are shown in the table 5.3:

Table 5.3: Sunflow - Bytecode size

Mode	Original (bytes)	After instr. (bytes)	Overhead ratio
Serialization	400559	408518	1.01
Identity	399819	416429	1.04
Full SSI	405687	542191	1.34

In the Sunflow application case, the quantity of classes that need to be instrumented is larger than in the previous Fibonacci application. As such, the Serialization mode bytecode overhead is less than its counterparts, as the programmer annotates the specific methods that require synchronization directly in the source code. Identity mode will add the `AutolockWrite` annotation to every method, and as such, the original bytecode is slightly smaller than its Serialization counterpart, but after applying the instrumentations, it becomes larger. The Full SSI still remains the mode that generates the largest bytecode, due to the extra methods that need to be added in each class. In the next sub-section, we are going to focus on the speed-up improvements and check if it compensates to run this application in our middleware.

5.2.3 Execution time

For the execution time measurements, we configured Sunflow to render one of the example images, with a number of threads directly proportional to the number of threads available. In short, we ran the application with no more than one thread per processor and measured the time taken by each mode with two, four, eight and twelve processors. We also tested our application in a standard local JVM, for comparison purposes with our distributed solution. The results for Identity and Serialization mode are presented in Figure 5.3.

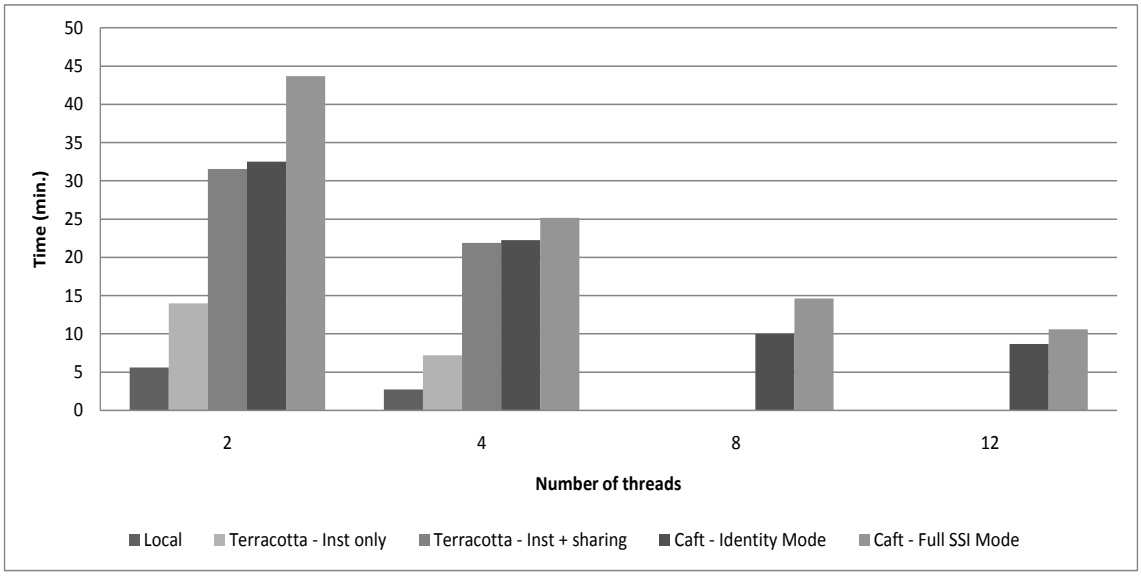

Figure 5.3: Sunflow - Identity and Full SSI modes

As we can observe in the graph, the Terracotta bytecode instrumentations add a considerable overhead, even when we do not share any data in the DSO. By adding the same data structures that are shared in both Identity and Full SSI modes, the execution times of the application in Terracotta for two and four threads are very similar to the ones presented by Caft in Identity mode, for the same number of threads. Our middleware can then obtain better execution times by using the extra processors and obtain scalability compared to Terracotta by itself. The Full SSI mode adds a more significant overhead, having the greatest execution times. However, we still do not obtain scalability when compared to a standard JVM. Considering this, we performed the necessary code changes to run Sunflow in Serialization mode, and sharing only the data structures necessary for storing the results, as described in section 5.2.1. For comparison purposes, we measured the execution time in Terracotta with instrumentations enabled, and also with an equivalent sharing of data. The results are presented in Figure 5.4.

