INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Efficient Thread Scheduling for Distributed Java VM

Navaneeth Rameshan

Dissertation submitted to obtain the Master Degree in
Information Systems and Computer Engineering

Jury

Chairman:	Prof. Luis Eduardo Teixeira Rodrigues
Supervisor:	Prof. Luis Manuel Antunes Veiga
Members:	Prof. Johan Montelius

July 2012

Acknowledgements

I would like to express my sincere gratitude to Prof. Luís Antunes Veiga who guided me through this research, for all the interesting discussions and brain storming sessions, and for being a great source of inspiration throughout the course of this thesis.

I would also like to thank all my EMDC friends who helped me through the course of my thesis, both with regards to technical discussions and having fun. Special mention to Paulo, Liana, Francesco and Vanco for making my stay in Lisbon a pleasurable experience.

I would finally like to thank my parents for all the support, care and patience while I was finishing my thesis without whom none of this would have been possible.

This work was supported by national funds through FCT - Fundação para a Ciência e a Tecnologia, under projects PEst-OE/EEI/LA0021/2011 and PTDC/EIA-EIA/113613/2009.

European Master in Distributed Computing, EMDC

This thesis is part of the curricula of the European Master in Distributed Computing (EMDC), a joint program among Royal Institute of Technology, Sweden (KTH), Universitat Politecnica de Catalunya, Spain (UPC), and Instituto Superior Técnico, Portugal (IST) supported by the European Community via the Erasmus Mundus program. The track of the author in this program has been has follows:

First and second semester of studies: IST
Third semester of studies: KTH
Fourth semester of studies (thesis): IST

Abstract

In this work, we propose RATS, a middleware to enhance and extend the Terracotta framework for Java with the ability to transparently execute multi-threaded Java applications to provide a single-system image. It supports efficient scheduling of threads, according to available resources, across several nodes in a Terracotta cluster, taking advantage of the extra computational and memory resources available. It also supports profiling to gather application characteristics such as dispersion of thread workload, thread inter-arrival time and resource usage of the application. Profiling and clustering capabilities are inserted with the help of byte code instrumentations. We developed a range of alternative scheduling heuristics and classify them based on the application and cluster behavior. The middleware is tested with different applications with varying thread characteristics to assess and classify the scheduling heuristics with respect to application speed-ups. Results indicate that, for a CPU intensive application, it is possible to classify the scheduling heuristic based on application and cluster properties and also achieve linear speed ups. Furthermore, we show that a memory intensive application is able to scale its memory usage considerably when compared to running the application on a single JVM.

Resumo

Neste trabalho propomos RATS, um middleware para estender a framework Terracotta Java, com a possibilidade de executar transparentemente aplicações Java multi-tarefa oferecendo a semântica single-system image. Suporta escalonamento eficiente de tarefas, de acordo com os recursos disponíveis, sobre os nós de um cluster Terracotta, tirando partido da capacidade extra em recursos computacionais e memória. O RATS também permite a análise de execução e desempenho (profiling) de modo a aferir características das aplicaçõs tais como a dispersão de carga entre tarefas, tempos entre lançamento de tarefas, e utilização de recursos pela aplicação. O profiling e a execução em cluster são tornadas possíveis pela instrumentação de bytecodes das aplicações Java. Desenvolvemos um conjunto de heurísticas de escalonamento de tarefas e classificamo-las de acordo com o comportamento de aplicações em execução no cluster. O middleware RATS foi testado com diferentes aplicações com características variadas em termos de número e carga de tarefas, de modo a analisar e classificar as heurísticas de escalonamento consoante os aumentos de desempenho (speed-up) conseguidos. Os resultdos indicam que, para aplicações de processamento intensivo, é possivel classificar as heurísticas baseado nas propriedades da aplicação e do cluster, obtendo speed-up linear. Para além disso, demonstramos que aplicações de utilização intensiva de memória também têm melhor escalabilidade quando comparadas com a sua execução numa única máquina virtual.

Keywords

Java
Parallel and Distributed computing
Scheduling
Byte Code Instrumentation
Terracotta

Palavras-Chave

Java
Computação Paralela e Distribuída
Escalonamento de Tarefas
Instrumentação de Byte code
Terracotta

Index

List of Figures

List of Tables

Listings

1 Introduction

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 [10]. Much research has been done in the area of SSIs, such as Distributed Shared Memory (DSM) systems and Distributed Virtual Machines that can run applications written in a high-level language in a cluster, behaving as if it were on a single machine. With such an abstraction, the developer is completely oblivious to the issues related to distributed computing and writes an application just like any other application meant for a single system. Issues like socket connection, remote method invocations and scheduling of threads are some of the many details that are abstracted. Unlike JavaSpaces [28], which provides an API for distributed programming, where the developer is expected to learn the API and use them wisely in order to scale an application, the idea of SSI follows no use of an API or no knowledge of distributed systems.

The current most popular system that uses a shared object space to provide a single system image is Terracotta [2]. One of the essential mechanisms necessary for providing SSI systems is efficient scheduling of threads to improve the performance and load balancing across the cluster. At present, Terracotta does not support scheduling of threads and instead multiple manual instances need to be launched to scale applications. In this thesis, we develop a middleware that leverages compute intensive multi-threaded java applications (including those that launch threads dynamically) for scalability, and supports efficient scheduling of threads to improve performance of already existing applications. We propose RATS, a Resource Aware Thread Scheduling for JVM level Clustering which is an extension of Caft [21]. Caft provides full transparency for running multi-threaded applications. RATS bridges the gap between transparency and efficient scheduling of threads using Terracotta to keep data consistent across the cluster and scale existing applications with ease.

Several studies have showed that no single scheduling algorithm is efficient for all kinds of applications. To the best of our 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 none of the system considers different characteristics of application behavior for improving performance. The requirements of a scheduling heuristic for a memory-intensive application is different from that of a CPU-intensive application. RATS

supports multiple scheduling heuristics and they target different characteristics of applications. The scheduling heuristics maintain state information of the worker in the form of resource usage and threads launched to make optimal decisions. RATS also provides a profiler that allows to characterize an application based on the dispersion of thread workload, thread inter-arrival time and the resource usage of the application. The information obtained from the profiler allows to opt for a scheduling heuristic that best suites the properties of the application and the cluster.

1.1 Contribution

Instead of proposing a single scheduling heuristic that suites all kinds of application, we develop multiple scheduling heuristics each targeted to suite a specific class of application and cluster behavior. The middleware supports a hybrid (centralized and distributed) form of scheduling under scenarios where threads spawn multiple threads so as to improve performance. The application behavior can be obtained using a profiler, that gives information about the workload distribution of each thread, the inter-arrival time between threads and the resource usage of the application. Although the profiler gives exact information of the application characteristics, in some cases, the developer may already know the behavior of the application with a definite certainty and may not want to profile the application. In order to facilitate efficient scheduling even under such scenarios where the exact values of application characteristics are not available, the scheduler does not rely on any values that define an application.

1.2 Results

The outcome of this thesis is a prototype middleware that bridges the gap between efficient scheduling of threads and transparency allowing for distributed execution of a multi-threaded java application meant for a single system. Although this introduces additional overheads in terms of size of bytecode and additional communication over the network, results indicate that it is possible to achieve significant improvement in performance. From the results, we also see that different scheduling heuristics perform differently for different classes of application and cluster. Thus, we are able to classify the heuristics based on these behavior.

1.3 Research Context

The research described in this thesis was done within the Distributed Systems research group at Inesc-ID Lisboa, in the context of a National FCT research project, *Synergy*, on new execution environments. A paper that describes part of this work is under review for publication in Portuguese national conference INForum 2012.

The paper is titled ”RATS - Resource aware thread scheduling for JVM level clustering”.

1.4 Document Roadmap

The remainder of this thesis is organized as follows. Chapter 2 describes the relevant related work concerning this thesis. Chapter 3 provides an overview of the architecture of the middleware along with the supported scheduling heuristics and profiling capabilities. Chapter 4 describes relevant details of RATS's implementation. Chapter 5 presents results obtained from evaluation of the middleware and finally Chapter 6 concludes the thesis and provides pointers for future work.

2 Related Work

This chapter describes the most relevant research work for the definition of our middleware, organized according to a top-down approach. The first section examines existing systems that allow a regular application written in Java to become cluster-aware and run seamlessly with minimal programmer intervention. Next section describes the important scheduler aspects, such as classes of algorithms, scheduling algorithms and existing techniques for thread scheduling on a MIMD architecture.

2.1 Distributed Virtual Machines

In this section, we study the different existing platforms that can make an application written in java cluster-aware without modifying the source code. There are three major approaches that exist for distributed execution in a cluster. They are: Compiler-based Distributed Shared Memory systems, Cluster-aware Virtual Machines and systems using standard Virtual Machines. Compiler-based Distributed Shared Memory Systems is a combination of a traditional compiler with a Distributed Shared Memory system. The compilation process inserts instructions to provide support for clustering without modifying the source code. Cluster-aware Virtual Machines are virtual machines built with clustering capabilities in order to provide a Single System Image (SSI). Systems using Standard VMs are built on top of a DSM system to provide a Single System Image for applications. 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.

Compile-based DSMs were developed in order to combine cluster awareness without compromising performance. During the compilation process, the compiler adds special instructions to achieve clustering capabilities without modifying the source code of the application. The application can then be executed as a native application. A fine-grained DSM provides a global address space which is implemented using a DSM coherence protocol. They do not suffer as much from false sharing as the memory is managed in small regions. Unlike a page-based DSM, the access checks need to be performed in software. Jackal [34] 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 maxi-

mum performance of execution possible without a JIT. Jackal does not support thread migration. Hyperion [5] has a runtime which provides an illusion of single memory and also supports remote creation of threads. The main performance bottleneck is in maintaining a consistent view of the objects, as it maintains a master copy which is updated after every write. Both these systems work only on homogenous clusters as the compiled code is native thus posing a major limitation.

The most popular system in the context of cluster-aware virtual machines is cJVM [6]. cJVM is able to distribute the threads in an application along with the objects without modifying the source or byte code of an application. It also supports thread migration. To synchronize the objects across the cluster a master copy is maintained and updated upon every access and is a major bottleneck. In Kaffemik [4], all objects are allocated in the same virtual memory address across the cluster thus allowing a unique reference valid in every instance of the nodes. However, it does not support caching or replication, and an object field access can result in multiple memory accesses, thus reducing performance.

Some of the most popular systems for compiler-based DSM approaches are java party [37], java Symphony[15] and JOrchestra[33]. J-Orchestra uses bytecode transformation to replace local method calls for remote method calls and the object references are replaced by proxy references. Java Symphony allows the programmer to explicitly control the locality of data and load balancing. All the objects needs to be created and freed explicitly which defeats the advantage of a built-in garbage collection in JVM. Java party allows to distinguish invocations as remote and local by modifying the argument passing conventions. The implementation does not satisfy the ideal SSI model as classes need to be clustered explicitly by the programmer.

2.2 Scheduling

In this section we classify the scheduling algorithms based on their nature and criteria for scheduling and then give a brief description of the various classical scheduling algorithms for distributed systems. A good scheduling algorithm should be able to achieve an acceptable performance in the long run while maintaining a balanced system with a fair share of load between different computing units. The following subsection classifies the scheduling algorithms.

2.2.1 Classification of Scheduling Algorithms

In order to compare and distinguish the different scheduling algorithms based on their nature, a classification needs to be made. In [11], Casavant et al. propose a hierarchical taxonomy for scheduling algorithms in general-purpose parallel

and distributed systems [13]. Different types of scheduling are classified in a top down approach and are: Local vs. Global, Static vs. Dynamic, Approximate vs. Heuristic, Distributed vs. Centralized, batch, immediate and adaptive scheduling. In the following, each of these classification is explained.

- **Local vs. Global** : At the highest level, scheduling can be divided into local and global [11]. Local scheduling deals with algorithms that are responsible for allocating processes on single CPU [13]. Global scheduling allocate processes to multiple processors based on the information of the system aiming at optimizing a system-wide performance goal [13]. Scheduling in Grid and Cluster fall in the category of global scheduling.

- **Static vs Dynamic**: In the hierarchy, Global scheduling can be further divided into Static vs. Dynamic. In [35] and [36], Xhafa et al. state that there exist two main aspects to determine the dynamics of scheduling: dynamics of job execution and dynamics of resources. Dynamics of job execution refers to the situation of job failure. Dynamics of resources refer to the possibility of resources joining and leaving the system and changes in policies for local resource usage. In static scheduling, it is assumed that all the information about the existing resources is available before schedule time. Each of the tasks in an application is assigned once to a resource. With static scheduling it is possible to estimate the costs of computation even before the tasks finish execution [13]. However, these estimations fail if any computing unit fails or if there is a network failure. These situations are highly possible and mechanisms such as rescheduling [12] were introduced to alleviate the problem.

  In dynamic scheduling, tasks are scheduled as they arrive. It is not possible to determine the time taken for execution in all the cases. Dynamic scheduling is particularly useful in such cases. Dynamic scheduling has two components: system state estimation and decision making. System state estimation is responsible for collecting information about the computing units and building an estimate of the global information. This estimate will provide the base for mapping a task to a resource. Since it is not possible to estimate computation costs before execution, load balancing will ensure maximizing resource usage in a fair way.

- **Approximate vs. Heuristic**: Approximate algorithms are based on the use of a formal computational model in order to find a good solution. It does not search the entire solution space for an optimal or best solution, instead uses the first solution that is sufficiently good. Thus, it is important to have a metric that allows the model to judge the value or relative quality of a solution. Heuristc algorithms are based on the most realistic knowledge of the system and the process, or job, as opposed to a formal assumption. These algorithms are most suited in scenarios where the application and/or

the resources are highly diverse, often dynamic. These algorithms may not give the most optimal solution but instead take the most reasonable amount of time taken to make a decision.

- **Centralized vs Decentralized vs Hierarchical**: The scheduling responsibility can be delegated to a single scheduler (Centralized) or be shared by multiple distributed schedulers (Decentralized). In the centralized approach there is only one scheduler. It is possible to monitor all the resources state which makes easier to create efficient schedulers [35]. Centralized scheduling allows for easy management [19] and implementation of the scheduler. However, centralized scheduling poses many limitations. They are a single point of failure [36] and do not scale [13, 19, 35, 36]. Condor [31, 36] uses a centralized scheduler based on the ClassAd matchmaker [26]. In the decentralized approach the responsibility of scheduling is spread over multiple computing units. The local schedulers resident on different computing units play an important role as all the scheduling requests are sent to them. These type of schedulers take in to account important issues such as fault-tolerance, scalability and multi-policy scheduling. In the hierarchical approach, local schedulers are organized in an hierarchical way. This approach is more scalable and fault-tolerant than the centralized approach, although not better than the decentralized approach, though with simpler coordination.

- **Immediate vs batch**: In the immediate approach, jobs are scheduled as and when they enter the system [36] using the system's scheduling algorithm. They do not rely on time interval for the scheduler to get activated [36] On the other hand, in the batch approach, jobs are grouped in batches and scheduled as a group [36]. Batch approach is more suited for applications with the properties of a bag of task, and can depend on the resource characteristics better than immediate since the time between activation of scheduler can provide a better view of the system. Batch scheduling for Grids waits for a number of jobs to arrive and then makes scheduling decisions as a whole, opposed to online scheduling required for multithreaded programs where the number of threads are not known in advance.

- **Adaptive**: This approach uses information regarding the current status of the resources and predictions of their future status in order to make a decision and the decision may change dynamically based on this status. The adaptation of the algorithm can depend on changes in the resource, on performance and also on the characteristics of the application. In [25], Othman et al refer that the system must be able to recognize the state of resources and propose an adaptable resource broker. An example of an adaptive scheduling algorithm can be found on Huedo et al. work [17].

Table 2.1 tabulates the classification of the scheduling approaches.

Design Choice	Approaches
Dynamics	Dynamic Static
Architecture	Centralized Hierarchical Decentralized
Mode	Immediate Batch
Decision Making	Approximate Heuristic Adaptive

Table 2.1: Classes of scheduling algorithms

2.2.2 Classic Scheduling Algorithms

In this section we present some of the classical scheduling algorithms in Grids and distributed systems.

- **First Come First Served**: In First Come First Served algorithm, execution of jobs happen in the order they arrive ie. the job that arrives first is executed first [20]. This algorithm however has a major disadvantage. If a large job arrives early, all the other jobs arriving later are stalled in the waiting queue until the large job completes execution. This affects the response time and throughput considerably. The situation is referred to as convoy effect.

- **Round Robin**: The disadvantage in the previous algorithm is overcome by Round Robin. In this algorithm algorithm, every job is assigned a time interval, called quantum, during which it is allowed to run [29]. Upon completion of the time quantum, if the job has not yet finished its execution, it is put back in the waiting queue until its next turn for the quantum [20]. Since jobs execute only for a specified quantum, the problem of larger jobs stalling jobs that arrive later is mitigated. The biggest challenge with this algorithm is to find a suitable length for the quantum [29].

- **Minimum Execution Time** The Minimum Execution Time (MET) algorithm assigns each task to the resource that performs it with the minimum execution time [22]. MET does not consider whether the resource is available or not at the time (ready time) [14, 22, 27] and can cause severe imbalance in load across resources [14, 22, 27]. The main advantage of the algorithm is that it gives to a task the resource that performs it in the smallest amount of time [22]. MET takes $O(m)$ time to map a task to a resource [14].

- **Minimum Completion Time** The Minimum Completion Time (MCT) algorithm assigns a task to the resource that obtains the earliest completion time for that task [14, 22, 27]. Completion time is the time that a machine will take to finalize the processing of the previous assigned tasks and the planned tasks. This criteria requires knowing, for each machine, the ready time and expected time to complete the assigned tasks. The following equation calculates the completion time of machine *m* in the ETC model [35, 36].

$$completion\,time[m] = ready\_times[m] + \sum_{j \in Tasks | schedule[j]=m} ETC[j][m]$$

  Where *schedule* is the schedule for machine m and *ETC* is the Estimated Time to Compute. The objective function consists in minimizing the completion time of all machines. It is possible that the resource with the minimum completion time does not have the minimum execution time [14, 22, 27]. MCT takes $O(m)$ time to map a task to a resource [14].

- **Min-min** The Min-min algorithm has two phases [14]. In the first phase, the minimum completion time of all the unassigned tasks are calculated [27]. In the second phase, the task with the minimum completion time among the minimum completion time that was calculated in the first phase is chosen. It is then removed from the task list and assigned to the corresponding resource [14]. The process is repeated until all the tasks are mapped to a resource.

