A MIDDLEWARE FOR SUPPORTING SPATIAL CONSTRAINTS IN DISTRIBUTED SYSTEMS

By

HUA ZHANG

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Engineering and Computer Science, Vancouver

MAY 2015

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of HUA ZHANG find it satisfactory and recommend that it be accepted.

Xinghui Zhao, Ph.D., Chair

Scott Wallace, Ph.D.

Sarah Mocas, Ph.D.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my advisor Dr. Xinghui Zhao for the continuous support of my study and research, for her patience, motivation, enthusiasm, and immense knowledge. Her guidance helped me in all the time of research and writing of this thesis.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Scott Wallace and Dr. Sarah Mocas, for their encouragement, insightful comments and brilliant questions.

I would also like to thank Dr. Wayne Cochran for his useful instructions on mobile development and iOS programming. A special thanks to Dr. David Chiu who gave me substantial and valuable advice for my study. With his guidance, I understand Hadoop ecosystem and job scheduling methodology. In addition, I am grateful to all CS professors and classmates. With their kind help and support, I have made my two years studying here a memorable experience.

Last but not the least, I would like to thank my parents, wife, son, sisters, relatives, and friends for supporting me, and being with me during good and hard times.

A MIDDLEWARE FOR SUPPORTING SPATIAL CONSTRAINTS IN DISTRIBUTED SYSTEMS

Abstract

by Hua Zhang, M.S.
Washington State University
May 2015

Chair: Xinghui Zhao

In a distributed system, the locations where computations take place are critical information which may affect many aspects of the system, e.g. data locality, load balancing, and fault tolerance. Traditionally, location control needs to be hard-coded in applications, and absolute locations must be specified when a computation is relocated. This is not necessary and limits the flexibility of the system. In this thesis, we propose a novel approach for supporting relative location constraints in distributed systems. We formulate the spatial coordination problem as a Constraint Satisfaction Problem (CSP) and solve it using an adapted backtracking algorithm. We have designed and implemented a middleware for AKKA platform (an implementation of the Actor model), namely Constraint Management Middleware (CMM), which enforces spatial constraints dynamically at runtime. Using our CMM, programmers can either specify primitive spatial constraints of the actors or develop more advanced spatial constraints by themselves. We illustrate the efficiency and usability of our CMM using two computations - WordCount and π Computation. We discuss how primitive spatial constraints can be used to develop more advanced spatial constraints, e.g., isolation and circulation. In addition, we have extended our CMM to Hadoop YARN sys-

tem. We take into account the location dependency between map and reduce tasks, and implement a new job scheduler based on CMM. Experiments have been carried out to evaluate our CMM based Hadoop scheduler using a traditional WordCount application and two Hadoop YARN benchmarks. Our results show that the CMM based scheduler outperforms the default Hadoop YARN schedulers for all computation-intensive applications.

TABLE OF CONTENTS

ACKNOWLEDGMENTS **iii**

ABSTRACT **iv**

TABLE OF CONTENTS **viii**

LIST OF TABLES **ix**

LIST OF FIGURES **1**

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Statement and Contributions 2

1.3 Organization 3

2 Related Work **4**

2.1 Distributed Systems and Spatial Constraints 4

2.2 Constraint Satisfaction Problems (CSPs) 5

2.3 Actor Model and AKKA 6

2.4 Load Balancing in Actor Based Systems 7

3 Spatial Coordination **9**

3.1 Spatial Constraints 9
3.1.1 Absolute Spatial Constraints 10
3.1.2 Relative Spatial Constraints 10
3.2 Problem Formulation 11

4 System Design and Implementation **14**
4.1 System Design 14
4.1.1 CMM Overview 15
4.1.2 System Architecture 16
4.1.3 Data Flow Diagram 17
4.1.4 Sequence Diagram 18
4.2 Implementation 19
4.2.1 Class Description 19
4.2.2 APIs 20
4.3 Programming Spatial Constraints 21
4.3.1 Circulation - Mailman Application 21
4.3.2 Actions in Group - Emergency Evacuation 21

5 Evaluation **24**
5.1 Environment 24
5.2 Effectiveness of CMM 24
5.3 Performance Analysis 26
5.3.1 Single Application 26
5.3.2 Multiple Applications 27
5.4 Scalability Analysis 28
5.4.1 Number of Constraints 28
5.5 Code Complexity Comparison 29