As we can observe in the graph, the overhead introduced by both the Terracotta instrumentations and sharing of data decreased and its execution time is comparable with a standard JVM. This is expected, as there are a lot less changes that need to be propagated to the Terracotta Server, comparing to Identity and Full SSI modes. Also, many classes that needed to be instrumented by Terracotta in the previous example are not included anymore. With our middleware on top, we are able to achieve better execution times than the ones that are possible with only one node and a standard JVM.

To summarize the results obtained, we also tested our application with only one thread in a standard JVM, calculating the speed-up obtained by our middleware in the several modes. The results are shown in Table 5.4.

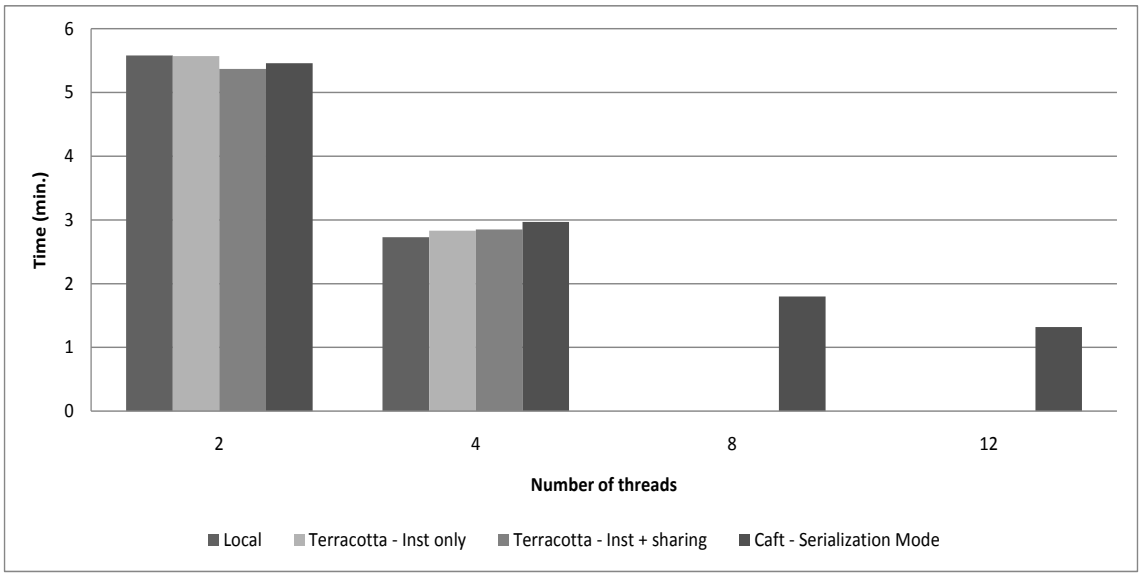

Figure 5.4: Sunflow - Serialization mode

Table 5.4: Sunflow - Speed-up

Mode Execution Time	2 threads	4 threads	8 threads	12 threads
Local	1.97	4.04	-	-
Serialization	2.02	3.71	6.13	8.35
Identity	0.33	0.49	1.10	1.27
Full SSI	0.25	0.43	0.75	1.04

As we can observe, the sharing of the entire thread context in Identity mode adds a large overhead, as every change computed in every node needs to be propagated to the Terracotta Server. Also, the necessary bytecode instrumentations of Terracotta also contribute to its overhead. In Serialization mode, where each thread only shares the data structures where the rendering results are kept, performance increases and can achieve speed-up with several nodes, compared to having just a local JVM. In the next section, we are going to test our middleware with an application that performs a multiplication of a matrix by a vector, and focus on the possibility of using our middleware to take advantage of having more memory, distributed by all nodes.

5.3 Matrix-vector multiplication

For testing purposes, we also developed a multi-threaded application that multiplies a matrix by a vector, splitting the matrix rows across the threads. It should be noticed that the matrix rows are generated randomly, each thread generating their own set of rows, storing them in local variables. In a more real world scenario, the matrix values could be read from a file which had to be accessible to all nodes. This was done in order to attempt a memory stress test, and check if it was possible to take advantage of the extra memory provided by several nodes, and if it would scale by adding more nodes into the cluster.

In our test scenario, we are limited to only one instance of the Terracotta Server, due to the fact that we are using the Open Source version of Terracotta. As such, we only obtained and measured memory scalability using the Serialization mode, where we can have local variables in the thread context that use only the local heap. Results in Identity and Full SSI mode could be different if we used the enterprise version of Terracotta, that allows the splitting of the global heap across several machines.