- **Min-max** The Min-Max algorithms has two phases [18, 27] and uses the minimum completion time (MCT) for the first phase and the minimum execution time (MET) for the second phase as metrics. The first phase of Min-Max is the same as the Min-min algorithm. In the second phase, the task whose $\frac{MET_{\{fastest\,machine\}}}{MET\{selected\,machine\}}$ has the maximum value will be selected for mapping [18]. The task is removed from the unassigned list, resource workload is updated and the process is repeated until the list is empty [27]. The intuition of this algorithm is that we select resources and tasks from the first step, so that the resource can execute the task with a lower execution time in comparison with other resources [18].

- **Max-min** The first phase is same as the Min-min algorithm [14, 18, 27]. In the second phase, the task with the maximum completion time is chosen, removed from the task list and assigned to the corresponding resource [27]. The process is repeated until all the tasks are mapped to a resource. Max-min can be combined with Min-min in scenarios where the tasks are of different lengths [20].

- **Suffrage** In the Suffrage algorithm the criteria to assign a task to a resource is the following: assign a resource to a task that would suffer the most if that resource was not assigned to it [20, 22]. In order to measure the suffrage, the suffrage of a task is defined as the difference between its second minimum completion time and its minimum completion time [18, 27]. These completion times are calculated considering all the resources [20]. Once a task is assigned to a resource it is removed from the list of unassigned tasks and the process is repeated until there are no tasks in the unassigned list.

- **Largest Job on Fastest Resource - Shortest Job on Fastest Resource**: This algorithm aims at minimizing the makespan and the flow time of the tasks.

  Makespan is defined as the finishing time of the last task. It is one of the most popular optimization criteria. Small values of makespan indicate that the scheduler is operating in an efficient way [16]. Considering the makespan as the only critera does not imply the optimization of other objectives.

  $$makespan = \max\{F_i\,,\, i = 1, \ldots, N\}$$

  Where F_i is the finish time of the i^{ith} task and N is the total number of tasks. The objective function consists in minimizing the maximum value of makespan.

  Flow time is the sum of the finishing times of tasks [27]. Flow time measures the response time of the system.

  $$flow\,time = \sum F_i\,,\, i = 1, \ldots, N$$

  Where F_i is the finish time of the i^{ith} task and N is the total number of tasks.The objective function consists in minimizing the flow time.

  The Largest Job on Fastest Resource - Shortest Job on Fastest Resource (LJFR-SJFR) algorithm allocates the largest job on the fastest resource in order to reduce makespan and the smallest job to fastest resource in order to reduce the flow time. In the first phase, the algorithm is the same as the Max-min algorithm with one difference, LJFR-SJFR does not consider all the jobs (*N*). Let $0 < m < N$ be the number of considered jobs on the first phase. At the end of the first phase, *m* jobs are assigned to *m* machines. In the second phase, the remaining jobs are assigned using Min-min and Max-min methods alternatively i.e. SJFR followed by LJFR [18, 27].

Table 2.2 provides a comparison of these scheduling algorithms and also classifies them based on their properties.

Algorithms	Order-Based	Heuristic	Mode	Complexity
FCFS	Yes	No	Immediate	*
Round Robin	Yes	No	Batch	*
MET	No	Yes	Immediate	$O(m)$
MCT	No	Yes	Immediate	$O(m)$
Min-Min	No	Yes	Batch	$O(s^2m)$
Min-Max	No	Yes	Batch	$O(s^2m)$
Max-Min	No	Yes	Batch	$O(s^2m)$
Suffrage	No	Yes	Batch	$O(s^2m)$
LJFR-SJFR	No	Yes	Batch	$O(s^2m)$

Table 2.2: Comparison of scheduling algorithms

2.2.3 Thread Scheduling

In order to schedule threads, existing techniques aim to achieve good locality and low space bounds [23]. To achieve good locality it is necessary to ensure that threads that share same data are scheduled on the same processor as long as the processor is not overloaded, in order to avoid the overhead of fetching the pages from memory. Low space requirements mean that the scheduling algorithm should consume minimum amount of memory so as to scale with number of processors and threads.

Work stealing scheduling techniques [8] achieve a good compromise between both locality and space requirements. In this approach, every processor maintains its own queue of threads and whenever the queue becomes empty, the processor steals threads from the other processor queue. Threads relatively close to each other in the computation graph are often scheduled to the same processor thus providing a good locality. The space requirement is not the best and can be improved.

Depth-first search scheduling [7] is another dynamic approach to scheduling. As the computation progresses, the scheduler computes the task graph. 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). A set of processors holds two queues, one for receiving tasks (input queue) and the other for storing newly created tasks (output queue). The remaining processors are responsible for taking tasks from the output queue and scheduling it to the input queue of another processor. However, as the created tasks have a relative high probability of being related with the previous computation, the locality is not as good but the space bound is much better than the work stealing approach.

These algorithms have been widely studied and were used to introduce scheduling in many parallel programming libraries and applications. Satin [24] is a Java-based grid computing programming library that implements a work stealing ap-

proach by allowing a worker node to steal a method invocation from another node. When considering applying this algorithm 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 [32]. 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.

2.3 Caft

In this section, we briefly describe Caft [21], a middleware that we extended to include scheduling heuristics and profiling capabilities. Caft runs on top of the Terracotta system and has the capacity to run simple multi-threaded Java applications in a transparent way, taking advantage of the extra computational and memory resources available, however without any support for resource monitoring or informed scheduling. It uses bytecode instrumentations to add basic clustering capabilities to the multi-threaded Java application, as well as extra synchronization if needed. It uses a master-worker paradigm where the master is responsible for launching the application and the worker is responsible for executing the threads. The main contribution of Caft is the three modes it supports, in order to achieve a balance between transparency and flexibility.

The 3 modes of transparency supported by Caft are: Identity, Full SSI and Serialization.

- **Identity**: In identity mode, it is assumed that proper synchronization exists within the application, or at least, that the user has access to source code for adding additional synchronization manually. All the thread fields of the application are shared in the Terracotta DSO to ensure that the writes are propagated and all methods are annotated with the AutolockWrite Terracotta annotation, so as to convert all synchronized access to a Terracotta transaction.

- **Full SSI**: In Full SSI mode, it is assumed that the application lacks proper synchronization or that 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 in order to allow for Terracotta transactions.

- **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 any kind of synchronization.

Summary

In this chapter, we described the most relevant research work for the definition of our middleware, organized according to a top-down approach. The first section examined existing systems allowing a regular application written in Java to become cluster-aware and run seamlessly with minimal programmer intervention. Then, we described the important scheduler aspects, such as classes of algorithms, scheduling algorithms and existing techniques for thread scheduling on a MIMD architecture.

3 Architecture

This chapter describes the architecture of the middleware, implemented to allow Terracotta to schedule threads for simple multi-threaded java applications on a cluster. Thread scheduling is facilitated by either minimum or no modifications to the source code of the application depending on the implementation of thread usage. This chapter begins with an introduction to Terracotta and explains the necessary internals needed to understand the development of the middleware. Then, a high level view of the architecture is described along with an example of how different components communicate with each other. All the supported Scheduling heuristics are discussed in the following section. Finally, the profiling capabilities of the middleware is described.

3.1 Terracotta

In this section, we introduce the purpose of Terracotta and provide a basic background of the main concepts needed to understand the RATS (Resource Aware Thread Scheduling) middleware. With this introduction, we expect the readers to be able to configure and cluster java applications using Terracotta. It is the underlying motivation of our work and allows to provide a single system image for any existing java application with minor or no modifications (transparent clustering). Terracotta achieves this functionality by instrumenting the byte code of java applications to inject clustered behaviour.

Terracotta is a Java infrastructure software that allows you to scale your application for use on as many computers as needed, without expensive custom code or databases [30]. Apart from scaling and a transparent clustering service, it also provides availability without a database. If an application is suspended manually or due to unforeseen reasons such as power failure, no information is lost when the application restarts. The state of all the objects are restored as they existed in memory before the application terminated. The main concepts of Terracotta are outlined below:

- **Architecture**: Terracotta adapts a client/server architecture and the nodes that run the application JVM are termed as *Terracotta-clients* or *Terracotta cluster nodes*. All the Terracotta clients run the same application code and it is injected with cluster behaviour according to the Terracotta configuration. This instrumentation is performed at runtime when the classes are loaded

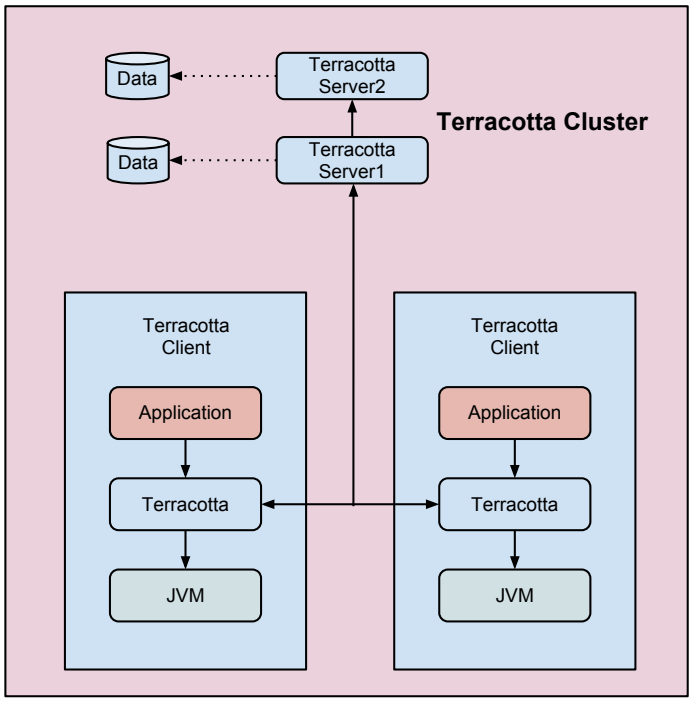

Figure 3.1: Terracotta Architecture

by each JVM and helps Terracotta achieve transparent clustering. Terracotta server is the heart of a Terracotta cluster and performs clustered object data management, storage, coordination and lock management. There can be multiple Terracotta servers and they behave as a cohesive array, with one server acting as the master and the others acting as a passive mirror in case the master crashes.

All Terracotta clients initiates a connection with the Terracotta server on start-up and communicate with the server for coordination. In case of a failure or network interruption, the clients try to connect to the server only for a configurable number of seconds during which time locks cannot be acquired and it blocks. After this time, the client connects to the passive mirror. Figure 3.1 shows the architecture of Terracotta along with their communication model.

- **Roots and Virual Heap**: In a single JVM, objects reside in the heap and are addressed by local references. If an application needs to be clustered, some objects need to be shared among the cluster and these objects are called *Distributed shared objects* or *Roots*. Once an object becomes a root, all the other objects reachable from the object reference becomes a clustered object. Since these objects are shared, they cannot be referenced from the local heap, instead they are placed on a virtual heap. Terracotta manages the virtual heap in a manner similar to virtual memory in an operating system and

thus provides an illusion of an unlimited physical heap. To the application these are just regular objects and are accessed just like accessing any other local objects. Terracotta injects instrumentation to the application and takes care of object management.

Terracotta server is responsible for keeping track of these objects and lazily load them to the clients as and when needed. Upon changing the state of a clustered object, the client notifies the server. The server upon receiving this information stores them on the disk and sends them to other Terracotta clients that need them. If any Terracotta client does not have a clustered object in its local heap, the server is responsible for serving them when requested. Figure 3.2 shows roots, clustered objects and virtual heap in a Terracotta cluster.

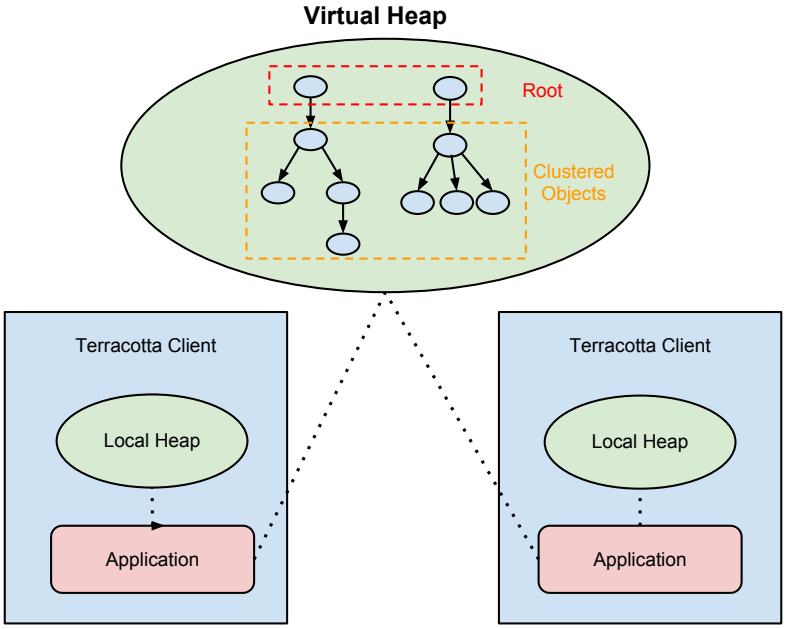

Figure 3.2: Root, Clustered Objects and Heap

Not all objects in the application are clustered objects. The developer has the freedom to choose the objects that need to be clustered. This can be configured by declaring an object as root.

- **Locks**: To achieve a single system image, multiple instances of the application are launched with Terracotta behaviour. Unless an object is clustered (declared as root), there is no coordination and they continue to run as independent instances. Clustering makes sense only when at least a single object is declared as root. However, when an object becomes a root, accesses to the object needs to be coordinated and this can be achieved by Terracotta locks.

The operations performed after obtaining a lock, and before releasing the lock, comprise a Terracotta transaction. Thus, locks serve as boundaries for Terracotta transactions. Locks are also used to coordinate access to critical sections of the code shared between threads.

There are two types of locks:

- *Autolocks*: This allows Terracotta to use already existing synchronization present in methods to access shared objects. It can be understood as providing a cluster wide meaning to locks for threads using synchronized keyword. If there is no synchronized method, Terracotta provides an auto-synchronized mode that behaves just like the method had the keyword synchronized. Autolocks acquire a cluster wide lock only if the object being synchronized on is a clustered object, otherwise only a local lock is acquired. This allows for a fine grained locking.
- *Named locks*: Named locks are acquired only at method entry and released on method exit. The lock needs to be obtained from the Terracotta server and allows only one JVM to access the method. As a result they are more coarse grained and need to be used only when autolocks are not possible since they affect the performance considerably.

- **Instrumentation**: Transparent clustering is achieved by byte code instrumentation. This instrumentation is done before the JVM loads the byte code and handles behind the scene communications with server, locking and access to virtual heap. Instrumentation is an over head, as it reduces the performance during run-time (clustering) and class load time. As a result, it is best to instrument only those classes that need clustering behaviour injected in them. All the classes that access a shared object needs to be instrumented.

- **Configuration**: Since there is no API, control over what gets clustered is defined explicitly by a configuration file. Roots, locks and instrumentation are specified in a configuration file that is used by Terracotta for instrumenting and injecting a clustered behaviour.

  It is to be noted that not all objects can be clustered. Because a clustered object resides on a virtual heap, it needs to be portable. Some classes are inherently not portable and one such class is the java.lang.thread class. Also, any subclass of a non-portable class cannot be clustered. For a portable class to be clustered, its super class if any, also needs to be instrumented.

Terracotta provides both scalability and availability as explained previously. However, the current limitations of Terracotta our middleware tries to overcome are:

1. Instances of the application need to be manually launched for each Terracotta client.

2. Threads that are launched in the application 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.

3. Leverage the support in the previous item with the ability to achieve performance speed-ups and engage resources of the cluster efficiently.

By providing an abstraction for Terracotta, we avoid launching multiple instances of the application. Instead the application is launched only on one Terracotta client and the threads that are created in that client are scheduled on other clients.

In the next section we present a high level architecture of our middleware that leverages Terracotta functionality in order to schedule threads on remote clients.

3.2 RATS - Resource Aware Thread Scheduling for JVM-level Clustering

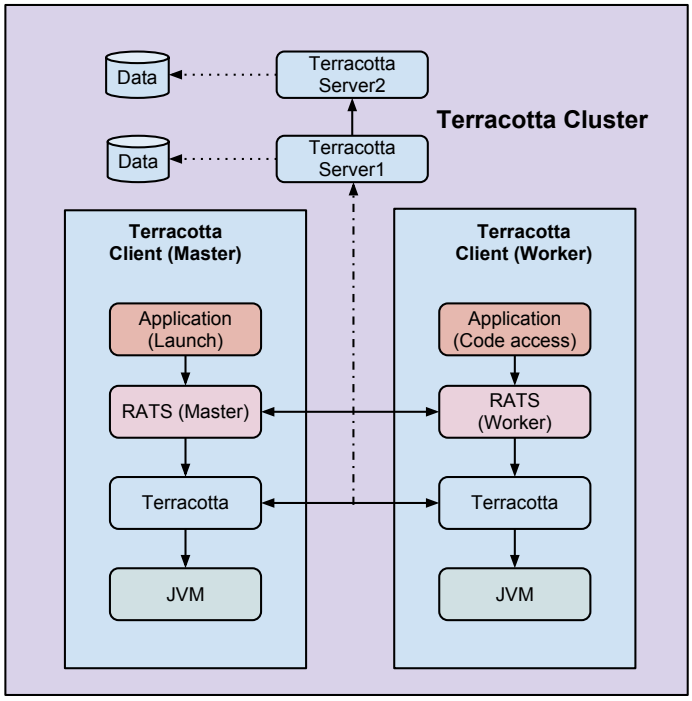

Figure 3.3: Architecture of RATS

RATS middleware consists of two components - A Master and Worker. The master is responsible for running the application and launches threads remotely

on the worker nodes according to the specified scheduling heuristic. The worker exposes an interface for launching threads and provides all the operations supported by java.lang.Thread class. The master, on the other hand is responsible for launching the application with an executable jar and uses a custom class loader that loads the class after performing necessary instrumentation to the application code. Both the master and worker need access to the class files; master requires the class files to launch the application and worker needs access to code to run newly spawned threads. Figure 3.3 provides a high level view of the RATS architecture.