5.6 Demonstration 29

6 Case Study **32**

6.1 Apache Hadoop YARN 32

6.1.1 Hadoop YARN Architecture 32

6.1.2 Hadoop YARN Job Scheduling 34

6.1.3 Limitation of FIFO and Fair Schedulers 35

6.2 CMM Scheduler 36

6.2.1 Design 36

6.2.2 Implementation 37

6.2.3 Configuration 38

6.3 Evaluation 38

6.3.1 Experimental Setup 38

6.3.2 Dataset 39

6.4 Benchmarks 39

6.4.1 Terasort 39

6.4.2 RandomWrite and RandomRead 39

6.5 Results 41

6.6 Summary 43

7 Conclusion and Future Work **44**

Bibliography **47**

List of Tables

3.1 Relative Location Constraints for Actors 11

4.1 CMM APIs 20

5.1 File Size for WordCount Application 27

5.2 Constraints in Three Settings 27

6.1 Random Write and Read Configuration 40

List of Figures

4.1 Constraint Management Middleware 14
4.2 System Overview 15
4.3 CMM System Architecture 16
4.4 Data Flow Diagram for Registering a New Actor 17
4.5 Sequence Diagram 18
4.6 CMM Class Diagram 19

5.1 Compute π in Cluster Approach 25
5.2 Result: compute π in Cluster Approach 26
5.3 Performance Comparison- WordCount 27
5.4 Performance Comparison: Two WordCount Applications 28
5.5 Scalability of CMM 29
5.6 Code Complexity of CMM for WordCount 30
5.7 Demonstration: Reactive Map 31

6.1 Apache Hadoop YARN Architecture 33
6.2 Terasort Basic Data Flow 40
6.3 Performance Comparison of Four Schedulers 41
6.4 Terasort benchmark 42
6.5 Random Write/Read benchmark 42

Chapter 1

Introduction

Over the past decades, distributed systems have been the *de facto* solution for large-scale problems, in which the computations or the volume of data – or both – exceed the capacity of a single machine. However, programming distributed systems is challenging because of the communication, coordination, and synchronization that must be involved in the code. In general, using distributed systems to solve large-scale problems can be considered a trade-off between scalability and programming complexity. In other words, with added, sometimes significant programming complexity, the system may scale up to a much larger scale. In this thesis, we propose to address this challenge - to a certain extent - by separating the concerns of spatial / location coordination from those of computations in distributed programming.

1.1 Motivation

The growing ubiquity of big data applications results in renewed interests in the mobility of computations. This is because moving required data to computations is no longer a viable solution. Instead, computations themselves should be migrated to the location where the required data resides.

Traditionally, in distributed programming, the location of a computation must be explicitly specified in the code. However, the fact that an absolute location must be specified in a computation limits the flexibility of distributed systems. In most of the cases, specifying absolute locations in not necessary, while relative location constraints are more meaningful.

The Actor model [1], provides a high-level programming abstraction for supporting concurrent and distributed systems. Because actors do not share states, developers do not need to deal with explicit locking and thread management, making it easier to write correct concurrent and parallel systems. However, absolute locations must be specified in actor migration, which restricts the flexibility of the Actor model.

In this thesis, we address this challenge, by developing a Constraint Management Middleware (CMM) which maintains various types of spatial constraints, and enforces those constraints at runtime. Using CMM, the location-related code is separated from the function-related code, providing a clear design, better modularity and reusability. In addition, CMM can be easily extended for supporting user-defined spatial constraints.

1.2 Thesis Statement and Contributions

In this thesis, we have designed a Java middleware to maintain spatial constraints for computations, to ensure that multiple computations are separated across several nodes for load balancing, or to collocate the data read computation with calculate computation in a data intensive environment. Our middleware is extensible, and can support customized constraints programmed by developers. To solve these separation and collocation constraints, we model this spatial coordination problem as a Constraint Satisfy Problem(CSP), and use a customized backtracking algorithm to solve this problem at runtime. We have also observed that with carefully designed spatial constraints, our CMM consistently outperforms the original AKKA system for various types of applications in a distributed cluster. In addition, we

have integrated this approach into Hadoop's runtime system to support user-defined spatial control. Experimental results show that our CMM based scheduler can achieve better resource utilization and outperform Hadoop's FIFO and Fair Scheduler for various applications and benchmarks.

1.3 Organization

The rest of the thesis is organized as follows. Chapter 2 presents the related work. Chapter 3 formulates spatial coordination as a Constraint Satisfaction Problem, and discusses the customized backtracking algorithm. Chapter 4 describes the design and implementation of the Constraint Management Middleware (CMM). Chapter 5 evaluates CMM in terms of the efficiency and scalability. Chapter 6 presents a case study in which we integrate CMM into Hadoop's runtime system to provide spatial control. Chapter 7 concludes the thesis.

Chapter 2

Related Work

In this chapter, we review related works in three aspects of this research including spatial coordination, Constraint Satisfy Problem and the Actor Model.

2.1 Distributed Systems and Spatial Constraints

Distributed systems are often organized under the form of networks where nodes and edges are embedded in space. Understanding how the spatial constraints affect the structure and properties of networks is crucial for improving the performance of distributed systems.