5.3.1 Source code changes

As with the previous applications, in Full SSI mode we simply edited the `tc-config.xml` file of our application to add the classes necessary to be instrumented by Terracotta. For Identity mode, we simply ran the application and hoped that the synchronization present would suffice. In the end, we just needed to add three extra synchronized blocks in the `MatrixVectorMultiplication` class, corresponding to writes in the shared fields of the `Runnable` target. We illustrate these changes in the code example below:

```
1  public class MatrixVectorMultiplication  implements Runnable {
2          private int nrows, ncolumns;
3          private int[][] rows;
4          private int[] vector;
5          private int[] result;
6
7          @AutolockRead
8          public synchronized int[] getResult() {
9                  return result;
10         }
11
12         ...
13
14         @AutolockWrite
15         public void run() {
16                 //Generate rows
17                 Random randNumGenerator = new Random();
18                 synchronized(this) {
19                         this.rows = new int[nrows][ncolumns];
20                 }
21                 for(int i=0; i < nrows; i++) {
22                         int[] row = new int[ncolumns];
23                         for(int j=0; j < row.length; j++) {
24                                 row[j] = randNumGenerator.nextInt();
25                         }
26                         synchronized(rows) {
27                                 rows[i] = row;
28                         }
29                 }
30
31                 //Apply multiplication
32                 for(int i=0; i < nrows; i++) {
33                         int[] row = rows[i];
```

```
34                          int acc = 0;
35                          for(int j=0; j < ncolumns; j++) {
36                                  acc += row[j]*vector[j];
37                          }
38                          synchronized(result) {
39                                  result[i] = acc;
40                          }
41                  }
42          }
```

For Serialization mode, we added the `CaftRoot` annotation to share the array responsible for storing the result of the multiplication of the matrix rows by the vector. We also changed the main loop in order to take into account that the array for storing results is now shared with every instance of the `MatrixVectorMultiplication` class. The idea is similar to the one applied when running Sunflow, share the results that need to be available in the master node, as well as the data structures relevant to keep the state of the master and workers coherent. Also, as every thread will write to different positions of the array, we can use a Terracotta transaction that allows multiple writers in the same synchronized block, by applying the `AutolockConcurrent` annotation. We illustrate the code changes in the example below:

```
1  public class MatrixVectorMultiplication  implements Runnable {
2          private int nrows, ncolumns, threadIndex;
3          private int[][] rows;
4          private int[] vector;
5          @CaftRoot(key="result")
6          private int[] result;
7
8          ...
9
10         @AutolockConcurrent
11         public void run() {
12                 //Generate rows
13                  ...
14                 //Apply multiplication
15                 for(int i=0; i < nrows; i++) {
16                         ...
17                         synchronized(result) {
18                                 result[i+nrows*threadIndex] = acc; //acc =
                                       row by vector multiplication result
19                         }
20                 }
21         }
22 }
```

```
1  public class App {
2
3          @CaftRoot(key="result")
4          private int[] result;
```

```
5
6           public App(int[] result) {
7                   this.result = result;
8           }
9
10          @AutolockRead
11          public synchronized int[] getResult() {
12                  return result;
13          }
14
15          ...
```

5.3.2 Bytecode size

As with the previous examples, we ran the application in our middleware in a single node, running both the master and the worker components, and used our custom Classloader to keep track of the bytecode size of each class, before and after applying our instrumentations in each node. The results are shown in the table 5.5.

Table 5.5: Matrix-vector multiplication - Bytecode size

Mode	Original (bytes)	After instr. (bytes)	Overhead ratio
Serialization	6561	7466	1.13
Identity	6472	6863	1.06
Full SSI	6252	8813	1.41

In this case, the results concerning the bytecode size are similar to the ones obtained by the Fibonacci application. Before we apply the instrumentations, code size is slightly smaller in the Full SSI mode due to the fact that the programmer does not introduce extra synchronization or Java annotations for clustering the application. In Identity mode however, we require that the programmer adds some synchronized blocks, impacting the original size. Code size is further increased in Serialization mode, by also adding Java annotations. After we apply the instrumentations, Full SSI mode generates the largest code, followed by Serialization and Identity mode.

5.3.3 Execution time

For the execution time measurements, we tested our application by multiplying a matrix of 32768 rows by 32768 columns and a vector of 32768 positions. As with previous applications, we ran the matrix by vector multiplication with no more than one thread per processor and measured the time taken by each mode with two, four, eight and twelve processors. We also tested our application in a standard local JVM, for comparison purposes with our distributed solution. The results for Identity and Serialization mode are presented in Figure 5.5.