RATS was implemented by modifying an existing middleware called CAFT (Cluster Abstraction for Terracotta) [21]. CAFT provides basic support for remote thread spawning. RATS extends CAFT to enhance byte code instrumentation along with support for multiple resource aware scheduling algortihms. To understand how the master worker paradigm allows for remotely spawning threads, we first provide a high level architecture of the communication between master and worker and in the following section we explain the different scheduling heuristics the system supports.

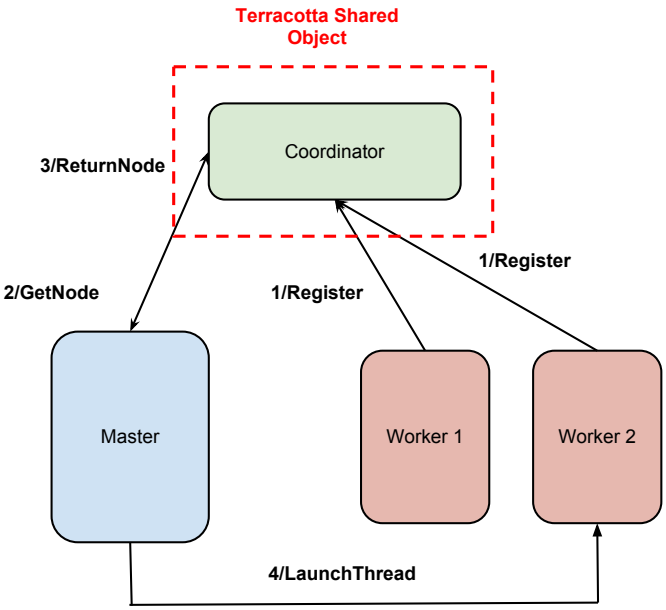

Figure 3.4: Master-Worker Communication

Figure 3.4 shows the communication between different components that are required to remotely spawn a thread. As soon as the workers are launched, they first register themselves with the coordinator (1/Register). The coordinator is a singleton object and is also a Terracotta shared object. The coordinator acts as

an interface between the worker and master and is used for sharing information between each other. By designing the coordinator as a shared Terracotta object, it leverages the power of Terracotta in avoiding to launch it as an independent component and also avoids the hassle of serializing and remotely sending over information for coordination. After registering themselves with the coordinator, the workers will start an RMI service using the Spring framework waiting to receive any runnable object to execute. When the master is started, a custom class loader loads the application and replaces the instantiation and any reference to java/Lang/Thread with a custom class ClusterThread that uses the RMI interface exposed by the worker for launching threads. When the application is launched, and a thread is instantiated, the master communicates with the coordinator component to fetch the node for launching the thread (2/GetNode). The coordinator communicates with other components responsible for scheduling and returns the node (3/ReturnNode). Upon receiving the information of the node for remotely spawning the thread, the master finally launches the thread on the worker (4/LaunchThread). Here in this example, worker 2 is chosen for running the thread.

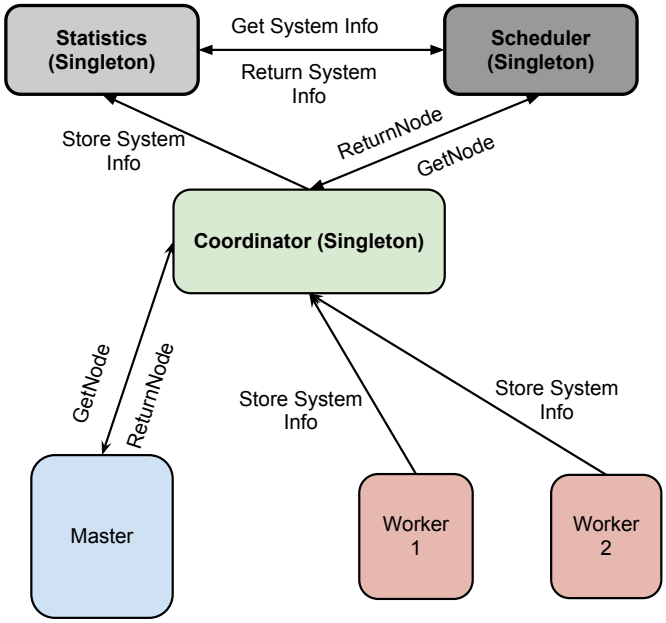

Figure 3.5: Communication between different components for making a scheduling decision

Now, we explain the communication of the coordinator component with the other components that are responsible for making scheduling decisions, and stor-

ing system information. The scheduler component is responsible for making scheduling decisions based on the scheduling heuristic chosen by the user. The statistics component is responsible for storing various information about the workers including load of the workers and state information. Figure 3.5 shows the communication and information shared between different components for maintaining state information and making scheduling decisions. Coordinator acts as an interface for communication between different components which may or may not reside on the same node. The workers upon starting, monitor their memory and CPU load using the SIGAR library [1]. SIGAR is an open source library for gathering system information and is platform independent. Because SIGAR is platform independent, it allows for the middleware to run on heterogenous clusters. The gathered system information such as CPU load and Free memory is sent to the coordinator which in turn stores this information using the statistics component. As explained in the previous paragraph, the master communicates with the coordinator when it needs to launch a thread. The coordinator delegates this call to the scheduler which makes a decision based on the chosen scheduling heuristic. If necessary, the scheduler communicates with the statistics component for fetching system information about the workers. All the three components (Coordinator, Scheduler and Statistics) are singleton objects and can only be instantiated once.

This concludes the section which describes the RATS architecture and communication between the core components of the middleware. In the next section we present the different scheduling heuristics supported by the middleware.

3.3 Scheduling Techniques

This section presents the core features of the RATS middleware. When an application launches a thread, the master is responsible for making scheduling decisions based on the chosen heuristic. The worker can also make scheduling decisions if a thread spawns multiple threads. The middleware also allows for profiling an application in order to choose the best scheduling heuristic for efficient load balancing and performance. The middleware supports two types of scheduling and they are presented below:

3.3.1 Centralized Scheduling

In centralized scheduling, the decisions are taken entirely by a single node. Here, the master is responsible for making every scheduling decision. Based on the specified heuristic, the master selects a worker for remotely executing the thread and also maintains state information. This includes information about the load of each worker in terms of CPU cycles consumed, memory consumed and a mapping of threadIDs to the worker executing the thread. Before we describe the different scheduling heuristics supported, we explain the motivation for keeping

the scheduling heuristics simple.

Most of the related work regarding thread scheduling on a single machine are concerned about locality of data and space bound. Some DSM systems have also considered locality based scheduling to minimize page thrashing. We however settle for simple scheduling heuristic mainly because of two reasons. Firstly because most of the data used by the thread itself is placed in the Terracotta server and fetched by the workers only when necessary. Terracotta virtual memory manager sets the references to objects that have fallen out of the cache to null so that they may become eligible for local garbage collection. Because clustered objects may be lazily loaded, purged objects will be transparently retrieved from the server as references to them are traversed. It would require a complex approach of tweaking Terracotta in order to minimize page thrashing at the risk of affecting memory scalability. Second, it is necessary to achieve good space bound which is possible by using a simple scheduling heuristic using as little memory as possible. As we will show in subsequent sections, in some cases workers themselves make scheduling decisions (to further reduce latency, and load on the master) and a lower space bound is only advantageous.

The centralized scheduling heuristics supported by the middleware are:

- **Round-Robin** : In round-robin scheduling, the threads launched by the application are remotely spawned on the workers in a circular manner. Threads are launched as and when they arrive and the scheduling is static by nature. It does not take into account any information from the system and the workers are chosen in the order they registered with the master.

- **Resource-Load**: Scheduling decisions are made depending on the load of every worker. The supported scheduling heuristics based on load information are:

  - **CPU-Load**: The CPU load of every worker is monitored by the master and the threads are remotely launched on the worker with the least CPU load. The master maintains state information about the CPU load of every worker. This scheduling heuristic is particularly helpful when the multi-threaded application is CPU-intensive.

  - **Memory-Load**: The master maintains state information about the memory of every worker and threads are launched on workers with the highest available memory. This scheduling heuristic is particularly helpful when the multi-threaded application is memory-intensive.

  - **Load-Average**: Load-Average is a moving average of the number of active processes. This scheduling heuristic aims to equalize the load

averages of all the machines. Load average values are not instantaneous and hence we use an estimation for equalizing the values.

- **Accelerated-Load-Average**: Accelerated-Load-Average is similar to Load-Average but instead aims at equalizing the instantaneous changes in load averages rather than the whole load-average. This makes it more accelerated and responsive when compared to Load-Average scheduling as it remains unaffected by the previous load of the system.

  The load information of CPU and memory is updated by the worker in one of the two ways:

  1. *On-demand*: When an application is just about to launch a thread, the master requests all the workers to provide their current CPU/memory load. Thus, state information is updated only on demand from the master. This is a blocking update and it incurs an additional overhead of round trip time delay to every worker for every thread launch.
  2. *Periodic*: The load information of CPU maintained by the master is updated after a constant period. The period required to perform updates is a configurable parameter which can be chosen by the user. All updates are performed asynchronously and hence they do not block remote launching of threads.

- **Thread load**: The master maintains state information about the number of threads each worker is currently running. The scheduling heuristic makes decisions to launch threads on workers with the least number of currently executing threads. If there are multiple workers with the same number of threads, the choice is made in the natural order the workers registered. This heuristic schedules in a circular fashion just like round robin until at least one thread exits. Once a thread exits, it ceases to behave like round robin. The state information is updated only when a thread begins or finishes execution.

3.3.2 Hybrid Scheduling

The Middleware also supports for hybrid scheduling, where local copies of information that help scheduling decisions are maintained. The trade-off between consistency and performance is handled optimally for distributed scheduling. Once a thread is scheduled to a worker, depending on the application, the thread itself may launch more internal threads. Under such a scenario, there are three possible ways of handling the scheduling decision each with its own advantages and disadvantages. We explain the three possible methods and the approach opted by our middleware for handling the scheduling of internal threads.

- **Master as Scheduler**:

  Once a thread is launched on a worker and if the thread launches multiple internal threads, the worker communicates with the master and the master makes the scheduling decisions and sends it to the worker which then spawns the thread remotely on the selected worker. This approach obviously involves a round trip time delay and blocks the thread until it receives the information from the master. The additional overhead incurred by the round-trip delay and central scheduling can be overcome if the worker makes the scheduling decision by itself. This leads us to the second approach as explained below.

- **Worker as Scheduler from Global information**:

  Instead of delegating the responsibility of scheduling the internal threads to the master, the worker can make its own scheduling decision. This would require a local scheduler at each worker. Whenever an internal thread is spawned, the worker communicates with the local scheduler which then fetches the state information from the global information table, a Terracotta shared object, by communicating with the coordinator. This is shown in Figure 3.6. After the local scheduler obtains the necessary information, it makes the scheduling decision and updates the information on the global table. This approach overcomes the problem of central scheduling but it still involves a one-way network delay for fetching state information before the worker can make any scheduling decisions. Another issue with this approach is that it could block the scheduling if many workers spawn internal threads simultaneously. Although, the communication with the global information table before any scheduling is read-only, when workers interleave their communication, any update on the table could result in obtaining stale data unless their access is synchronized. The global table may never have a real view of the system. The following example provides such a scenario.

  Consider a cluster with one master and two workers. The master spawns 2 threads remotely, one on each worker and both threads spawn multiple internal threads. Say, threadID-1 is spawned on worker-1 and threadID-2 on worker-2. If threadID-1 launches internal threads before threadID-2, worker-1 communicates with the global table first and fetches state information after which the local scheduler in worker-1 updates the global table. If threadID-2 spawns any internal thread during this update, worker-2 will block on the global table if access is synchronized or it could result in worker-2 obtaining stale data. This simple example shows the potential problem of a dirty read. It could also result in a dirty write, in which case the global table will never have a real view of the system. This approach could incur an overhead if access is synchronized. When there are multiple workers with multiple internal threads, the overhead could be quite substantial. The following approach tries to overcome this problem and is explained below.

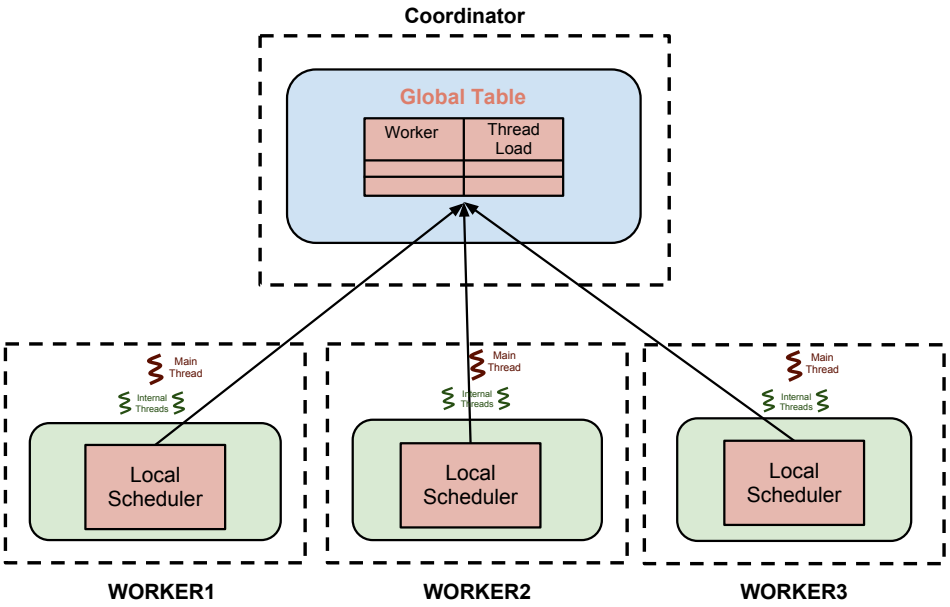

Figure 3.6: Communication for Worker to perform Scheduling from Global Information Table

- **Worker as Scheduler from Local Information** (Chosen Approach):

  In this approach, the master asynchronously sends the state information table to every worker before any thread launch. The workers on receiving the information table store a copy of the table locally. This is shown in Figure 3.7. Workers use this local table for making scheduling decisions after which they update the local table and then the global table. Once a worker updates its local table, there are inconsistencies between the information table with the workers. Although there are inconsistencies between the local tables of every worker, they are lazily consistent and the final update on the global table is always the most recent and updated value. We achieve this by considering updates only to entries corresponding to that worker, in both the global and the local table. Unlike the previous approach, this restriction also prevents updates to global table from blocking.

  In this context, performance and consistency are inversely proportional to each other and we aim to improve performance by sacrificing a bit on consistency. If a worker has to schedule based on thread load and makes a choice by always selecting the worker with the least loaded node from its local table, then it could result in every worker selecting the same node for remotely spawning an internal thread eventually overloading the selected node. This happens because the workers do not have a consistent view of

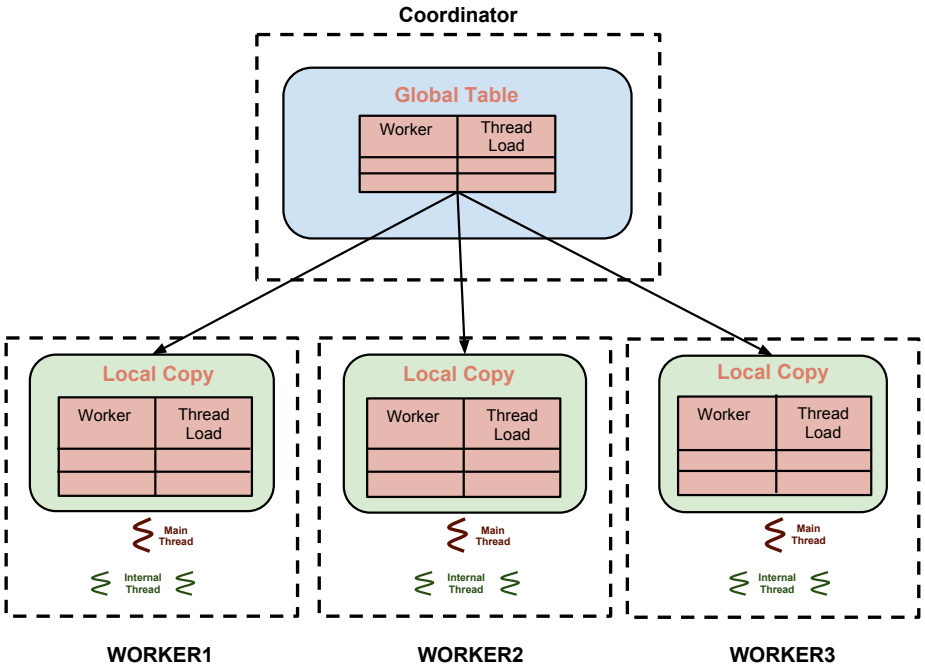

Figure 3.7: Communication for Worker to perform Scheduling from Local Information Table

the information table. To prevent this problem, workers make their choice based on weighted random distribution. More details about weighted random distribution is provided in the implementation section.

We now give an example with the aim to provide the reader a better understanding of how this approach prevents the need for any synchronization. Let us consider a scenario with one master and three workers. Say, the initial global information table of thread load looked like the one shown in Table 3.1 and local tables looked like the table shown in Table 3.2. All the tables are initially the same because the workers have received a fresh and most recent copy of the table from the master.

Master	
Worker ID	ThreadLoad
Worker-1	2
Worker-2	3
Worker-3	2

Table 3.1: Initial Global Table of Master

If now, worker-1 and worker-2 spawn an internal thread and decides using weighted random distribution to remotely launch the spawned thread

Worker-1		Worker-2		Worker-3	
Worker ID	ThreadLoad	Worker ID	ThreadLoad	Worker ID	ThreadLoad
Worker-1	2	Worker-1	2	Worker-1	2
Worker-2	3	Worker-2	3	Worker-2	3
Worker-3	2	Worker-3	2	Worker-3	2

Table 3.2: Local tables of Workers

on worker-2 and worker-3 respectively, i.e. worker-1 remotely launches on worker-2 and worker-2 remotely launches on worker-3, the new tables would look like the one showed in Table 3.3. Since worker-1 launches its internal thread on worker-2, it increments its local table entry for worker-2 to 4 and similarly worker-2 updates its local table entry for worker-3 to 3. Upon receiving the internal threads, worker-2 and worker-3 update their corresponding entries. The updated table is show in Table 3.4. It can be noticed that the local tables of all the workers are still inconsistent. However, since every worker updates its corresponding entry after receiving a thread, it ensures that they have an up to date view of themselves.