In [2], Barthélemy et al. review the most important models of spatial networks and discuss various processes which take place on these spatial networks, such as disease spread, navigation, random walks and synchronization

Besides networks, spatial relationship among computations have also attracted more and more interests. Among the existing works in this area, FarGo [3], VCAE [4], and [5] are closely related to our work.

The FarGo system [3] provides a programming model that allows users to specify various collocation relationships between components. It provides a compiler and runtime support

for component mobility and enforcement of relative collocation invariants. This is similar to our relocation mechanism in that it allows all components to have their own policies, but it is limited to addressing collocation requirements, while our middleware supports multiple spatial constraints, as well as user-defined constraints.

VCAE [4] is an application consolidation engine, which provides constraint management and application placement as key mechanisms to minimize the number of machines on which applications are placed while satisfy certain constraints. Although VCAE addresses both collocation and separation, it mixes location and resource concerns. The key idea of our work is to separate the computation and spatial constraints in the runtime system.

Nadeem et al. proposes a mechanism which allows application programmers to maintain spatial relationships between computations in the runtime system [5]. Although the authors provide an approach to maintain spatial relationships, they do not discuss or support user-defined spatial relationships. In CMM, we support customized spatial constraints developed by programmers. We also support conditional spatial relationship.

2.2 Constraint Satisfaction Problems (CSPs)

Spatial coordination problems can be considered as a branch of the more general field of Constraint Satisfaction Problem [6]. To solve spatial coordination problems, we use the techniques that have been proven effective in solving CSPs [7].

Yokoo et al. give an overview of CSPs, adapt related search algorithms and consistency algorithms for applications to multi-agent systems, and consolidate recent research on cooperation in such systems [8].

Climent [9] presents a search algorithm that searches for both stable and robust solutions for CSPs. Meeting both criteria simultaneously is a desirable objective for solving constraints in dynamic environments.

Costa et al. solve human resource allocation problems in cooperative health services by using a Constraint Satisfaction Problem approach [10]. They have also studied the behavior of the backtracking algorithm when used to solve the CSP problem. Their research method is similar to ours. We adapt the backtracking algorithm to solve spatial constraints. New constraints can be inserted or removed whenever needed, making it possible to use this middleware to solve different problems.

2.3 Actor Model and AKKA

In distributed systems, one of the most pronounced challenges is to write correct concurrent, fault-tolerant and scalable applications. The Actor model [1], provides a high-level programming abstraction for supporting concurrent and distributed systems.

In an Actor system, actors are objects which encapsulate state and behavior, and they communicate by asynchronous message passing. These messages are placed in the recipients mailbox. Upon receiving a message, an actor can 1) generate more messages; 2) modify their behavior; 3) create new actors.

Based on the Actor model, AKKA [11] provides a toolkit to build highly concurrent, distributed, and scalable software. To achieve load balancing, concurrency and remoting in a distributed system, AKKA allows the developers to distribute their actors across multiple machines.

The Actor model and AKKA have been used in various applications in distributed systems. Mohindra et al. present a framework for composing and executing big data analytics in batch and interactive workflows across the enterprise [12]. Using the Actor Model and AKKA, the authors implement the framework which provides a unified local and distributed programming model to data centers.

To overcome the high-load image process challenges in ancient manuscripts and prints

digitization, Schöneberg et al. introduce a decentralized, event-driven workflow system [13]. It leverages dynamic routing between workflow components, and thus is able to adapt to the sources' unique requirements. This system also provides a scalable approach to soften out high computational loads on single nodes by using distributed computing.

Motesnitsalis et al. use AKKA to reduce the execution time of the falling object detection analysis by a factor of 10 on a local machine and the first attempts to execute the program on a cluster reduce the execution time by 96% [14].

MOSDEN is a collaborative mobile sensing framework that can operate on smartphones to capture and share sensed data between multiple distributed applications and users [15]. MOSDEN separates the application-specific processing from the sensing, storing and sharing. It is scalable and requires minimal development effort from the application developers. This design pattern is similar to that of our CMM.

Cherix et al. develop a programming framework based on SCALA actors that run on the Android operating system [16]. This framework, named ActorDroid, enables the execution and distribution of Java and Scala programs among a dynamic network of mobile devices. ActorDroid allows the programmers to focus on the computation and execute all those actors on a single device or multiple mobile devices. This idea is similar to our CMM, but ActorDroid do not consider the spatial constraints between the actors.

2.4 Load Balancing in Actor Based Systems

Work has been done to balance the workload for AKKA and other Actor based systems.