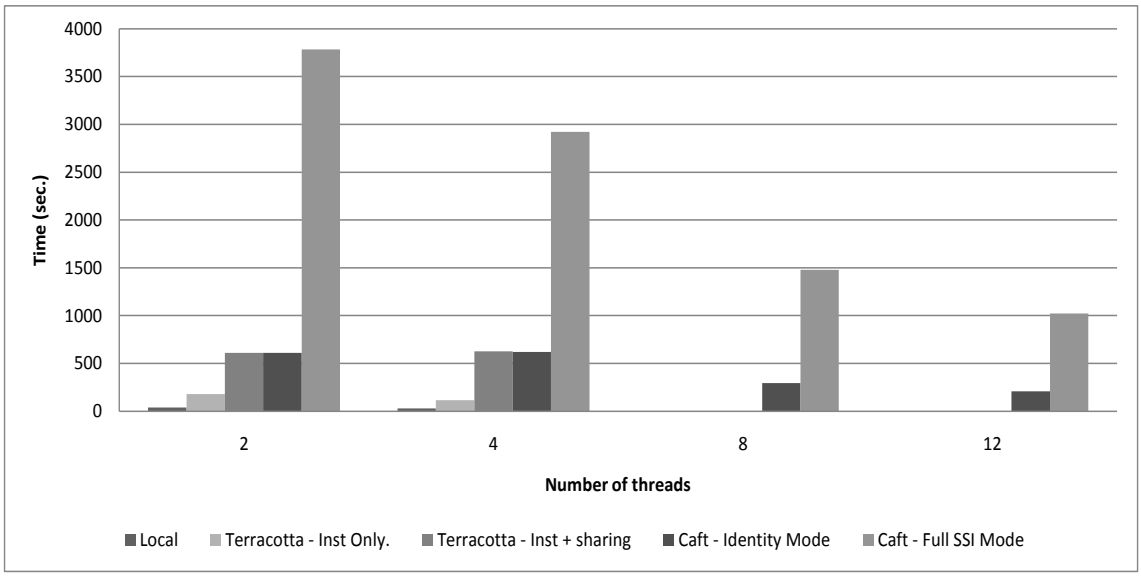

Figure 5.5: Matrix*vector - Execution times for Identity and Full SSI modes

As we can observe in the graph, the Terracotta bytecode instrumentations add a small overhead, even when we do not share any data in the DSO. By adding the same data structures that are shared in both Identity and Full SSI modes, the execution times of the application in Terracotta for two and four threads are very similar to the ones presented by Caft in Identity mode, for the same number of threads. Our middleware can then obtain better execution times by using the extra processors and obtain scalability compared to Terracotta by itself. The Full SSI has an execution time much greater than any of its counterparts, as every write in an array of results needs to be propagated to the DSO.

As with the Sunflow application, we still do not obtain scalability when compared to a standard JVM. Considering this, we performed the necessary code changes to run the application in Serialization mode, and sharing only the data structures necessary for storing the results, as described in section 5.3.1. For comparison purposes, we measured the execution time in Terracotta with instrumentations enabled, and also with an equivalent sharing of data. The results are presented in Figure 5.6.