Worker-1		Worker-2		Worker-3	
Worker ID	ThreadLoad	Worker ID	ThreadLoad	Worker ID	ThreadLoad
Worker-1	2	Worker-1	2	Worker-1	2
Worker-2	4	Worker-2	3	Worker-2	3
Worker-3	2	Worker-3	3	Worker-3	2

Table 3.3: Local tables of Workers after thread launch

Worker-1		Worker-2		Worker-3	
Worker ID	ThreadLoad	Worker ID	ThreadLoad	Worker ID	ThreadLoad
Worker-1	2	Worker-1	2	Worker-1	2
Worker-2	4	Worker-2	4	Worker-2	3
Worker-3	2	Worker-3	3	Worker-3	3

Table 3.4: Local tables of Workers after receiving the threads

Master	
Worker ID	ThreadLoad
Worker-1	2
Worker-2	4
Worker-3	3

Table 3.5: Final Global Table of Master

Now, each worker updates only its corresponding entry in the global table i.e. worker-1 updates only the entry for worker-1 in the global table and so on. Thus, it avoids the need for any synchronization in the global table as

each worker corresponds to only one entry. The final table of the master is shown in Table 3.5. Whenever the master spawns another thread remotely, the local tables are updated asynchronously and all the workers have a consistent view all over again. This is our adopted approach for scheduling internal threads. Bottle necks in scheduling are round-trip latency, overhead in thread spawning, and decision load. Decision load becomes significant only with higher number of threads running and high rate of incoming threads. Round-trip latency and the overhead incurred in thread spawning always exists, but with hybrid scheduling we avoid them, without sacrificing the efficiency of allocation decision significantly and at the end it pays off.

This concludes the section which describes the various scheduling techniques supported by the middleware and provides a comparison of the different approaches for decentralized scheduling with its relative advantages and disadvantages. In the next section, we discuss about profiling an application along with the different metrics that can be measured.

3.4 Profiling

The system also supports profiling of applications for measuring various metrics. These metrics help the user choose the right scheduling heuristic to gain maximum performance. Profiling is done prior to executing the application on the middleware. It is not required to execute the application completely, instead sampling a part of the application is enough to obtain the necessary metrics. However, accuracy of the metrics are directly proportional to amount of data sampled. Most accurate information about the metrics is obtained by executing the application till they finish.

We have identified four important metrics that help to evaluate an application. They are as follows:

- **Dispersion of Thread Load** : Threads in an application may not be uniform and each thread may take an arbitrary time to finish its execution. Thread imbalance is a measure of variation in the execution times of the threads. If all the threads of an application have equal execution times, then the application is said to be balanced. On the other extreme, if the threads have extremely different execution times, they are said to be highly imbalanced.

  To the identify the same, the execution times of every thread is profiled and the obtained values are analyzed. We use standard deviation to identify the spread in execution times. A lower standard deviation means that all the values are clustered around the mean of observed data and a higher standard deviation indicates they are spread out. Because standard deviation is not a normalized value, it is not possible to make out any meaningful

information from this value alone. So, a standard score for every thread is computed and their distance from the mean is calculated. If more than fifty percent of standard score falls within one level of standard deviation we consider the application balanced, otherwise it is imbalanced. Since we do not know the number of threads in an application in advance, we cannot estimate the standard deviation of the population from a sample. So we simply calculate the standard deviation of the observed values and compute the standard score for the sample.

If we obtain a sample of execution times, the standard deviation of the sample is represented by S_n is:

$$S_n = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n}}$$

where $x_1, x_2, ..., x_n$ are the observed values of the sample items and $\bar{x}$ is the mean value of these observations. Standard score, denoted by Z, is given by:

$$Z = \frac{x_i - \bar{x}}{S_n}$$

If $Z \in [-1, 1]$, it means Z falls within one level of standard deviation. If more than fifty percent of Z values lie outside this range, the application is considered as imbalanced.

We give an example of why measuring the spread alone could give incomplete information. Say, we have two independent sets of values for thread inter-arrival times, $[1, 1, 1, 1, 1]$ and $[20, 20, 20, 20]$. The standard deviation for both these sets are zero, indicating they are clustered around their mean values but this does not give any information about the range although they are significantly different. A 2-tuple value with standard deviation and mean provides information about both the spread and range.

- **Thread Inter-arrival time**: A thread may be launched at any time during the lifetime of the application. Thread dynamism is a measure of the variation in inter arrival times between consecutive threads. If threads are launched regularly, the application has low thread dynamism. Extreme variation in inter arrival times imply high dynamism. The dispersion of inter-arrival time is calculated the same way as dispersion of thread load.

  If the period is high, the updates are less frequent and hence the load information is often incorrect, thus minimizing the scheduling efficiency. On the

other hand, if the period is low, there are too many unnecessary updates and it could overwhelm the workers. It is important to find an optimal value for periodic updates. To quantify the optimal period, we define two metrics: Scheduling efficiency (S_e) and Usefulness factor (U_f).

– **Scheduling efficiency**: Scheduling efficiency is a measure of the efficiency in scheduling. It is the highest if the scheduling decision taken is always based on the most updated information from the other nodes, i.e. it is inversely proportional to the period (p).

$$S_e \propto \frac{1}{p}$$

$$S_e = \frac{c}{p} \text{ where c is a constant and } p \geq c \, .$$

If t_m represents the time taken by a node to monitor its own load information and RTT represents the round trip time to the master, then,

$$\text{minimum time required to update the load information} = t_m + \frac{RTT}{2}$$

$$\Rightarrow S_e = \frac{2 * t_m + RTT}{2 * p} : S_e \in [0, 1] \tag{3.1}$$

we want to maximize Equation 3.1 to improve the scheduling efficiency. S_e is continuous in the interval [0,1]. By extreme value theorem, there is an absolute maximum. Because it is a strictly decreasing function, its absolute maximum is at the highest end point and not at critical points. Maximum value of the function is when $S_e = 1$. Thus, to achieve maximum scheduling efficiency, period is defined as:

$$\Rightarrow p = \frac{2 * t_m + RTT}{2}$$

– **Usefulness factor** : Usefulness factor is a measure of the usefulness of the periodic updates. It is highest when the number of updates are equal to the number of threads launched (one update for every thread launch). If N represents the total number of threads in an application and n_u represents the number of updates required, then

$$U_f = \frac{N}{n_u} \text{ where } n_u \geq N$$

If the last thread arrives at time t_l and the period is p, then

$$n_u = \frac{t_l}{p}$$

$$\Rightarrow U_f = \frac{N * p}{t_l} : U_f \in [0, 1] \tag{3.2}$$

To improve the usefulness factor, we need to maximize Equation 3.2. The function is continuous in the interval [0,1]. By extreme value theorem, there is an absolute maximum. Because the function is a strictly increasing function, absolute maximum is at the highest end point and not at critical points. Maximum value of the function is when $U_f = 1$. Thus to achieve maximum usefulness factor, period is defined as:

$$\Rightarrow p = \frac{t_l}{N}$$

In almost all cases, we can use the value for maximum scheduling efficiency as the CPU cycle consumption for monitoring is considerably low. The need to minimize usefulness factor would necessarily imply that the worker has very low CPU available for any computation, thus failing the purpose of scheduling a thread on a worker. It would only make sense to improve the usefulness factor for a worker with very little CPU to execute a memory intensive application.

In order to get an optimal value for period, we need to simultaneously maximize Equation 3.1 and 3.2. The optimal period is obtained when $S_e = U_f$.

$$\frac{2 * t_m + RTT}{2 * p} = \frac{N * p}{t_l}$$

$$\Rightarrow p = \sqrt{\frac{t_l * (2 * t_m + RTT)}{2 * N}} \tag{3.3}$$

Equation 3.3 represents the value for an optimal period. Depending up on the needs of the user, one of U_f or S_e can be sacrificed to obtain the required period.

- **Memory Consumption**: Total memory consumed for an application is both the heap and non-heap memory. Every thread in an application has its own stack space and it is not allocated from the heap. When a thread runs on a remote node with shared heap memory, the actual memory consumed on the local machine by the thread is only its stack memory. Thus, in this context, memory consumption would be the stack space consumed by every thread. Because it is hard to monitor the stack space consumed for every thread, we instead monitor the total memory consumed by the application. This affects the granularity of the metric but it however gives an idea of the memory intensity of the application. We finally classify the application as: high memory consumption or low memory consumption.

- **CPU Consumption**: CPU consumption is a measure of the number of CPU cycles consumed by an application. The profiler classifies the CPU consumption of an application as: high CPU consumption or low CPU consumption.

Summary

This chapter gave an introduction to Terracotta necessary for using the middleware and then proceeds to provide a high level view of the architecture along with the communication model between different components. first, the different scheduling techniques supported by the middleware were listed. Finally, the profiling capabilities along with the various metrics the system supports were explained. In the next chapter, we explain the implementation details of the middleware.

4 Implementation

This chapter describes the implementation details of the RATS middleware. We start by explaining the functionality of the most relevant and important classes of the system along with their corresponding UML diagrams. We then proceed to describe the bytecode instrumentations performed in the RATS middleware followed by a description of some of the techniques used for scheduling.

4.1 RATS module decomposition and Structure

This section describes in further detail the module decomposition and structure of some of the core components of the 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. The most important and relevant packages that will be discussed in the following subsections are listed below:

- *Singleton Package*
  - *Statistics*
  - *Coordinator*
  - *Scheduler*
- *ClusterThread*
- *Profiler*
- *StartMaster*
- *Worker*
  - *StartWorker*
  - *InformationServiceImplementation*
  - *SystemInfo*

4.1.1 Singleton Package

This package consists of classes that have only one instance and is accessible by both the master and worker through the interface provided by the Coordinator class. The choice of singleton over static modifier is because the former is a hair faster than the latter. The three classes in this package are Statistics, Coordinator and Scheduler.

4.1.1.1 Statistics Class

The Statistics class is responsible for storing information about the workers and maintaining state. It shares many data structures which are used by the Scheduler for making scheduling decisions. Figure 4.1 shows the UML diagram of the Statistics class. Some of the Important data structures of this class are:

Figure 4.1: Statistics Class

- *tidNodes*: The tidNodes is a java ConcurrentHashmap with threadID as key and worker address as value. This hashmap maintains information about all the threads that have been instantiated on the worker nodes.

- *nodesLoad*: The nodesLoad is a java ConcurrentHashmap with string representing the worker address as key and the number of threads in each worker as value. This map allows the scheduler to keep track of the number of threads launched on each worker.
- *nodeToMem*: The nodeToMem is a java ConcurrentHashmap with string representing the worker address as key and the available free memory in mega bytes as value. This hashmap is updated when necessary and helps the scheduler get information about free memory available on each worker.
- *nodeToCPU*: The nodeToCPU is a java ConcurrentHashmap with worker address as key and a java Double value representing the total CPU load of worker. This hashmap is also updated when necessary.
- *nodes_rr*: The nodes_rr is a java ArrayList that stores the address of every running worker in the cluster. This list helps the scheduler keep track of all the available workers and also in round robin scheduling.

4.1.1.2 Coordinator and Scheduler class

The Coordinator class contains a reference to both Statistics and Scheduler class. It contains methods responsible for registering the workers for RMI service and to set properties and delegate scheduling decisions to both these classes. The Scheduler class contains a reference to Statistics class for fetching system information and also accessing data structures to make scheduling decisions. index_rr is a java AtomicInteger that acts as an index to the list containing the worker address in Statistics class and is used for round robin scheduling. The different methods in this class are used for different scheduling techniques listed in the previous chapter. The scheduling technique used is specified by enum SchedMode while launching the master and worker. The UML diagram is presented in Figure 4.2.

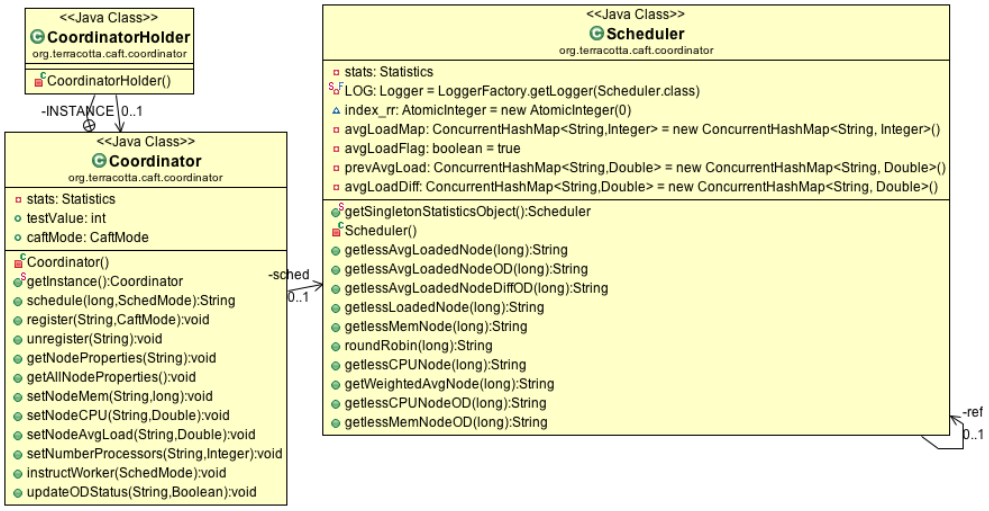

Figure 4.2: Coordinator and Scheduler Class

4.1.2 ClusterThread Class

The ClusterThread class is a common class and is present in both the master and worker components. Byte code instrumentations carried out in the application code for thread instantiations are replaced by this class. They contain methods that allows for remote method invocations through the ThreadService interface exposed in the worker. The runnableTargets field is a java Concurrent Hashmap that maps the threadID to its runnable object and is a Terracotta shared object. isMaster is a java boolean that identifies if the instance of the class is present in the master or the worker and is particularly useful is identifying whether to use distributed or centralized scheduling. The field InformationService is an interface exposed by the worker using RMI for making local copies of information for distributed scheduling. Figure 4.3 shows the UML diagram of the class ClusterThread.

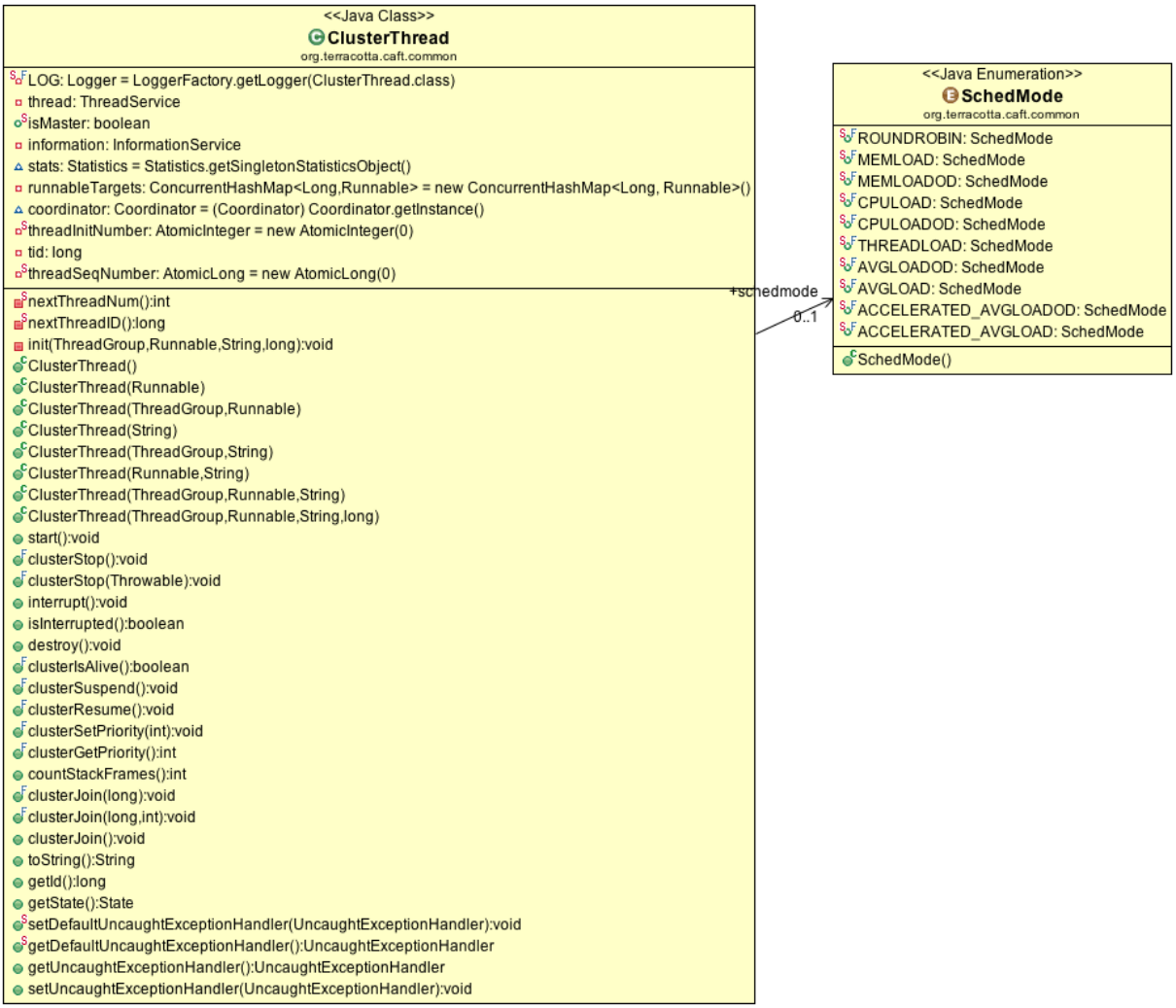

Figure 4.3: ClusterThread Class

4.1.3 Profiler

Figure 4.4 shows some of the relevant classes of the profiler. Class AddProfileThread is used to add necessary bytecode instrumentations. ProfileClassLoader is responsible for loading the classes in order to profile and ProcessInfo is used to gather statistics about the resource usage by the application. The class ThreadExit is used to identify when a thread exits and perform calculations so as to identify the dispersion of thread-workload and thread inter-arrival times.