In AKKA 2, the dispatcher coordinates the message dispatching to the actors mapped on the underlying threads. They optimize the resources and process the messages as quickly as possible [17]. AKKA provides multiple dispatch policies that can be customized according to the underlying hardware resource (number of cores or memory available) and type of

application workload: 1) The BalancingDispatcher dispatches a message to an actor only when that actor would otherwise be idle, 2) The SmallestMailboxRouter dispatches the message to the actor with the least number of messages in its mailbox. Moreover, AKKA allows us to write our own dispatcher implementation.

Zhao et al. present a novel approach for dynamically load-balancing non-uniform parallel computations [18]. Their approach is to explicitly reason about resource requirements and commitments at run-time in order to make fine-grained scheduling decisions. For a diverse set of computations, their approach outperforms work-sharing approach and shows better scalability .

Chapter 3

Spatial Coordination

In distributed systems, the spatial control related code and computation related code are mixed. Developers have to handle the spatial coordination by their own code, which is a time consuming task. The key idea of our approach is to support spatial coordination in distributed systems by separating the specification and enforcement of location constraints from computations. Specifically, we formulate the problem as a Constraint Satisfaction Problem (CSP), and use the backtracking algorithm [19] to solve it. We integrate this approach into the AKKA platform, and provide application programming interfaces (APIs) for programming different spatial coordination strategies.

3.1 Spatial Constraints

In distributed systems, the developers must take into consideration both computations and their locations. For an AKKA cluster which includes one master node, multiple computation nodes and data nodes, the developers must make decisions on how to deploy their actors to the physical nodes.

3.1.1 Absolute Spatial Constraints

In Actor systems, the location related control is usually hard-coded in the programs. For example, in a basic actor operation `migrate`, the absolute location of a destination must be given. However, this kind of absolute constraint is not necessary. In many real world applications, relative location relationships are more meaningful.

3.1.2 Relative Spatial Constraints

We propose a middleware for supporting *relative* spatial constraints among actors, because the absolute paths and IP addresses of actors are unnecessary information which increases the program complexity. Here we propose two primitive spatial constraints: `separation` and `collocation` in actor systems. For instance, if two actors have no relationship, we can separate them into different nodes or physical locations. If two actors are related, e.g. two actors which need to exchange messages, we can collocate them for better performance and locality.

With these two primitive location constraints, we also support two advanced location constraints: `isolation` and `circulation`. Isolation can be implemented by separating the target actor from all other actors in the system. Circulation can be implemented by the combination of two primitive constraints (separation and collocation). To circulate one actor A_{target} within actors list { A_1, A_2, A_3, ..., A_n}. The basic scenario, in chronological order, let A_{target} first collocate with actor A_1, and then separate with A_1; then, collocate with A_2, and then separate with A_2, and so on. In this way, actor A_{target} visits all the other actors in the actors list. This scenario is quite useful in the wireless network. With this circulation pattern, a mobile data collection device can visit all the sensors and monitors which are deployed areas that are dangerous or difficult to receive data from.

Table 3.1 shows a list of relative location constraints for actors.

Constraints	Parameters	Description
collocation	A_1, A_2	A_1 A_2 are collocated
separation	A_1, A_2	A_1 A_2 separated
isolation	A_1, Actorlist	A_1 is isolated from all other actors.
circulation	A_1, Actorlist	A_1 visit each actor in ActorList

Table 3.1: Relative Location Constraints for Actors

3.2 Problem Formulation

We formulate the spatial coordination as a Constraint Satisfy Problem (CSP), which consists of three components, A, N, and C [19], defined as follows.

A is a set of actors , { A_1, ..., A_n}

N is a set of nodes, {N_1, ..., N_n}, which actors are placed

C is a set of constraints, {C_1, ..., C_n}, which specify allowable continuations of value.

Each constraint C_i consists of a pair $\langle$scope, rel$\rangle$, where scope is a tuple of variables that participate in the constraint, and rel is a relation that defines the values for those variables.

Each constraint C_i

(1) has set of variables, called its scope: e.g., a,b,c.

(2) is a Boolean function that maps assignments to these variables to true/false.

e.g. C_1 {A_1=a,A_2=b,A_4=c} = True

this set of assignments satisfies the constraint.

e.g. C_2 {A_1=b,A_2=c,A_4=c} = False

this set of assignments falsifies the constraint.

Therefore, a set of constraints C can be presented as

C = { C_1, C_2} = { $\langle${A_1=a,A_2=b,A_4=c}, True$\rangle$, $\langle${A_1=b,A_2=c,A_4=c) , False$\rangle$}

Constraint satisfaction problems on finite domains can typically be solved using a search algorithm. The most widely used algorithms are variants of backtracking, constraint propagation, and local search.

The backtracking algorithm is an important tool for solving constraint satisfaction problems. It enumerates a set of partial candidates that could be completed in various ways to give all the

possible solutions to the given problem. It maintains partial assignments for the variables until a solution is found. Initially, all variables are unassigned. At each step, a variable is chosen, and all its possible values are tested one at a time against the pre-defined constraints. If a valid value is found, a recursive call is performed to assign values for the rest variables. If all possible values have been tested and no solution is found, the algorithm backtracks [19].