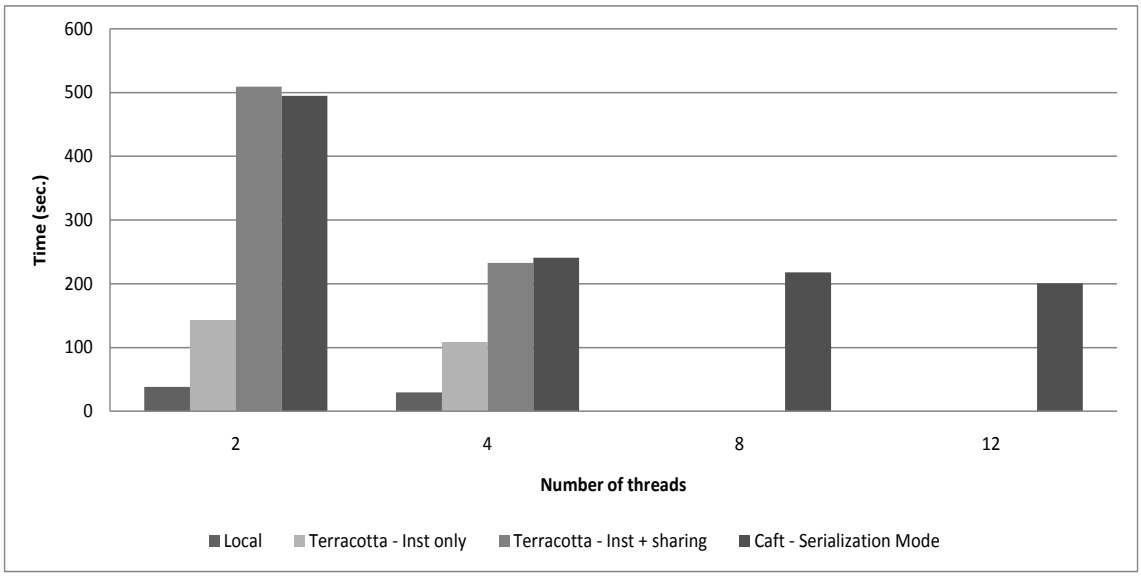

Figure 5.6: Matrix*vector - Execution times for Serialization mode

As we can observe in the graph, the results are slightly better than the previous modes, but the execution times are still not better than a local JVM. This can be explained by the fact that this application is much more memory-intensive than CPU intensive, spending a more considerable amount of time writing results to array positions instead of computing the matrix, which is pretty trivial in comparison with the previous applications, where we computed Fibonacci numbers or performed ray-tracing calculations. This type of applications should not scale very well in Terracotta, in terms of speed-up, but we believe the extra memory available in the cluster can still give a competitive advantage over a single node. This assumption shall be tested and measured in section 5.3.4.

To summarize the results obtained, we also tested our application with only one thread in a standard JVM, calculating the speed-up obtained by our middleware in the several modes. The results are shown in Table 5.6. As we can observe, despite the fact that the speed-up increases a little with the number of nodes, it has no comparison possible with a local JVM.

Table 5.6: Matrixvecmul - Speed-up

Mode Execution Time	2 threads	4 threads	8 threads	12 threads
Local	1.38	1.80	-	-
Serialization	0.11	0.22	0.24	0.26
Identity	0.09	0.09	0.18	0.26
Full SSI	0.01	0.02	0.04	0.05

5.3.4 Memory usage

In this section, we attempted to stress test our middleware from the memory usage perspective, to check if it was possible to take advantage of the extra memory provided by several nodes, and if it would scale by adding more nodes into the cluster. It should be noticed that measuring the exact memory taken by an instance of a Java object is tricky, as it will depend a lot on the JVM implementation. As such, we attempted to run the application using three different matrix sizes: 32768x32768, 62556x32768, 53090x53090. The memory occupied by each of them was estimated considering that each `int` value has at least 4 bytes. This technique will not determine the exact amount of maximum memory that we could allocate with a certain number nodes, as the actual size in memory is dependent on the JVM implementation itself, but it can demonstrate that by adding more nodes we can allocate more memory. The heap sized was fixed at 7 GB of data, as the machines were limited to 8 GB of RAM and we wanted to save some space for other JVM objects and other applications running in each node. The results are shown in Figure 5.7:

In this example, we managed to allocate a matrix of 62556x32768 with two nodes, corresponding to 8 GB of data, while with one node only we would get a `java.lang.OutOfMemoryError`. In a similar way, with three nodes we could allocate about 10.5 GB of data, while with only two nodes we got the same exception. In conclusion, adding more nodes allowed us to perform computations with a matrix in memory split across several machines, which would not be possible if the application was running in a local JVM.