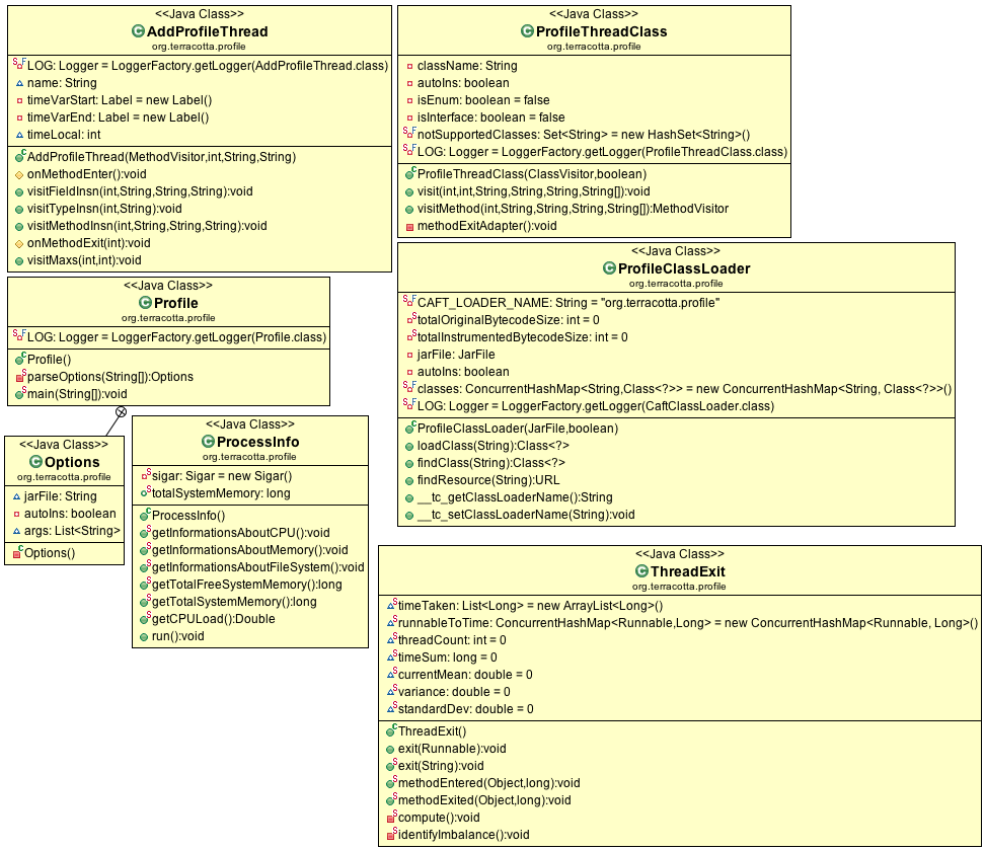

Figure 4.4: Profiler Class

4.1.4 Worker Package

The worker package is represented in the UML diagram shown in Figure 4.5. The StartWorker class contains the main class and starts the RMI ThreadService and InformationService and waits for requests from master. It also starts the monitoring process by instantiating the SystemInfo class. SystemInfo class uses the SIGAR library for gathering the system information about memory and CPU. The LocalScheduler class performs distributed scheduling using the local copy of information the worker received through the InformationServiceImpl class from the master. The worker takes certain command line arguments when started and

it is parsed using the args4j framework. The available options are:

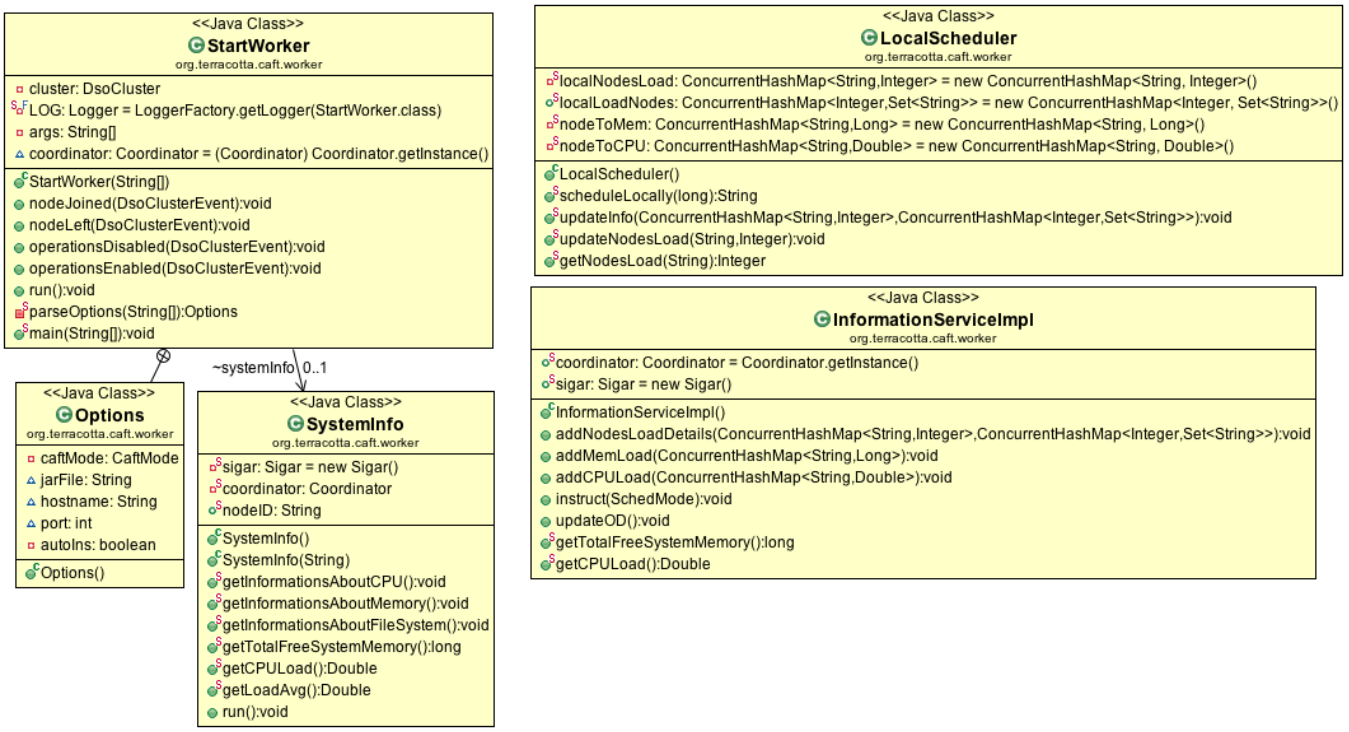

Figure 4.5: Worker Package

- **–mode**: Chooses the sheduler mode for distributed scheduling.
- **–jar**: The jar file of the application.
- **–hostname**: The hostname of the worker.
- **–port**: The port in which the worker will wait for connection from the master.
- **–autoins**: Instructs the middleware to add auto instrumentation for Terracotta configuration and avoids having to configure it in the .xml file.

4.1.5 StartMaster

The StartMaster class is represented by a UML diagram in Figure 4.6. This class contains the main method for starting the Master and is responsible for loading the application using a custom class loader. It requires a few parameters and these are specified as arguments. Like the worker, the arguments are parsed using the args4j framework. The arguments are listed below:

- **–jar**: The jar file of the application.

- **–schedmode**: Chooses the sheduler mode for centralised scheduling.
- **–autoins**: Instructs the middleware to add auto instrumentation for Terracotta configuration and avoids having to configure it in the .xml file.
- **–args**: The arguments required for the application to launch.

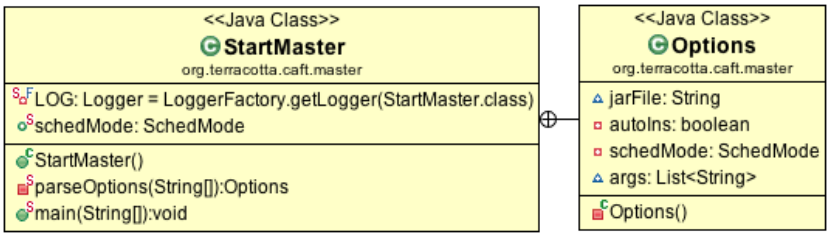

Figure 4.6: StartMaster Class

4.2 Bytecode Instrumentations

In this section we explain the bytecode instrumentations performed in the application code to provide transparency and achieve additional functionality for RATS. The ASM framework [9] is used for performing byte code transformations. Transparency is provided by byte code instrumentations already existing in the caft middleware. Additional transformations are required in the application byte code for both the scheduler and profiler to:

- **Identify Thread Exit**:

  Both stop() and destroy() methods of the java thread class are deprecated and the threads in the application code may never call these methods during exit. Thus, it is required to add additional code to identify when a thread completes execution. Thread exit needs to be identified when scheduling is based on the thread load of the worker. The data structures maintaining state information about the number of currently executing threads should be updated upon thread exit.

  To illustrate how a thread exit is identified, we present and explain the following bytecode:

Listing 4.1: Identify Thread Exit

```
1 new org/terracotta/rats/common/ThreadExit //create a new
    object instance of the ThreadExit class and push it on
    the stack
```

```
2  dup //duplicate the top value on stack
3  invokespecial org/terracotta/rats/common/ThreadExit.<init
       >()V  //invoke the constructor of the ThreadExit class
4  invokevirtual org/terracotta/rats/common/ThreadExit.exit()V
       // Invoke the exit() method the the ThreadExit class
```

We use the ASM framework to step through the byte code and identify the run() method of the thread. At the end of every run() method, we make a call to the method exit() of our custom class ThreadExit. This behaves like a notification upon every thread exit. The bytecode shown in Listing 4.1 is responsible for creating an instance of the class ThreadExit and invoking the exit() method of the custom class. Line 2 duplicates the top value of the operand stack and pushes it on to the stack. The reference to the object is duplicated as the next instruction consumes the value on the top of the stack. In this particular case the bytecode would work even if the reference is not duplicated as we do not need the reference for later use. Duplicating is helpful to extend the middleware with new instrumentations that require a reference to the ThreadExit class. Line 3 invokes the constructor of the class and Line 4 finally calls the exit method of the custom class. The exit() method then updates the state information table as required.

- **Remote Execution of Threads**:

  Scheduler needs to decide if a thread needs to be launched locally or remotely before scheduling. Thus, it is necessary to be able to capture the thread and replace it with a custom class so as to enable either remote or local execution of the threads. Listing 4.2 shows an example bytecode for executing a thread on a Runnable object.

Listing 4.2: Bytecode for launching a thread on a Runnable object

```
1  new java/lang/Thread //create a new object instance of the
       Thread class and push it on the stack
2  dup //duplicate the top value on stack
3  aload 0 //Push the object reference corresponding to the
       Runnable target
4  invokespecial java/lang/Thread.<init>(Ljava/lang/Runnable;)V
       //invoke the constructor of the Thread class that
       receives a Runnable target from the top of stack
5  astore 1 //Store the new object on index 1 of the local
       variable array
```

```
6 aload 1 //Push on to the stack the object reference
    corresponding to the instance of Thread class
7 invokevirtual java/lang/Thread.start()V //Begin thread
    execution
```

In order to capture the runnable object and make any decision before scheduling, we modify this bytecode to the one shown in Listing 4.3.

Listing 4.3: Bytecode for capturing Runnable object upon thread launch

```
1 new org/terracotta/rats/common/CustomThread //create a new
    object instance of the CustomThread class and push it on
    the stack
2 dup  //duplicate the top value on stack
3 aload 0  //Push the object reference corresponding to the
    Runnable target
4 invokespecial org/terracotta/rats/common/CustomThread.<init
    >(Ljava/lang/Runnable;)V  //invoke the constructor of the
     CustomThread class that receives a Runnable target from
    the top of stack
5 astore 1  //Store the new object on index 1 of the local
    variable array
6 aload 1 //Push on to the stack the object reference
    corresponding to the instance of CustomThread class
7 invokevirtual org/terracotta/rats/common/CustomThread.start
    ()V //Invoke start() method of the CustomThread class
```

The new bytecode replaces every reference of the Thread class to a CustomThread class. The constructor of the CustomThread class stores the reference to the Runnable object for the purpose of monitoring. The start() method of the CustomThread class takes necessary decisions if any before launching the thread and finally spawns a new thread with the captured Runnable object.

- **Compute thread execution time**:

  The time taken for execution of a thread is required for the profiler to measure thread imbalance. The run() method of the application code is modified to compute the execution time of a thread. This can be achieved by

computing the system time difference between the beginning and the end of the run() method. It is required to create a local variable to store the system time at the beginning so as to compute the difference at the end. But creating a local variable within a method would require recomputing the frames.

By default the local variable array of a frame contains a reference to the class object at index zero, followed by the parameters of the method. The local variables declared within the method take up the next indices. If we want to create a local variable within the method, it is necessary to store the variable in a index that will not overwrite any of the existing values in the array. The ASM framework provides a method that allows to obtain an index in the local variable array by recomputing the entire frame. Once a new index is obtained for the local variable, it can be stored for future reference.

Listing 4.4: On entering the run() method

```
1 invokestatic java/lang/System.currentTimeMillis()J  //Invoke
    the static method of the System class to obtain the
    current time in milliseconds
2 lstore 2 // Store current system time in the newly obtained
    index (say, 2) of the local variable array
3 new org/terracotta/profile/ThreadExit  //create a new object
    instance of the ThreadExit class and push it on the
    stack
4 dup //duplicate the top value on stack
5 invokespecial org/terracotta/profile/ThreadExit.<init>()V //
    invoke the constructor of the ThreadExit class
6 lload 2 //push on to the stack the system time
7 invokevirtual org/terracotta/profile/ThreadExit.
    methodEntered(J)V //Invoke the methodEntered() method
    with system time as parameter
```

Listing 4.4 shows the bytecode instrumentation added at the beginning of the run() method. This example assumes that the obtained index for the new local variable is two. The value returned by java/lang/System.currentTimeMillis() method is stored at the new index. Also, the bytecode invokes a method of the custom class ThreadExit with the start time as a parameter. This start time is used by the profiler to measure inter-arrival time between threads in order to measure thread dynamism.

Listing 4.5: On exiting the run() method

```
1 new org/terracotta/profile/ThreadExit  //create a new object
    instance of the ThreadExit class and push it on the
    stack
2 dup //duplicate the top value on stack
3 invokespecial org/terracotta/profile/ThreadExit.<init>()V //
    invoke the constructor of the ThreadExit class
4 invokestatic java/lang/System.currentTimeMillis()J  //Invoke
    the static method of the System class to obtain the
    current time in milliseconds
5 lload 2 //Push on to the stack the starting time of the
    thread
6 lsub // subtract the current time with starting time and
    push on stack
7 invokevirtual org/terracotta/profile/ThreadExit.methodExited
    (J)V  // invoke the methodExited() method of the
    ThreadExit class with the execution time as parameter
```

At the end of the run() method, the bytecode shown in Listing 4.5 is inserted. This bytecode computes the execution time of the thread and passes this value by invoking a method of the ThreadExit class. This value is later used for measuring the thread imbalance in the application.

This concludes the instrumentation section. Here we provided examples of bytecode to explain the instrumentations performed in the application code. These instrumentations are necessary for both the scheduler and profiler to add required functionality to the middleware.

4.3 Scheduling

In this section we discuss the implementation details of the scheduling heuristics. The load information of CPU and memory of the system is obtained using the library called SIGAR [1]. The information about load averages in Linux are obtained from the command line utility top. The current implementation does not support windows for getting information about the processor queues.

SIGAR runs on all the worker machines and monitors load based on the instruction provided by the master. When the master begins execution, it instructs the worker, through an interface exposed by the worker using the Spring framework [3], about the type of monitoring the worker needs to perform. The in-

struction can either be on-demand monitoring or periodic depending upon the scheduling heuristic used. If the scheduling heuristic is on-demand, the master requests the worker to update its load information in the statistics object. The master waits for a notification from the worker after it finishes the update and uses this information to schedule the threads. In case of scheduling based on periodic updates the worker constantly monitors and updates the load after specific periods and a thread is launched based on the most recent load of worker the master has seen.

CPU-load and Mem-load scheduling can be done based on the values obtained from SIGAR as these values are instantaneous. However, the values of load average obtained from top are not instantaneous. They are measured in three ranges as a moving average over one minute, five minute and fifteen minutes. In all Linux kernels the time taken for updating the moving average is five seconds. If multiple threads are launched instantaneously within a five second window, it is possible that all the threads are launched on the worker with a lower load average. There are two approaches that can circumvent this problem: Recompile the Linux kernel and modify the time taken for computing the moving average. In the Linux kernel file *sched.h*, the macro LOAD_FREQ defines the frequency for updating the load calculation time. By default it is set at 5HZ. 5HZ is 500 ticks and each tick corresponds to 10 milliseconds. This means that 5HZ corresponds to 5 seconds. By modifying the value of this macro it is possible to change the frequency of update. The following code segment shows the macro LOAD_FREQ and CALC_LOAD in the Linux kernel code, defined in *sched.h*.

```
#define FSHIFT 11               // nr of bits of precision
#define FIXED_1 (1<<FSHIFT)      // 1.0 as fixed-point
#define LOAD_FREQ (5*HZ+1)         // 5 sec intervals
#define CALC_LOAD(load,exp,n) \
          load *= exp; \
          load += n*(FIXED_1-exp); \
          load >>= FSHIFT;
```

However, in our implementation we resort to estimation based on the current values of load average.

Listing 4.6: Scheduling heuristic based on load average

```
1
2 if (loadAvgMonitor == true){
3         for each worker:
4                 if(avgLoad< NumberProcessors)
5                         avgLoadMap.put(nodeID, NumberProcessors -
                              avgLoad)
```

```
6                   else
7                           avgLoadMap.put(nodeID,1)
8
9           loadAvgMonitor=false
10  }
11  selectedNode = NodeID with maximum value in avgLoadMap
12  avgLoadMap.put(selectedNode, value-1)
13  if (all values in avgLoadMap.valueSet == 0){
14          loadAvgMonitor = true
15  }
```

In the pseudocode listed in Listing 4.6, the number of threads to be scheduled on a worker is inversely proportional to the load average. If the load average is less than the number of processors, only so many threads are launched to fill up the processor queue to the number of processors in the worker. Any further load monitoring is performed only after all these threads are scheduled. This overcomes the problem of multiple threads being scheduled on the same worker when they arrive instantaneously.

Listing 4.7 provides the pseudo code for scheduling heuristic accelerated-load-average. This heuristic is not very conservative and takes into account instantaneous changes in load average. New load average values are monitored immediately after scheduling the minimum number of threads possible based on the previous load average. This is achieved by remapping the lowest value in the avgLoadMap to 1. This will allow for scheduling the minimum number of threads possible while keeping the estimation correct and at the same time aiding in using a recent value of load average. Similarly, difference in load average is also inversely proportional to the number of threads to be scheduled. In order to achieve scheduling at least one thread on any worker, the highest difference is remapped to 1 and others are remapped accordingly.