For our spatial coordination model, we use a customized backtracking-search algorithm to solve the constraints, as shown in Algorithm 1. Note that heuristic approaches can be used in the algorithm to choose potential locations for an actor. In addition, the complexity of the algorithm can be reduced by deriving inferences from certain types of constraints, e.g., collocation, as suggested in [20].

Algorithm 1 shows the customized backtracking algorithm. Unlike the standard backtracking algorithm, this adapted algorithm will stop after finding the first solution. CMM uses two HashMaps to maintain the status of actors and nodes and one ArrayList to store relative constraints.

ALGORITHM 1: `CSP_Solver` (constraints, yellowpage, map)

```
if all actors have been deployed then
    return map;
end
else
    get the next actor a;
end
for each node in possible_locations(a, map, constraints) do
    if place a to node is consistent with map then
        place actor a to node;
        derive the inferences of the deployment;
        if inferences do not lead to failure then
            add inferences to map;
            result = CSP_Solver(constraints, yellowpage, map);
            if result != failure then
                return result;
            end
        end
    end
    remove deployment of a and its inferences from map
end
return failure;
```

Chapter 4

System Design and Implementation

4.1 System Design

Using the spatial coordination model and the adapted backtracking algorithm, we have developed a middleware on AKKA for supporting relative location constraints among actors.

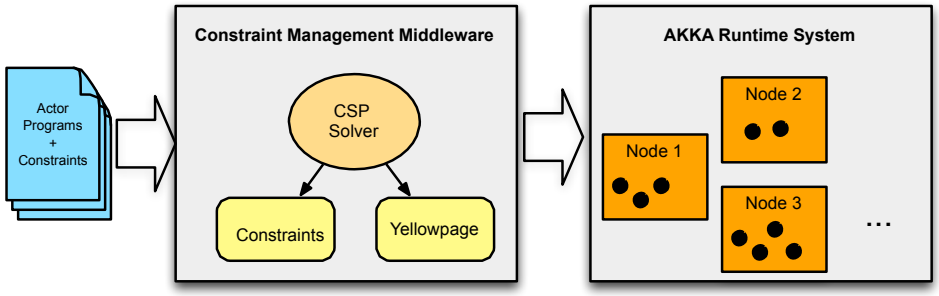

Figure 4.1: Constraint Management Middleware

As shown in Figure 4.1, CMM accepts actor programs and user-defined constraints, which can be specified in either a static or a dynamic way.

A static constraint specification is in the form of a separate constraint configuration file, which lists actor names and their relative location constraints. This form is suitable for specifying pre-

dictable, fixed constraints. For instance, a computation actor should be collocated with the data that it requires.

In contrast, a dynamic constraint specification can be combined in the actor program so that constraints can be dynamically generated at run time. This form is particularly useful when the location constraints evolve during the course of the computation, i.e., location constraints need to be programmed. For example, a data aggregation actor should initially be placed at a location close to the actors from which it collects data; however, after the data has been collected, it can be free to move elsewhere for other purposes.

4.1.1 CMM Overview

There are two main components in our CMM show in Figure 4.2. The first component is a Meta Actor module which supervises and monitors the child Actors. The second one is the CSP Solver module which uses the modified Backtracking algorithm to find an assignment for the child actors.

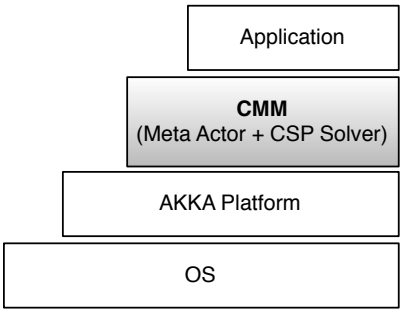

Figure 4.2: System Overview

In Meta Actor module, we create a actor system and a Meta Actor. The Meta Actor is a master actor which supervises and monitors the computation (child) actors. Meta Actor maintains a HashMap to keep track the status of all the child actors. The child actors are created by the Meta Actor. The CSP Solver module is responsible for finding a solution to satisfy the location constraints among the actors. For remote actors assignment in an actor system, our CSP Solver module can assign locations to the actors according to the current location constraints automatically.

4.1.2 System Architecture

Figure 4.3 shows the system architecture of the CMM. An application created by developers sends a request, such as separate, collocate, or isolate actors, to the Meta Actor. The Meta Actor receives this request and parses the constraints in the configuration file or dynamic input by developers, and invokes the CSP Solver to solve the constraints problem, and find a solution to deploy/ re-allocate actors in the AKKA runtime system. Finally, the application is informed to continue its computation with the actors.