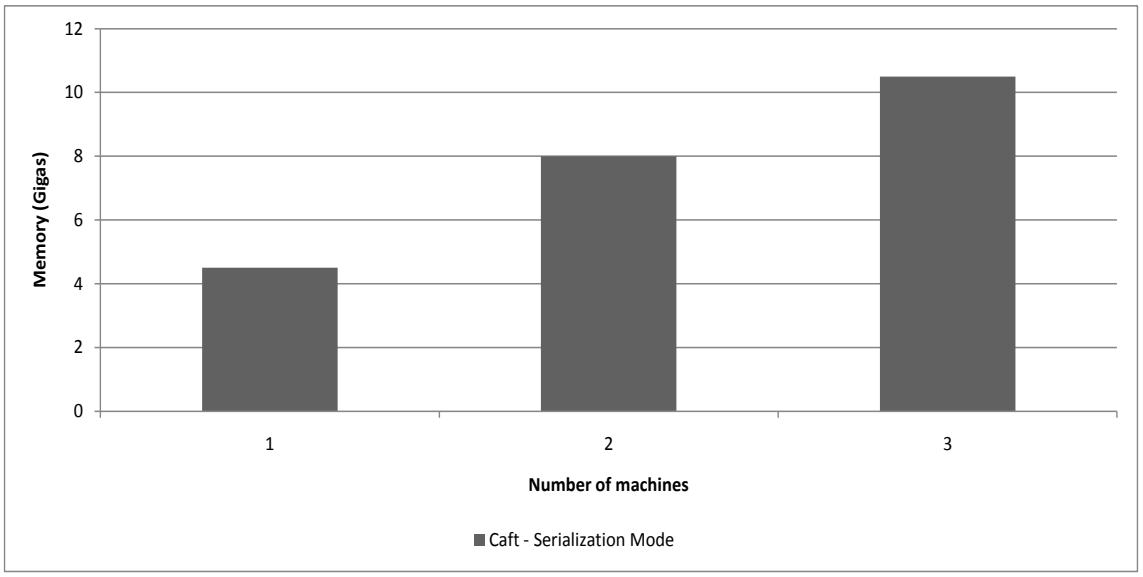

Figure 5.7: Matrix*vector - Memory Stress

5.4 Compatibility issues

In this work, we tried to keep the middleware with the greatest compatibility possible, allowing it to run any Java multi-threaded application. However, by running our middleware on top of Terracotta, we inherit the following limitations:

- **Volatile keyword:** Terracotta does not support the volatile keyword. Volatile fields will behave just as normal fields. Applications that use it, or rely on its semantics for synchronization may not work correctly.
- **Non-serializable fields:** Some fields may not be serializable and we need to mark them as transient. This workaround may not work if the field really needs to be accessible on another machine. For example, we may have a Java CyclicBarrier accessed by a thread that is supported in Terracotta, but not serializable, which limits our prototype to use only the Identity and Full SSI modes.
- **Non-portable fields:** Some fields may not be portable at all by Terracotta. These types are well documented by Terracotta and correspond mainly to network and I/O classes.

5.5 Summary

In this chapter, we evaluated our middleware using three multi-threaded Java applications: Fibonacci, Sunflow renderer and Matrix by vector multiplication. We focused on the code changes necessary to run each application on all three modes, in order to illustrate the transparency and compromises of our implementation. Considering performance, we measured the code size overhead generated by our instrumentations, as well as speed-up or memory usage depending on the application. We concluded that the code changes necessary are mostly local and easy to implement, and the most transparent mode is Full SSI, followed by Identity mode and Serialization mode.

From the evaluation of the three applications, we concluded that the best use case for our middleware is for running already existent Java multi-threaded applications, designed for running in a single node, and in which an adaptation for running in a clustered environment should not be trivial to implement. Using our round-robin scheduler, our middleware will scale with applications that have about the same amount of work per thread, and require little or no communication. Such problems are often referred in the literature as *Embarrassingly parallel*. In applications that are not very CPU intensive and require more I/O writes, the performance of our middleware will not be that good. Despite these limitations, our solution is scalable, in the sense that it can achieve shorter execution times by adding more nodes to the Terracotta cluster, in all modes.

Comparing our solution to running an an application in a single, standard JVM, more transparency comes with a trade-off of performance. Identity and Full SSI modes are dependent on the nature of the fields shared and manipulated. These problems can be circumvented by sharing only what is necessary in some applications that implement an embarrassingly parallel problem, such as Sunflow. Concerning memory usage, our middleware can take advantage of the extra memory given by several nodes in a scenario where threads allocate a big amount of local objects.

Chapter 6

Conclusion

In this work, we explored a different use case for Terracotta, the running of simple, multi-threaded applications, that were not designed with Terracotta or clustering in mind. We attempted to develop an approach with the best transparency possible that would not require deep changes in the application, and still be powerful enough to achieve good performance.

We started by developing a method to create threads and schedule them to worker nodes, while using the Terracotta DSO to store the Runnable target. We concluded that this method allows the programmer to run multi-threaded applications in a distributed way, using only pure Java and adding synchronization as necessary. The overhead will be very dependent on the thread context itself, concerning the amount of data shared and manipulated.

By implementing a bytecode approach that adds synchronization on field and array access, we concluded that it is possible to improve the Terracotta DSO usage by automatically adding some extra synchronization that is needed for defining transaction boundaries. It should be noticed that some of this synchronization would not be needed when running in a single JVM, as the semantics of the code running can guarantee that there are no write conflicts by itself. This mode should be kept optional in our middleware, due to the extra overhead observed in the evaluation section. As Terracotta is developed, and new transaction batching algorithms are integrated, the performance impact of coarse-grained versus fine-grained transactions should be minimized.