Listing 4.7: Scheduling heuristic based on accelerated load average

```
1
2  if (loadAvgMonitor == true){
3          if(!first run){
4          for each worker:
5                   avgLoadDiffMap.put(nodeID, AvgLoadMap - prevAvgLoadMap)
6          }
7          if (first run || all values in avgLoadDiffMap.valueSet == 0){
8                   for each worker:
9                           if(avgLoad< NumberProcessors)
10                                  avgLoadMap.put(nodeID, NumberProcessors
                                        - avgLoad)
```

```
11                          else
12                                  avgLoadMap.put(nodeID,1)
13                  loadAvgMonitor=false
14          }
15          //remapping maximum value in loadAvgDiff to 1
16          if(all values in avgLoadDiffMap.valueSet != 0){
17                  for each worker:
18                          avgLoadMap.put(nodeID, loadAvgDiff.currentvalue
                                /max(loadAvgDiff.valueSet))
19          }
20          Copy values avgLoad to prevAvgLoadMap
21          // remapping minimum value in avgLoadMap to 1
22          for each worker:
23                  avgLoadMap.put(nodeID, avgLoadMap.currentvalue/ min(
                        avgLoadMap.valueSet))
24  }
25  selectedNode = NodeID with maximum value in avgLoadMap
26  avgLoadMap.put(selectedNode, value-1)
27  if (all values in avgLoadMap.valueSet == 0){
28          loadAvgMonitor = true
29  }
```

4.3.1 Local Scheduling

In order for the workers to perform local scheduling based on the information about number of threads in its local table, they use weighted random distribution. If all the workers simply choose a worker with the lowest number of threads, for scheduling, it can end up in a situation where a particular worker might get overloaded as the decision is local.

To perform weighted random distribution, each worker assigns a weight for all the other workers based on the information in its local table. The number of threads currently executing on any worker is inversely proportional to the weight assigned. After weights are assigned, the worker that needs to schedule a thread generates a uniform random number between zero to the total weight of all the workers in its table. Based on the generated random number, all the workers are looped and the total score of weights is maintained. Whenever the total score exceeds the random number that is generated, the current worker is our weighted random choice. This method consumes less memory as the set of weights are not expanded in memory.

A worker can also select itself for scheduling a thread. It is necessary to ensure that every worker has a correct view of the number of threads it is executing.

Since any worker up on receiving a thread, updates its local table, care is taken to ensure that scheduling a thread on itself does not update its current load twice. Eventually the tables need to be consistent and this can be ensured only when the master gets an updated view from all the workers. Each worker, apart from maintaining the weights in its local table, also maintains the difference in the number of threads it is executing as seen in global table and the local table. Whenever the difference exceeds a threshold (current implementation threshold of 1), the worker updates the global table.

Summary

This chapter began by explaining the functionality of the most relevant and important classes in the system along with their corresponding UML diagrams. Finally, we provided the implementation details of some of the important modifications necessary to perform byte code instrumentation, profiling and scheduling.

5 Evaluation

In this chapter, we describe the methodology used for evaluating the middleware and present the results obtained. 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, 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.

This chapter is organised as follows. The first section evaluates the correctness to ensure that the semantics of the application is not broken when executed on the middleware. In the next section, we measure the overhead incurred by the middleware in the form of time taken to launch a thread and increase in the size of bytecode caused by instrumentation. The following section evaluates the speed-up achieved when executing an application on top of the middleware. The next section compares the different scheduling algorithms and classifies these algorithms based on application behavior.

5.1 Correctness

It is important to ensure that executing the application on top of the middleware does not break the semantics and functionality of the application. Correctness is measured for three multi-threaded applications: MD5 hashing, web crawler and Fibonacci number generation using Binet's Fibonacci number formula. In order to verify the correctness, each of these applications are executed on a JVM and on the middleware.

In MD5 hashing, multiple messages are hashed using MD5. Their corresponding hashes are compared for equality when executed on a JVM and when executed on top of the middleware. Table 5.1 provides a comparison of results obtained from MD5 hashing. Fibonacci number generation is verified by comparing the equality of the fibonacci sequences generated on a single system and on top of the middleware. Table 5.2 provides a comparison of the fibonacci sequences generated. Both of these applications are embarrassingly parallel and share no data among threads.

MD5 Hashing	JVM	RATS
Number of Messages	50	50
Number of threads	2	2
Hash match	100% match	
Number of Messages	200	200
Number of threads	2	2
Hash match	100% match	
Number of Messages	400	400
Number of threads	5	5
Hash match	100% match	

Table 5.1: Correctness verification of MD5 hashing

Fibonacci generation	JVM	RATS
Number of sequences	10	10
Number of threads	1	1
Sequence match	100% match	
Number of Sequences	300	300
Number of threads	2	2
Hash match	100% match	
Number of Messages	800	800
Number of threads	4	4
Hash match	100% match	

Table 5.2: Correctness verification of Fibonacci number generation

Web crawler	JVM	RATS
Number of threads	2	2
Depth	1	1
Links match	100% match	
Number of threads	4	4
Depth	2	2
Links match	100% match	
Number of threads	5	5
Depth	3	3
Links match	100% match	

Table 5.3: Correctness verification of Web Crawler

It is necessary to evaluate correctness when threads share data and require synchronization. A multi-threaded web crawler is used for an application requiring synchronization between threads. The web crawler ensures that the same link is never crawled twice. So, it maintains a list of visited URLs and a link is

crawled only if it is not present in the list. It also maintains two URL queues for storing links obtained at different depths. Threads need to synchronize among themselves before pushing and popping elements from the queue. The web crawler is executed on a single system and on top of the middleware. The website www.ist.utl.pt was crawled with different number of threads and multiple depth levels. Table 5.3 compares the results obtained. Also, the webcrawler uses hybrid scheduling as threads spawn multiple threads. A thread is responsible for populating the queues and when it finishes, it spawns multiple internal threads to crawl the links in the queue. These results indicate that the middleware allows for remote launching of threads without breaking the functionality of the application.

5.2 Overhead Incurred

Executing java applications on the middleware incurs an additional overhead of increase in the size of bytecode and delay in launching a thread. Depending on the scheduling algorithm used, there may be additional overhead in updating load information. Periodic updates of load information are asynchronous and it is difficult to directly measure the overhead incurred. For this reason and to maintain generality, this section measures the overhead involved in launching a thread and the increase in the size of bytecode.

Bytecode Instrumentation Overhead		
Original size	**After Instrumentation**	**Percentage Increase**
3367 bytes	3539 bytes	5.04 %

Thread Launch overhead			
No. of threads	**Avg. time to launch a thread**	**Total overhead**	**Percentage Increase**
2	0.39 secs	0.78 secs	-
4	0.39 secs	1.58 secs	100.5 %
8	0.39 secs	3.16 secs	100 %
16	0.39 secs	6.27 secs	98.4 %

Table 5.4: Overhead for MD5 Hashing

It can be observed from Table 5.4 that the size of bytecodes have increased because of additional instrumentations to provision transparency, remote launching of threads, monitoring runnable objects and capturing thread execution times. This increase in the size of bytecode does not consider the instrumentations done by Terracotta. As the number of threads in the MD5 hashing application doubles, the total overhead incurred for launching threads also doubles. The overhead is considerable and indicates that the middleware is not suited for applications that are compute non intensive.

From Table 5.5, it can be seen that the percentage increase in the size of byte code added by the middleware is only 6.8%. However, the average time taken to

Bytecode Instrumentation Overhead		
Original size	**After Instrumentation**	**Percentage Increase**
6275 bytes	6702 bytes	6.8 %

Thread Launch overhead			
No. of threads	**Avg. time to launch a thread**	**Total overhead**	**Percentage Increase**
2	0.52 secs	1.04 secs	-
4	0.51 secs	2.04 secs	96.15 %
8	0.51 secs	4.08 secs	100 %
16	0.51 secs	8.16 secs	100 %

Table 5.5: Overhead for Fibonacci generation

launch a thread for fibonacci generation is different from the average time taken to launch a thread for MD5 hashing. Apart from scheduling decision and RTT, it also involves the time taken to store the runnable object in the virtual heap. As the size of the runnable object increases, the overhead also increases. The total time taken to launch threads doubles as the number of threads double. In order to achieve any gain in performance, the gain obtained must be greater than the overhead incurred. Otherwise, the middleware deteriorates the performance of the application. The middleware is thus suited for compute intensive applications.

5.3 Execution Time

5.3.1 Fibonacci number generation

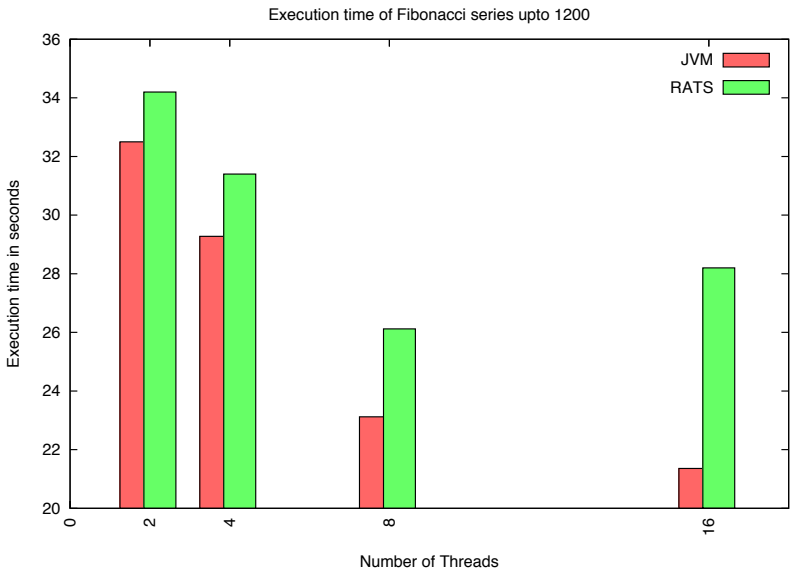

Figure 5.1: Execution time for Fibonacci number generation.

Fibonacci number generation is configured to compute the first 1200 numbers of the Fibonacci sequence, with number of threads directly proportional to the number of processors available. The time taken with two, four, eight and sixteen threads is measured. The application is tested in a standard local JVM, for comparison with the distributed solution. The distributed solution executes the application with two workers. Figure 5.1 compares the execution times for different number of threads.

It can be seen from the figure that the middleware decreases the performance of the application. Fibonacci number generation is not compute intensive, hence the overhead incurred in launching the threads and gathering the results, exceeds the gain obtained in execution time. As the number of threads double, the execution time taken using the middleware increases drastically. Since the application is evaluated for 1200 numbers, the load of the application remains a constant. This means that, increasing the number of threads decreases the load per thread. As the load per thread decreases, the gain obtained by distributed execution, decreases, but the overhead incurred in launching threads increases. Hence, distributed execution of the application deteriorates the performance, when the number of threads increase for a constant load. Fibonacci is highly recursive and it ends up allocating pages for stacks, which on a fast processor leaves a lot of available CPU for other threads. There is a lot of context switching for very small functions.

5.3.2 Web Crawler

For measuring the performance of the web crawler, it was tested under different scenarios. The number of websites crawled were increased for each evaluation. Number of threads for crawling within a single website is maintained as a constant at three. These three threads require synchronization among themselves. Every website is crawled up to a depth of two. Execution time is measured for crawling ten, twenty and thirty websites. Since the difference in execution times vary extremely, results obtained for crawling ten websites is plotted separately and is shown in Figure 5.2.

As the number of threads increase within a single JVM, the thread idle time increases, because of contention for the available processors. Any gain is achieved by minimizing this idle time. By distributing the threads on multiple workers, the idle time is greatly reduced as the number of threads per processor decreases. From Figure 5.2, it can be seen that the time taken to crawl ten websites until depth two is only three seconds. The overhead incurred in launching ten threads alone exceeds three seconds. Thus, any gain obtained in minimising the idle time is not visible. But as the size and the number of websites increase, the time taken to crawl them also increases. Figure 5.3 shows the improvement gained in performance when executed using the middleware. As the number of workers increase, the execution time also decreases. These results indicate that the distributed so-

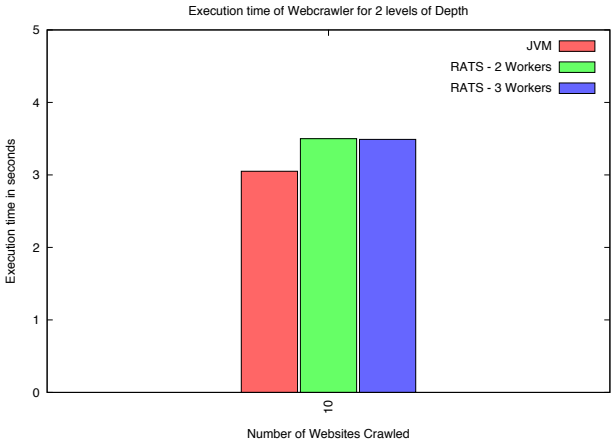

Figure 5.2: Execution time for web crawler - 10 websites.

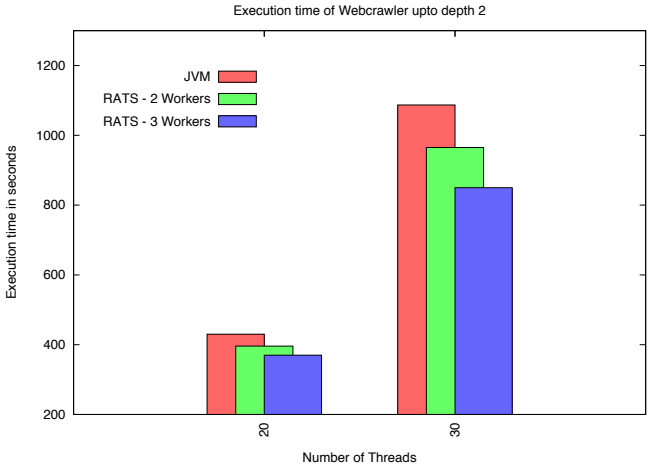

Figure 5.3: Execution time for web crawler - 20 and 30 websites.

lution scales linearly.

5.3.3 MD5 Hashing

For measuring the execution time of MD5 hashing, the number of messages to be hashed by each thread is kept at a constant of five hundred messages. The performance is compared by executing the application on a single machine and using the middleware. Two and three workers are used for the purpose of comparison and time taken with five and ten threads is measured. Figure 5.4 shows the results obtained.

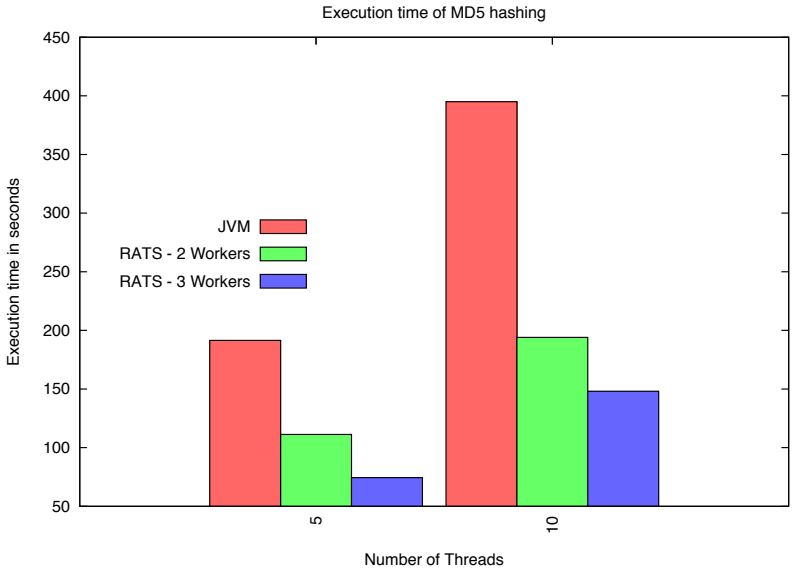

Figure 5.4: Execution time for MD5 hashing

MD5 hashing is a CPU-intensive process. As can be seen in Figure 5.4, when the number of workers increase, the time taken to execute the application decreases. This is because the available CPU increases and hence a speed-up is obtained.

5.4 Comparison of Scheduling Heuristic

In this section, the different scheduling algorithms are evaluated with different application behaviour. All the experiments are carried out with MD5 hashing and the application behaviour is modified in order to be able to classify applications based on its thread characteristics. To understand how different scheduling algorithms behave with different application characteristics, thread behaviour is modified in an ordered fashion. The application is then profiled using the profiler in order to test the correctness of the profiler. The characteristics varied for the application are: work load of each thread and thread inter-arrival times. The following subsections were evaluated on a dedicated cluster where there are no other workloads with a period equal to provide maximum scheduling efficiency.

5.4.1 Low dispersion of thread workload and low inter-arrival times

We begin the experiment by letting each thread compute a similar number of messages. The dispersion of thread workload is very low, as they perform similar amount of work. The threads are launched one after another with very little delay, i.e. they arrive almost instantaneously.

- **High thread workload**

  The dispersion of thread workload is low and clustered around a high range of values. In order to evaluate the performance under extreme scenarios, the workload needs to be clustered around extreme high values. Any workload that cannot finish its execution before all the threads are scheduled is considered an extreme high value. For this experiment the chosen workload is between 500- 550 messages. The results obtained are shown in Figure 5.5.