Static constraints are usually set up when the actor system is initialized. The developers provide the necessary basic constraints in the configuration file. Dynamic constraints occur when the actor system is running and add/remove/migrate single or several actors.

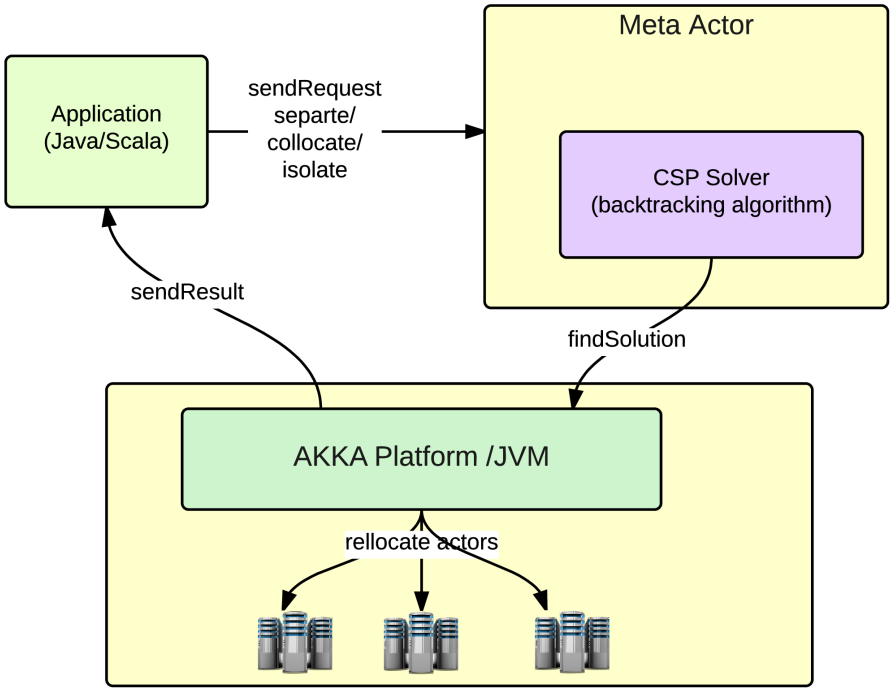

Figure 4.3: CMM System Architecture

4.1.3 Data Flow Diagram

Figure 4.4 shows the data flow diagram of CMM for creating a new actor in the existing actor system.

- Application sends location register request to Meta Actor, when a new actor is created
- Meta Actor uses CSP Solver(backtracking) to check for a conflict with existing constraints
- After CSP Solver solves all constraints, Meta Actor gets the solution for this actor and other existing actors
- All actors will be deployed to the correct nodes

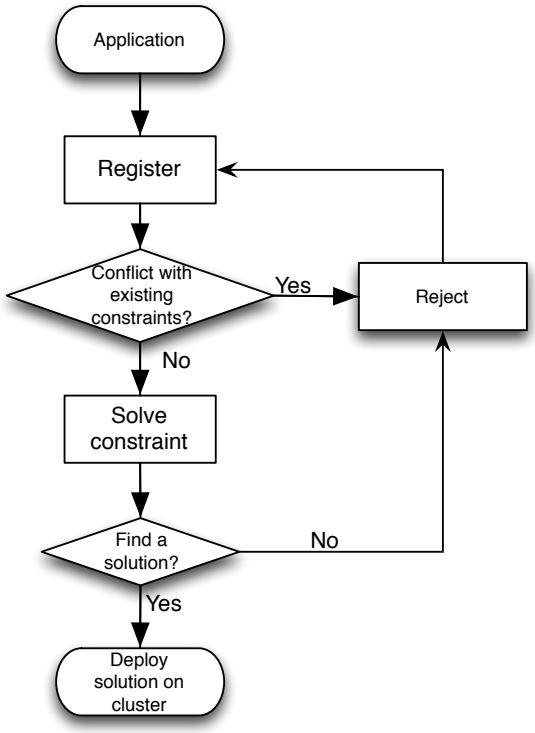

Figure 4.4: Data Flow Diagram for Registering a New Actor

4.1.4 Sequence Diagram

Figure 4.5 shows a sequence diagram for processing and solving spatial constraints for actors. In this sequence diagram, the Meta Actor serves as a mediator. It receives a request from the application, invokes the CSP Solver's methods, and manipulates the actors. If the CSP Solver can solve the problem and find a valid solution, the Meta Actor can enforce the deployment based on the solution in the end.