And last, the Serialization approach allows for a more fine-grained sharing of objects on the global heap. This provides a "mixed" semantic that is not very typical of Terracotta, as the most common use case is to use "Ehcache" replication which either serializes every object put on cache or preserves identity. We concluded, from the evaluation section, that this approach can hold very good results in real-world applications such as the Sunflow renderer.

6.1 Future work

The current approach of our middleware assumes that every machine has the same resources in terms of memory and computational power. Also, the computation cost of every thread has to be similar, in order to preserve load-balancing. In this section, we present a non-exhaustive list of improvements to the

current design and implementation that could be done in order for the middleware to work with other kind of applications:

- Add mechanisms that would make Caft aware of the resources available in the cluster, such as the current number of threads executing, the CPU load of each node, and memory available. These mechanisms could be used to implement extra load-balancing mechanisms based on the cluster state.
- Implement fault-tolerance mechanisms that allows the coordinator component to assign threads to other workers, if one of them goes off-line.
- Allow for clustered threads with classes that extend the Java Thread class.
- For the array writes in Full SSI mode, research code static analysis techniques that could be applied to avoid the linear growth of the number of Terracotta transactions with the number of loop iterations.
- Integrate thread migration mechanisms that can stop a thread executing in a certain node, and resume it on another node which should be less loaded.

Bibliography

[1] Introduction to the spring framework.

[2] C. Amza, A. L. Cox, S. Dwarkadas, H. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29:18–28, 1996.

[3] J. Andersson, S. Weber, E. Cecchet, C. Jensen, V. Cahill, J. A. Ý, S. W. Ý, E. C. Þ, C. J. Ý, V. C. Ý, and T. College. Kaffemik - a distributed jvm on a single address space architecture, 2001.

[4] G. Antoniu, L. Boug, P. Hatcher, M. MacBeth, K. Mcguigan, and R. Namyst. The hyperion system: Compiling multithreaded java bytecode for distributed execution, 2001.

[5] Y. Aridor, M. Factor, and A. Teperman. cjvm: a single system image of a jvm on a cluster. In *In Proceedings of the International Conference on Parallel Processing*, pages 4–11, 1999.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[7] B. Bershad, M. Zekauskas, and W. Sawdon. The midway distributed shared memory system. In *Compcon Spring '93, Digest of Papers.*, pages 528–537, Feb 1993.

[8] G. E. Blelloch, P. B. Gibbons, G. J. Narlikar, and Y. Matias. Space-efficient scheduling of parallelism with synchronization variables. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 12–23, New York, NY, USA, 1997. ACM.

[9] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[10] S. Bouchenak and S. Bouchenak. Pickling threads state in the java system. In *Third European Research Seminar on Advances in Distributed Systems (ERSADS'99*, 1999.

[11] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[12] R. Buyya, T. Cortes, and H. Jin. Single system image. *Int. J. High Perform. Comput. Appl.*, 15(2):124–135, 2001.

[13] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, 1999.

[14] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. *SIGOPS Oper. Syst. Rev.*, 25(5):152–164, 1991.

[15] G. G. H. Cavalheiro, F. Galilée, and J. louis Roch. Athapascan-1: Parallel programming with asynchronous tasks. In *Yale University*, page 98, 1998.

[16] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance sparc cmt processor. *IEEE Micro*, 29(2):6–16, 2009.

[17] P.-C. Chen, C.-I. Lin, S.-W. Huang, J.-B. Chang, C.-K. Shieh, and T.-Y. Liang. A performance study of virtual machine migration vs. thread migration for grid systems. *Advanced Information Networking and Applications Workshops, International Conference on*, 0:86–91, 2008.

[18] W. Z. Cho-Li, C. li Wang, and F. C. M. Lau. Lightweight transparent java thread migration for distributed jvm. In *In International Conference on Parallel Processing*, pages 465–472, 2003.

[19] M. Choi, J.-L. Yu, H.-J. Kim, and S.-R. Maeng. Improving performance of a dynamic load balancing system by using number of effective tasks. *Cluster Computing, IEEE International Conference on*, 0:436, 2003.

[20] B. Dimitrov and V. Rego. Arachne: A portable threads system supporting migrant threads on heterogeneous network farms. *IEEE Transactions on Parallel and Distributed Systems*, 9:459–469, 1998.