  It can be seen from the results that round-robin takes the least time to finish execution and the scheduling heuristic thread-load has a comparable performance. Round robin performs better because the threads have equal workload and are equally spaced with their arrival times. The scheduling heuristic thread-load schedules jobs exactly in the same manner as round-robin but since it involves a slight overhead at maintaining state information, round-robin performs better in this case. CPU load - on demand incurs the overhead of obtaining the load information from every worker before making a decision and does not perform as good as round-robin. On the

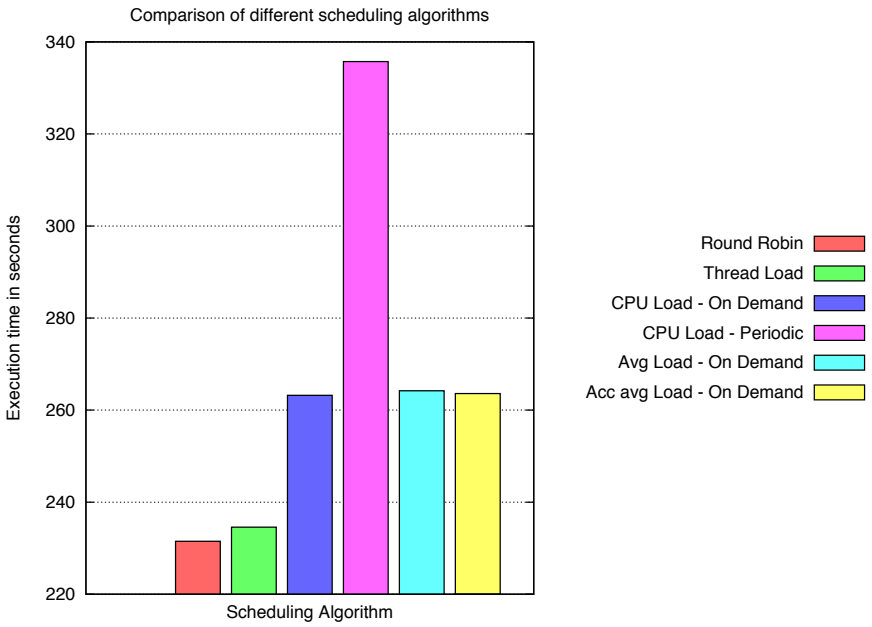

Figure 5.5: Execution time for different scheduling heuristics

other hand, CPU load - periodic takes a much higher time to finish execution. Although the scheduling decisions are based on the workers CPU load, the information obtained is not the most recent. Here, in this case, the least period to perform an update of load information is higher than the inter-arrival time some of the threads. As a result, some workers get overloaded with threads. Avg-load and Accelerated-avg-load on demand has comparable performances with CPU-load on demand. Since all the threads perform similar workload, estimation based on the size of processor queue will result in a scheduling more or less similar to round-robin as the initial queue size is the same. In this case, there is no difference between accelerated-load-avg and avg-load scheduling because there is no previous load in the system. However, it is not possible to give a definite comparison of the performance with the other scheduling heuristic.

- **Low thread workload**

  The dispersion of thread workload is low and clustered around a low range of values. In order to evaluate the performance under an extreme scenario, the workload needs to be clustered around extreme low values. Any workload that can finish its execution before all the threads are scheduled is considered an extreme low value. Since the threads arrive almost instantaneously, an extreme low value means almost no workload and it is not practical. For this reason, the workload is set considerably low at around 10-15 messages. The results obtained are shown in Figure 5.6.

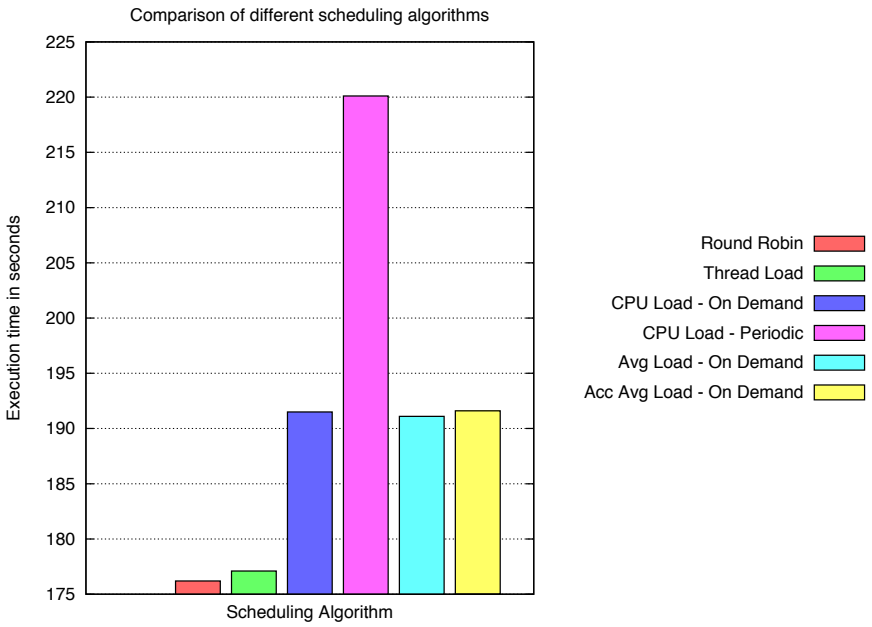

Figure 5.6: Execution time for different scheduling heuristics

The results obtained are similar to the previous results, except that the the time taken for execution is considerably lower. This is because the workload is lesser. The combination of thread workload with inter-arrival times does not affect any of the scheduling algorithms and as a result the behavior of the scheduling algorithms remain the same.

5.4.2 Low dispersion of thread workload and high inter-arrival times

The application is modified to make threads perform similar amount of computation with threads arriving after a large amount of time. In other words, the dispersion of thread work load is low and the time taken for arrival of threads is high, ranging between three seconds and twenty seconds.

- **High thread workload**

  The dispersion of thread workload is low and clustered around a high range of values. The workload is clustered around extreme high values. Any workload that cannot finish its execution before all the threads are scheduled is considered an extreme high value. For this experiment the chosen workload is between 500- 550 messages. The results obtained are shown in Figure 5.7.

  Since the jobs are of similar sizes, scheduling heuristic, thread-load and round-robin take similar amount of time to finish execution. It can be no-

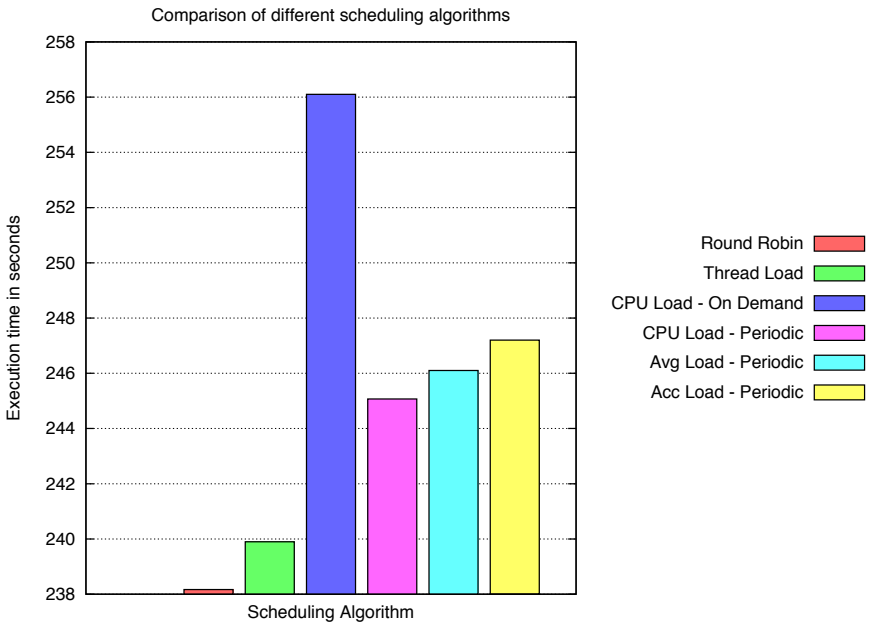

Figure 5.7: Execution time for different scheduling heuristics

ticed that CPU load - periodic finishes faster than CPU load - on demand unlike the previous scenario. This is because the inter-arrival times are high. The lowest period is enough time to update the state information asynchronously as opposed to synchronous update for on-demand. CPU-load periodic takes a little longer than thread-load and round-robin. Upon analysis, we found that the CPU load almost saturates by the time most of the threads are scheduled. Because CPU load information is instantaneous, minor differences in this information affect the scheduling. This effect of minor variation in CPU load is similar to the one shown in Figure 5.11. Load-avg and accelerated-load-avg are based on estimates and do not work as well as CPU-load sched. The values are not instantaneous and as a result, the information about the size of processor queue is not accurate. The result shown here is not representative of its behavior in every case similar to this application characteristic and varied each time the application was executed with different number of threads.

- **Low thread workload**

  The dispersion of thread workload is low and clustered around a low range of values. The workload needs to be clustered around extreme low values. Any workload that can finish its execution before all of the threads are scheduled is considered an extreme low value. The workload is set considerably low at around 10-15 messages. The results obtained are shown in Figure 5.8.

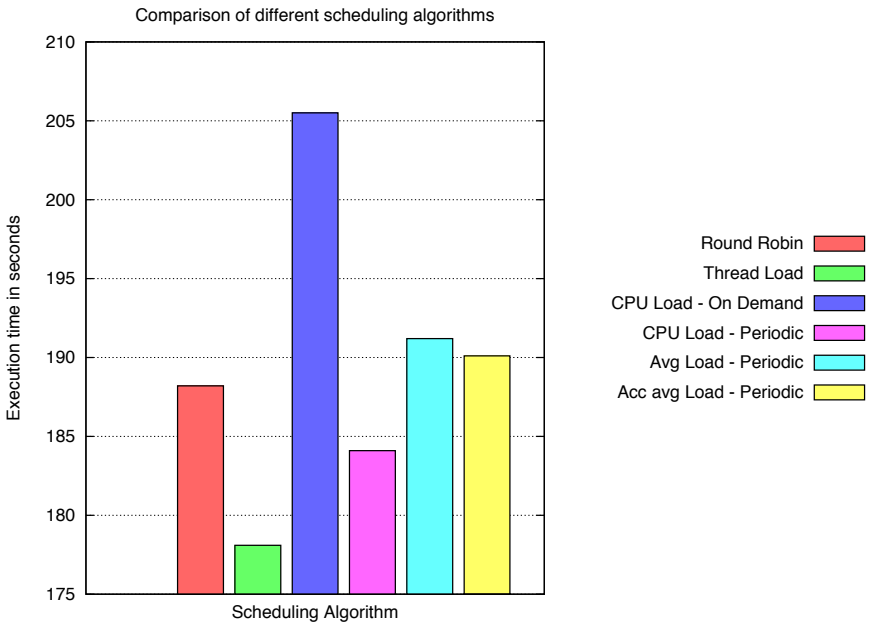

Figure 5.8: Execution time for different scheduling heuristics

From the figure it can be seen that thread load scheduling takes the least amount of time to finish computation. Because the thread workload is dispersed around very low values, the scheduling heuristic gets a more up dated view of the thread status in terms of completion. Before all the threads are scheduled, some of the threads finishes its execution and the thread load heuristic is able to make a better decision than round robin. The overhead incurred by monitoring the CPU load increases the time taken to schedule threads and hence CPU-load heuristic performs worse than round-robin and thread load heuristic.

5.4.3 High dispersion of thread workload and high inter-arrival times

The application is modified to make the threads perform different amount of workloads. The workload in this case is the number of messages to hash, and it is varied between one and two thousand randomly. We use an ordered seed, in order to compare the performance with different scheduling heuristics. The threads in the application arrive between three and twenty seconds. The results obtained are shown in Figure 5.9.

From the results, it can be seen that the scheduling heuristic CPU load -periodic consumes the least amount of time to finish execution. The thread inter-arrival time is highly spread out. Hence, the periodic update has enough time to provide the master with the most recent view of the state of workers. The workload of threads are unequal and scheduling heuristic based on CPU load, tries to

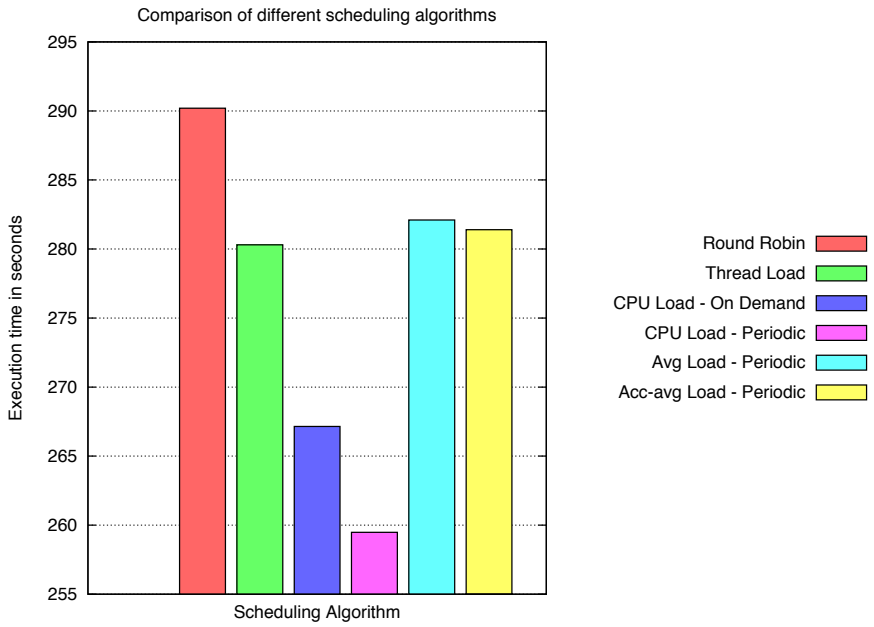

Figure 5.9: Execution time for different scheduling heuristics

greedily equate the CPU load consumed by each thread. Although the scheduling heuristic has no information about the time taken to finish a job, some jobs finish much earlier than all the threads are scheduled. This information helps the heuristic make a better decision and spreads out threads of high and low workloads equally among the different workers. CPU load - on demand, performs better than the scheduling heuristic thread-load and round-robin. Thread-load on the other hand performs better than round robin because it adapts to the state of threads running on the workers. It no longer behaves like round-robin as it updates the number of threads on workers as they finish execution and is thus able to make better decision than round robin. Also, CPU-load heuristic is affected by the minor variation as the values saturate. Avg-load and accelerated-average-load are also able to notice when threads finish executing, but since the values are a moving average, the information obtained is only a very minor difference in the size of processor queues and as a result the scheduling continues in a fashion more or less similar to thread load. This is because the minor differences when rounded to an integer (implementation specific), remains the same as the previous load unless there is a huge difference.

5.4.4 High dispersion of thread workload and low inter-arrival times

The application is modified to have a high dispersion in the thread work-load and threads are made to arrive quickly, one after another. For the experiment, the work load of threads are varied between one and two thousand messages and threads arrive almost instantaneously. The results obtained are shown in Figure

5.10.

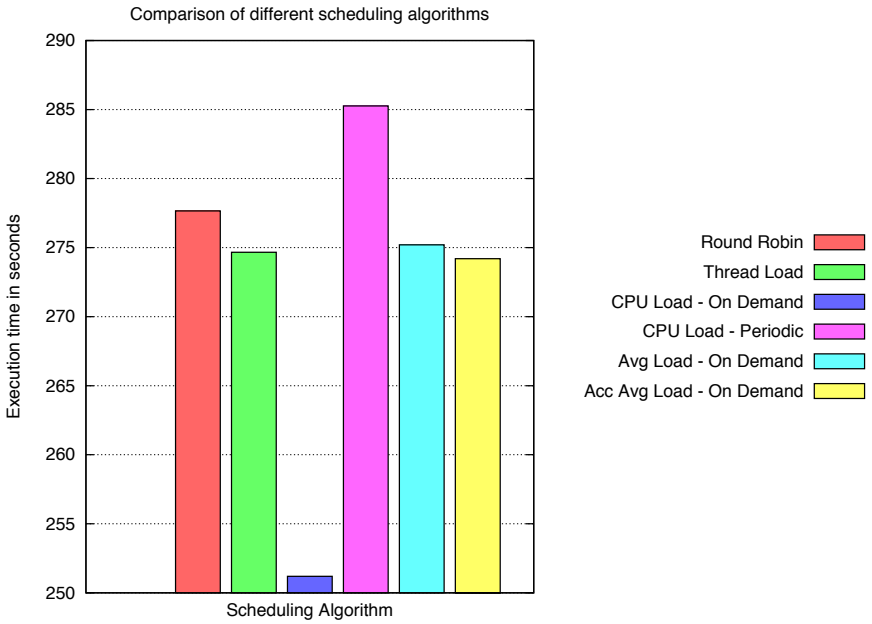

Figure 5.10: Execution time for different scheduling heuristics

CPU load - on demand performs better than any of the other scheduling heuristics, because the work load of threads are unknown and this heuristic aims to greedily equalise the CPU load of different workers as and when threads arrive. CPU load - periodic takes a much higher time as the lowest possible period to update the state information is higher than the inter-arrival time between some of the threads. Most of the threads are hence scheduled on the same worker. Thread-load and round-robin have similar performance.

5.4.5 Non-uniform cluster

A non-uniform cluster is shared between multiple processes or users and can have other workloads running along with our middleware, exemplifying the case of a multi-tenant cloud-like infrastructure. This results in a varied load among the different machines in the cluster. In the following subsection we show how certain applications can affect the behavior of our scheduling algorithms.

When the cluster is shared between multiple users or processes, there is a variation in the existing load of different machines. We evaluate the impact of this load on different scheduling heuristics. Since the scheduling heuristic round-robin and thread-load does not take the load of the system into account, their performance deteriorates as they schedule threads in a circular fashion. Machines with very high load may end up getting the same number of threads as machines with very low load and thus degrade the performance of the application. On the

other hand, scheduling heuristic based on CPU-load takes into account the CPU load of the machines and tries to equalise the load among different machines. Figure 5.11 shows that the CPU-load scheduling heuristic tries to greedily adapt to the current CPU load of the two workers. Although worker2 starts at a very high previous load of 0.6, the CPU load almost remains equal during the period of execution. On the other hand, scheduling using thread-load shown in Figure 5.12 schedules equal number of threads on both the workers despite the variation in workload and worker3 finishes execution relatively earlier, while worker2 takes a considerably longer time to finish execution.

Figure 5.11: CPU Load over time for scheduling based on CPU load

Figure 5.12: CPU Load over time for scheduling based on thread load

Thus, it is expected that the scheduling heuristic CPU-load always tries to equalise the CPU load of different workers. However, Figure 5.13 shows a situation where this may not always hold true. MD5 hashing application was executed on a cluster with two workers. Worker2 was already executing an I/O intensive application and worker3 was not loaded. Worker2 starts at a previous load of 0.3 while worker3 starts at a load of 0.