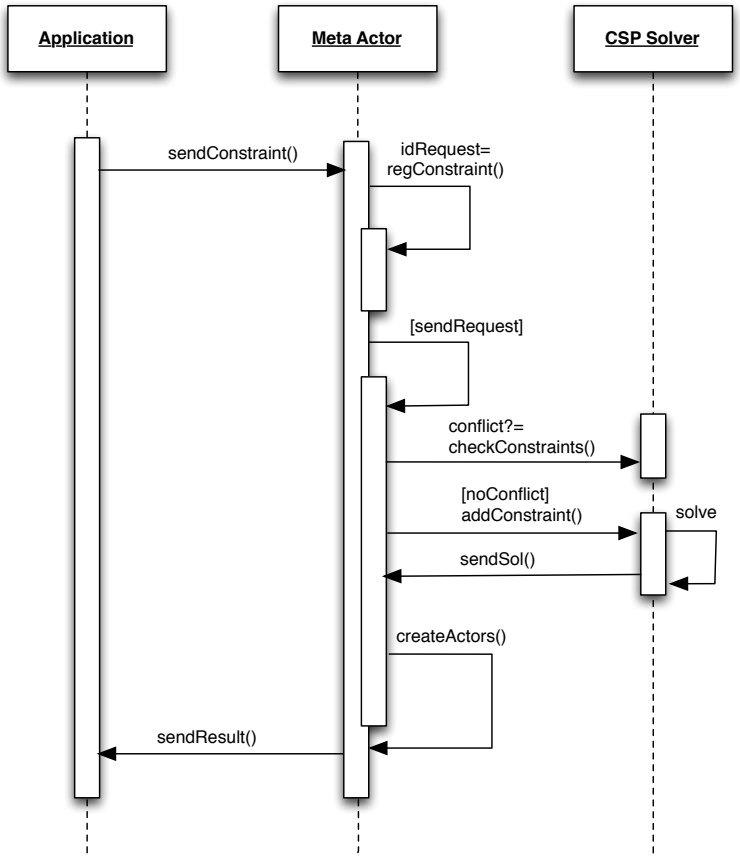

Figure 4.5: Sequence Diagram

4.2 Implementation

Using the design above, we implement four key classes in CMM: Class MetaActor, Class CSP which has children class - CSPListener monitor the state of CSP, Class Assignment and Class BacktrackingStrategy. Figure 4.6 is CMM class diagram.

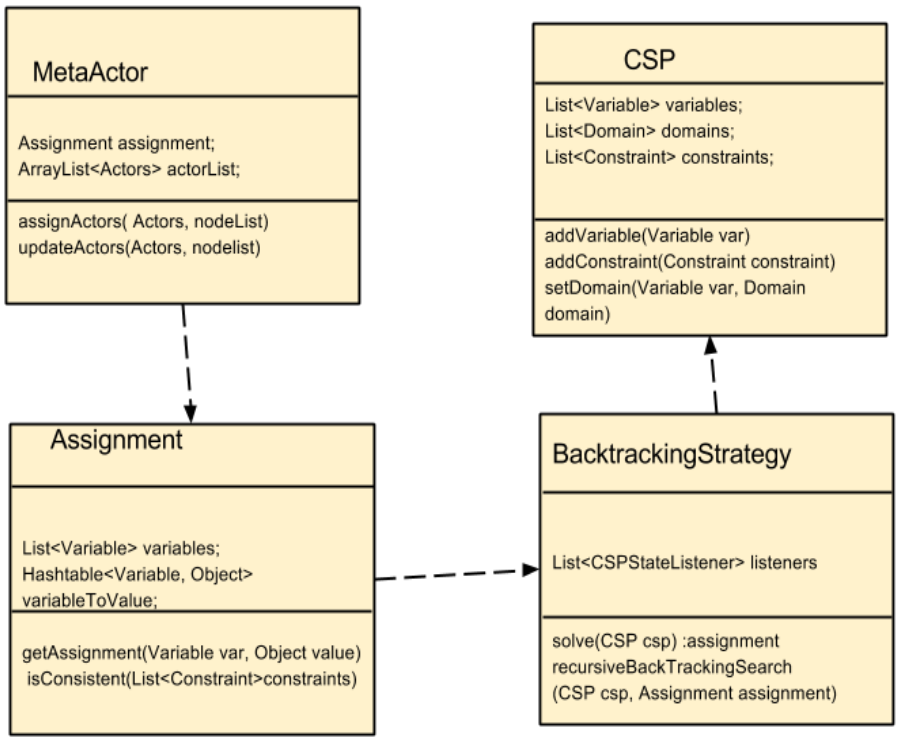

Figure 4.6: CMM Class Diagram

The basic relationship between these four classes: (1) when a new instance of MetaActor is created, it initially creates an instance of Class Assignment. (2) To get the assignment, we use the BacktrakcingStrategy to solve the CSP. (3) Once the assignment is ready, MetaActor will update the actors accordingly.