[21] M. Factor, A. Schuster, and K. Shagin. Javasplit: A runtime for execution of monolithic java programs on heterogeneous collections of commodity workstations. *Cluster Computing, IEEE International Conference on*, 0:110, 2003.

[22] T. Fahringer. Javasymphony: A system for development of locality-oriented distributed and parallel java applications. In *In Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2000*. IEEE Computer Society, 2000.

[23] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[24] J. Gosling and H. McGilton. The java language environment: A white paper, May 1996. url-http://java.sun.com/.

[25] M. Herlihy, J. E. B. Moss, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *in Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[26] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *In The 2nd IEEE Symposium on High-Performance Computer Architecture*, pages 14–25, 1996.

[27] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *In Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 277–287, 1996.

[28] F. Informatik, T. H. Darmstadt, T. Kunz, and T. Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(17):725–730, 1991.

[29] A. Itzkovitz, A. Schuster, and L. Shalev. Thread migration and its applications in distributed shared memory systems. *Journal of Systems and Software*, 42:71–87, 1997.

[30] M. Janjua, M. Yasin, F. Sher, K. Awan, and I. Hassan. Cejvm: "cluster enabled java virtual machine". *Cluster Computing, IEEE International Conference on*, 0:389, 2002.

[31] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *In Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, 1992.

[32] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[33] K. Li. Ivy: a shared virtual memory system for parallel computing. pages 94–101, Aug. 1988.

[34] T.-Y. Liang, C.-K. Shieh, and J.-Q. Li. Selecting threads for workload migration in software distributed shared memory systems. *Parallel Comput.*, 28(6):893–913, 2002.

[35] M. J. M. Ma, C.-L. Wang, and F. C. M. Lau. Jessica: Java-enabled single-system-image computing architecture. *J. Parallel Distrib. Comput.*, 60(10):1194–1222, 2000.

[36] G. J. Narlikar. Scheduling threads for low space requirement and good locality. In *SPAA '99: Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pages 83–95, New York, NY, USA, 1999. ACM.

[37] R. V. Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Satin: Simple and efficient java-based grid programming. In *In AGridM 2003 Workshop on Adaptive Grid Middleware*, 2005.

[38] L. Parziale, E. M. Dow, K. Egeler, J. J. Herne, C. Jordan, E. L. Alves, E. P. Naveen, M. S. Pattabhiraman, and K. Smith. *Introduction to the new mainframe: z/vm basics.* IBM Corp., Riverton, NJ, USA, 2007.

[39] J. Protić, M. Tomašević, and V. Milutinović. *Distributed Shared Memory: Concepts and Systems.* John Wiley & Sons, 1998.

[40] M. Rosenblum. The reincarnation of virtual machines. *Queue*, 2(5):34–40, 2004.

[41] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[42] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode transformation for portable thread migration in java. In *ASA/MA 2000: Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, pages 16–28, London, UK, 2000. Springer-Verlag.

[43] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, 1992.

[44] B. H. Sirac, S. Bouchenak, and D. Hagimont. Approaches to capturing java threads state. In *In Middleware 2000*, 2000.

[45] J. E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[46] E. Speight and J. K. Bennett. Brazos: A third generation dsm system. In *IN PROCEEDINGS OF THE 1ST USENIX WINDOWS NT SYMPOSIUM*, pages 95–106, 1997.

[47] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of "legacy" java software. pages 236–255. Springer-Verlag, 2001.

[48] Terracotta. A technical introduction to terracotta. 2008.

[49] K. Thitikamol and P. Keleher. Thread migration and communication minimization in dsm systems. In *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, pages 487–497, 1999.

[50] K. Thitikamol and P. Keleher. Thread migration and load balancing in non-dedicated environments. *Parallel and Distributed Processing Symposium, International*, 0:583, 2000.

[51] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. pages 178–204. Springer-Verlag, 2002.

[52] R. Veldema, R. Bhoedjang, and H. Bal. Distributed shared memory management for java. In *In Proc. sixth annual conference of the Advanced School for Computing and Imaging (ASCI 2000*, pages 256–264, 1999.

[53] F. C. M. L. W. L. Cheung, C. L. Wang. Building a global object space for supporting single system image on a cluster. *Annual Review of Scalable Computing*, 4:225–260, 2002.

[54] W. YU and A. COX. Java/dsm: A platform for heterogeneous computing. 1997.

[55] M. Zenger. Javaparty - transparent remote objects in java, 1997.

[56] W. Zhu, C.-L. Wang, and F. C. M. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. *Cluster Computing, IEEE International Conference on*, 0:381, 2002.