It can be seen from the figure that the CPU load of both these workers vary considerably. To explain this variation in CPU loads despite trying to equalize, we carried out an experiment with both the workers executing the same CPU-intensive application without our middleware. While worker2 had an I/O intensive process alongside, worker3 executed only our application. The results are shown in Figure 5.14. It should be be noticed that despite running the same CPU-intensive application the CPU load of worker2 does not go beyond 0.6, while with the same application the CPU load of worker2 reaches 0.9. The I/O intensive application remains idle during most of its CPU time as it either waits or performs an I/O operation. Because all the processes are scheduled by the processor for a definite quanta, the overall CPU usage is influenced by the I/O intensive process. Although our application is CPU-intensive the overall CPU usage continues to remains deceptively low. In Figure 5.13, a initially threads are launched on worker3 till the CPU load of worker3 rises. Once the CPU-load of worker3 increases be-

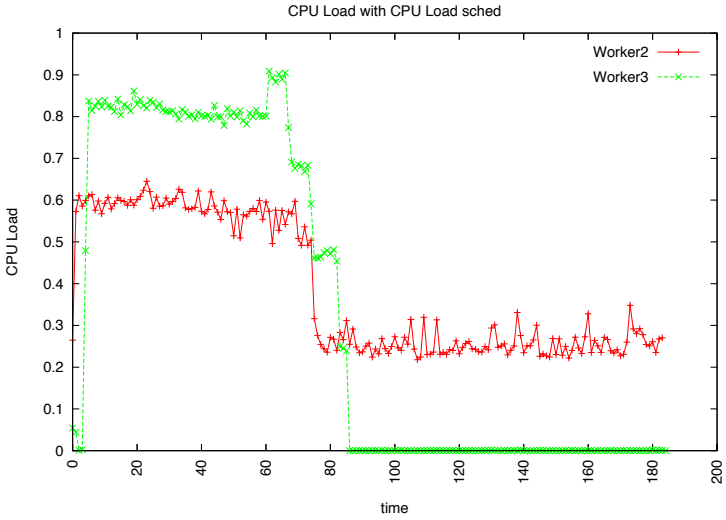

Figure 5.13: CPU Load over time for scheduling based on CPU load alongside an I/O intensive process

yond that of worker2, threads are scheduled on worker2 till the loads of both these workers become equal but since the load of worker2 does not rise beyond 0.6, all consequent threads are launched on worker2. For applications with many number of threads, the scheduling can prove rather detrimental than useful as it considerably affects the response time.

Figure 5.14: CPU Load over time for same application

To circumvent this problem we need a metric that provides a true view of the system properties. One such metric is the load average. The load average tries to measure the number of active processes at any time. As a measure of

CPU utilization, the load average is simplistic, but far from useless. High load averages usually mean that the system is being used heavily and the response time is correspondingly slow. By combining CPU load along with load average, it is possible to get a real view of the system. The impact of the scheduling heuristic CPU-load on the load average is shown in Figure 5.15. It can be noticed that there is a considerable variation in the load averages and the load average of worker2 rises drastically as more and more threads get scheduled on worker2.

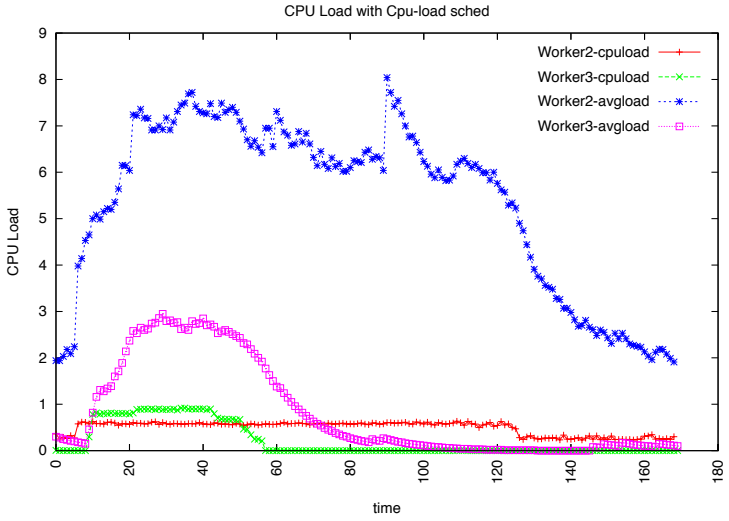

Figure 5.15: Impact of CPU-load sched on load average and CPU utilization

The results obtained by the heuristic load-average is shown in Figure 5.16. The load averages of both the workers are very similar and it thus mitigates the problem of overloading a worker that runs an I/O intensive process. However, load-average scheduling is very conservative as it always takes into account the previous load of the system. From the figure it can be seen that worker2 finishes execution at around 60 while worker3 finishes execution only at time 100. This is because of the conservative nature of the scheduling heuristic.

Scheduling heuristic accelerated-load-average on the other hand takes into account the instantaneous change in load averages caused by the application and thus performs better than load-average scheduling. The results obtained are shown in Figure 5.17. This problem of conservative scheduling is mitigated by accelerated-Load-Average. It can be seen that the execution finishes earlier because more threads are launched on worker2 at the expense of tolerating minor differences in overall load average. This is possible because the scheduling heuristic does not consider the previous load in the system and in effect only accounts for the changes caused in load average due to the application that is being executed.

Non-uniform cluster has the same behavior with respect to on-demand and periodic updates of resource usage as that of uniform cluster. It is important to note that the behavior of scheduling heuristic round-robin and thread-load are unpredictable in a non-uniform cluster with any kind of application. Similarly

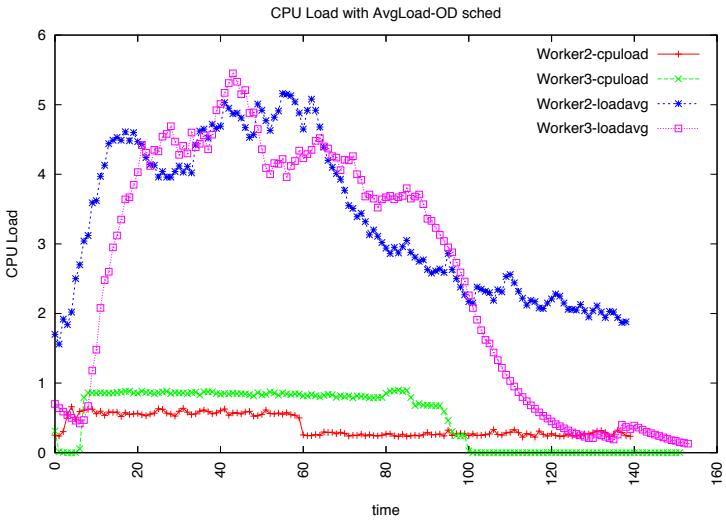

Figure 5.16: Impact of load-avg sched on load average and CPU utilization

with an existing I/O or network intensive load, the scheduling heuristic CPU-load becomes irrelevant and accelerated-load-average performs the best.

5.5 Memory usage

In order to stress test the system for scalability in terms of memory, we developed a memory intensive application aimed to generate a synthetic memory load by creating as many objects as possible and simply iterate through them. In order to test the application, we allocated 7GB of memory for the JVM and ran the application on a single JVM and on top of our middleware using 2 workers. Each thread created an object with integer array of size 10000 and the size of integer is assumed to be 4 bytes. The results obtained are shown in Figure 5.18.

It can be seen from the results that for a single JVM, approximately 4GB of memory was allocated and beyond that the system gave an out of memory error. But using the middleware, the system scales with respect to memory as the cluster now behaves like a virtual memory with the Terracotta server responsible for handling large heaps. As a result, we were able to allocate up to 15GB of data using the middleware. This result shows that the application scales in terms of memory.

5.6 Application Modeling

Based on the results obtained, we model application based on the characteristics of the cluster and the thread characteristics of the application. The cluster characteristics are classified into three categories: dedicated or uniform cluster, highly unbalanced cluster, and I/O intensive cluster. Table 5.6 classifies the application according to the most suited scheduling algorithm on a dedicated cluster and

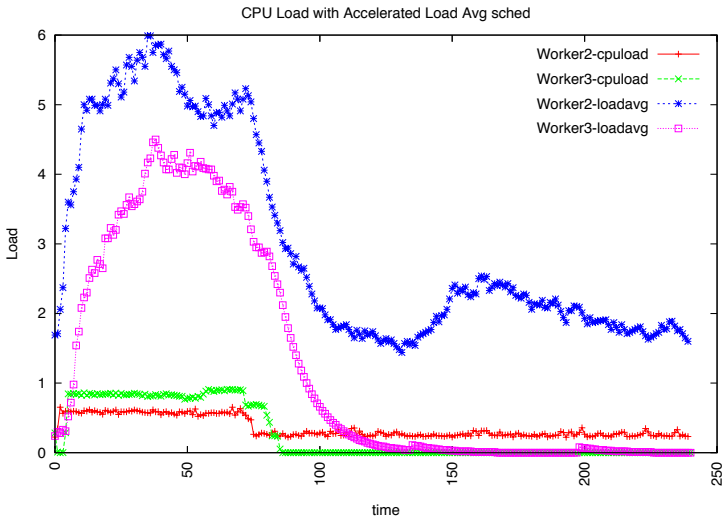

Figure 5.17: Impact of accelerated-load-avg sched on load average and CPU utilization

Table 5.7 classifies the application based on the previous load in a non-uniform cluster.

Summary

In this chapter, we described the methodology used for the evaluation of the proposed middleware and presented the results obtained. We tested three multithreaded 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. First, we assessed correctness to ensure that the semantics of the applications is not broken when executed on the middleware. Next, we measured the overhead incurred by the middleware in the form of time taken to launch a thread, and the increase in the size of bytecode caused by instrumentation. Finally, we evaluated the speed-up achieved when executing an application on top of the middleware, and compared the different scheduling algorithms, classifying them based on application behavior.

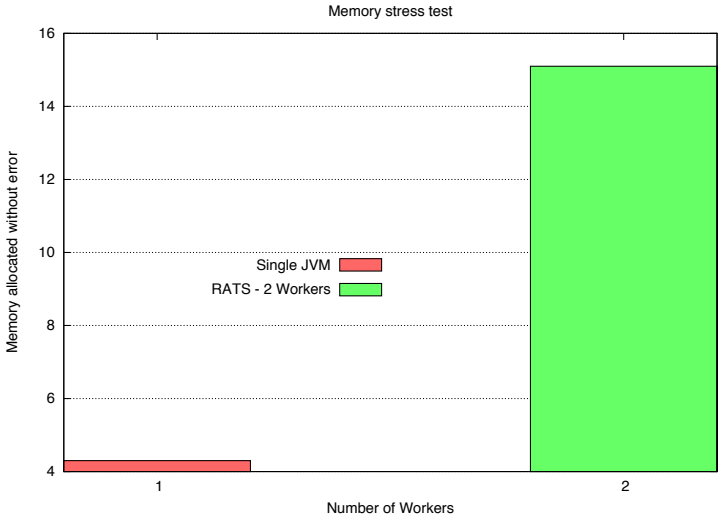

Figure 5.18: Memory stress test

Application Type	**Thread Workload**	**Inter Arrival time**	**Execution time of Scheduling Heuristic**
CPU Intensive	Low Dispersion, high load	low	RoundRobin ≈ Thread-load < Cpuload-OD < Cpuload-Periodic Load-avg and Acc-Load-avg (uncomparable)
	Low Dispersion, low load	low	RoundRobin ≈ Thread-load < Cpuload-OD< Cpuload-Periodic. Load-avg and Acc-Load-avg (uncomparable)
	Low Dispersion, high load	high	RoundRobin ≈ Thread-load < Cpuload-Periodic < Cpuload-OD. Load-avg and Acc-Load-avg (uncomparable)
	Low Dispersion, low load	high	Thread-load < RoundRobin < Cpuload-Periodic < Cpuload-OD. Load-avg and Acc-Load-avg (uncomparable)
	High Dispersion	high	Cpuload-Periodic < Cpuload-OD < Thread-load ≈ Load-avg and acc-Load-avg < RoundRobin
	High Dispersion	low	Cpuload-OD < Thread-load ≈ Load-avg and acc-Load-avg < Cpuload-Periodic < RoundRobin

Table 5.6: Application Modeling on a dedicated cluster

Previous Load	Application Type	Execution time of Scheduling Heuristic
Non I/O or network intensive	CPU intensive	CPU-load < Accelerated-Avg-Load < Avg-Load. (Others are irrelevant)
I/O or network intensive	CPU intensive	Accelerated-Avg-Load < Avg-Load. (Others are irrelevant)

Table 5.7: Application Modeling on a Non-uniform cluster

6 Conclusion

When the workstations in a cluster work collectively to provide the illusion of being a single workstation with more resources, we refer to this as a Single System Image. With such an abstraction, the developer is completely oblivious to the issues related to distributed computing and writes an application just like any other application meant for a single system. We extended a clustering middleware called Caft to incorporate efficient scheduling and profiling for multi-threaded Java applications. The middleware uses Terracotta for sharing runnable objects and shared data structures between different machines in the cluster.

The middleware supports a range of scheduling heuristics such as RoundRobin, ThreadLoad, CpuLoad, MemoryLoad, AverageLoad and Accelerated-AverageLoad with periodic and on demand updates about state information from the workers. It also supports a hybrid form of scheduling where workers themselves take local decisions for cases where threads spawn multiple threads. This hybrid scheduling is lazily consistent and aims at leveraging performance at the cost of sacrificing strict consistency. The performance of the scheduling heuristics vary for different types of application. Results indicate that it is possible to classify these scheduling heuristics based on the application properties while achieving linear speed-ups. A profiler allows to gather information about the application properties such as thread inter-arrival time, thread workload and resource usage of the application. With this information, the user is able to choose to the most efficient scheduling that suites the characteristics of the application.

6.1 Future Work

At present, the middleware does not support load-balancing. In order to facilitate load balancing, the middleware would need to support thread-migration. The middleware also assumes that none of the worker nodes or the master nodes fail during the period of execution, i.e, the current implementation is not fault tolerant. Any multi-threaded application that spawns threads by extending the Thread class does not schedule threads on workers using the middleware. This is because, the Thread class is non-portable and Terracotta does not allow to cluster any objects that extend the Thread class.

Bibliography

[1] http://www.hyperic.com/products/sigar.

[2] http://www.terracotta.org.

[3] Introduction to the spring framework.

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

[5] Gabriel Antoniu, Luc Boug, Philip Hatcher, Mark MacBeth, Keith Mcguigan, and Raymond Namyst. The hyperion system: Compiling multithreaded java bytecode for distributed execution, 2001.

[6] Yariv Aridor, Michael Factor, and Avi 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.

[7] Guy E. Blelloch, Phillip B. Gibbons, Girija J. Narlikar, and Yossi 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.

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

[9] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[10] Rajkumar Buyya, Toni Cortes, and Hai Jin. Single system image. *Int. J. High Perform. Comput. Appl.*, 15(2):124–135, 2001.

[11] T.L. Casavant and J.G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *Software Engineering, IEEE Transactions on*, 14(2):141 –154, feb 1988.

[12] K Cooper, A Dasgupta, K Kennedy, C Koelbel, A Mandal, G Marin, M Mazina, F Berman, H Casanova, A Chien, H Dail, X Liu, A Olugbile, O Sievert, H Xia, L Johnsson, B Liu, M Patel, D Reed, W Deng, and C Mendes. New Grid Scheduling and Rescheduling Methods in the GrADS Project. *International Journal of Parallel Programming*, 33:209–229, 2005.

[13] Fangpeng Dong and Selim G Akl. Scheduling Algorithms for Grid Computing : State of the Art and Open Problems. *Components*, pages 1–55, 2006.

[14] K. Etminani and M. Naghibzadeh. A min-min max-min selective algorihtm for grid task scheduling. In *Internet, 2007. ICI 2007. 3rd IEEE/IFIP International Conference in Central Asia on*, pages 1 –7, sept. 2007.

[15] Thomas 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.

[16] Pavel Fibich and Hana Rudov. Model of Grid Scheduling Problem. *Centrum*, 2001.

[17] E. Huedo, R.S. Montero, and I.M. Llorente. Experiences on adaptive grid scheduling of parameter sweep applications. In *Parallel, Distributed and Network-Based Processing, 2004. Proceedings. 12th Euromicro Conference on*, pages 28 – 33, Feb. 2004.

[18] H. Izakian, A. Abraham, and V. Snasel. Comparison of heuristics for scheduling independent tasks on heterogeneous distributed environments. In *Computational Sciences and Optimization, 2009. CSO 2009. International Joint Conference on*, volume 1, pages 8 –12, april 2009.

[19] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience*, 32(2):135–164, February 2002.

[20] Yun-Han Lee, Seiven Leu, and Ruay-Shiung Chang. Improving job scheduling algorithms in a grid environment. *Future Generation Computer Systems*, 27(8):991–998, October 2011.

[21] Joao Lemos. Distributed clustering and scheduling of vms, master thesis.

[22] M. Maheswaran, S. Ali, H.J. Siegal, D. Hensgen, and R.F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, pages 30 –44, 1999.

[23] Girija 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.

[24] Rob Van Nieuwpoort, Jason Maassen, Thilo Kielmann, and Henri E. Bal. Satin: Simple and efficient java-based grid programming. In *In AGridM 2003 Workshop on Adaptive Grid Middleware*, 2005.

[25] A. Othman, P. Dew, K. Djemame, and I. Gourlay. Adaptive grid resource brokering. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 172 –179, sept. 2003.

[26] R. Raman, M. Livny, and M. Solomon. Resource management through multilateral matchmaking. *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*, pages 290–291.

[27] Rajendra Sahu. Many-Objective Comparison of Twelve Grid Scheduling Heuristics. *International Journal*, 13(6):9–17, 2011.

[28] Christian Setzkorn and Ray C. Paton. Javaspaces - an affordable technology for the simple implementation of reusable parallel evolutionary algorithms. In Jesus A. López, Emilio Benfenati, and Werner Dubitzky, editors, *Proceedings of the International Symposium on Knowledge Exploration in Life Science Informatics, KELSI 2004*, volume 3303 of *Lecture Notes in Artificial Inteligence*, pages 151–160, Milan, Italy, 25-26 November 2004. Springer.

[29] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

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

[31] Douglas Thain, Todd Tannenbaum, and Miron Livny. *Condor and the Grid*, pages 299–335. John Wiley and Sons, Ltd, 2003.

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

[33] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic java application partitioning. pages 178–204. Springer-Verlag, 2002.

[34] R. Veldema, R.A.F. Bhoedjang, and H.E. 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.

[35] Fatos Xhafa and Ajith Abraham. Meta-heuristics for Grid Scheduling Problems. pages 1–37, 2008.

[36] Fatos Xhafa and Ajith Abraham. Computational models and heuristic methods for Grid scheduling problems. *Future Generation Computer Systems*, 26(4):608–621, April 2010.

[37] Matthias Zenger. Javaparty - transparent remote objects in java, 1997.