4.2.1 Class Description

Class MetaActor: create an actor system, maintain an actorList and assignment

Class CSP: a base class for solving CSP

Class BacktrackingStrategy: using the backtracking algorithm to find a solution

Class Assignment: assigns values (locations) to some or all actors.

4.2.2 APIs

For the developers, we provide the three methods to complete the basic location operations: (collaborate, separate, and isolate actors). Table 4.1 is a summary of the APIs in our CMM. In this

Table 4.1: CMM APIs

Type	**Method and parameters**	**Description**
void	Collocate(Actor A1, Actor A2)	Collocate actor A1 and A2
void	Separate(Actor A1, Actor A2)	Separate the actor A1 and A2
void	Isolate(Actor A1)	Isolate actor A1 from all other actors.

section, we demonstrate how to use CMM to implement an actor program. Here are general steps to use CMM:

(1) the developers import our CMM jar library file into their own project.

(2) to enforce location constraints in their code, they must create a Meta Actor with CSP Solver, and provide input constraints in static or dynamic style.

(3) after the developers create the Meta Actor, the Meta Actor will use the assignment which was created by CSP Solver, deploy actors to different nodes, monitor actors in an actor system and assign the location for actors dynamically.

e.g., when one actor is created, the Meta Actor is able to specify the location for this new actor according to current constraints. When an actor is removed, the current constraints HashMap will also be updated. Therefore, our CMM is able to satisfy the location constraints among the existing actors and new actors in runtime.

4.3 Programming Spatial Constraints

Our CMM also supports the more advanced spatial constraint requirements, such as circulate, or isolate constraints. The programmers might use our CMM to customize their own spatial constraints and get the optimal solution. E.g. we use two application patterns - circulation and action in group- to elaborate how to customize the complex constraints.

4.3.1 Circulation - Mailman Application

In Mailman application, our CMM handles the circulation scenario in which mailman visits all the other actors in the group. To circulate mailman actor A_{target} within actors list { A_1, A_2, A_3, ..., A_n}, the basic scenario, in time order, let A_{target} first collocate with actor A_1, send or pickup the mail, and then de_collocate with A_1; then, collocate with A_2, and then de_collocate with A_2, and so on.

We provide the pseudocode for this circulation scenario. Algorithm 2 is our pseudo-code for mailman application. Note that the primitive constraints we described in section 3.1.2 are used here to implement this application. For `collocate` (traget, actor A_i), we use the CSP Solver to find a solution by the adapted backtracking algorithm. In this case, the backtracking search is not complex, we can satisfy this constraint by move A_{target} to the location of A_{target} .

4.3.2 Actions in Group - Emergency Evacuation

For the advanced spatial constraints, developer can implement their own spatial constraints by using our primitive APIs in CMM.

In this case, we assume that there are 2000 employees in a headquarter. When a fire alarm is trigged, we need notify people and guide them evacuate from the dangerous area. As this headquarter only has four exits, how do we let them go outside in a couple of minutes? For this scenario, how do we deal with this problem? We need to use an advanced constraint pattern. In the beginning, there are a few factors to be considered.

ALGORITHM 2: `Circulation_Application` (target, ActorList)

```
if target actor visited all actors in ActorList then
    return true;
end
else
    for each actor A_i in ActorList do
        collocate( target, actor A_i) ;
        do_computation(target, actor A_i) ;
        if computation is done then
            de_collocate(target, actor A_i);
        end
    end
end
return failure;
```

- Distance between people and the exit
- Number of people that can exit safely through a path /exit.

  If the number of people moving to one exit exceeds its limit, it may result in serious problems.
- Group

  It is not easy to make sure individual get the text message alert or email alert. If we can notify the group rather than individuals, it is more effective.

  It is better to group the people who are close to each other into one group. If the number of people in one group exceeds an upper limit, there is an option to split it into two groups.
- Combine/Spilt the groups.

  If both of two groups are small, and the sum is less than the upper limit of one group, we combine two or more groups into one bigger group. If the group size is larger than specified restriction, we split the large group to two or three groups. When more people are added into group, we can move the people via a group. and design the reasonable route for each group.

We can program this spatial constraint described above using APIs provided by CMM. Algorithm 3 is the pseudocode for programming spatial constraints.

ALGORITHM 3: `Customize_Constraint` (constraints, yellowpage, map)

```
if all actors have been deployed then
    return map;
end
else
    get the next actor a;
end
for each node in possible_group(a, map, constraints) do
    if place a to node is consistent with map then
        if Current group is available then
            place actor a to node in group;
            derive the inferences of the deployment;
            if inferences do not lead to failure then
                add inferences to map;
                result = Customize_Constraint(constraints, yellowpage, map);
                if result != failure then
                    return result;
                end
            end
        end
    end
    remove deployment of a and its inferences from map
end
return failure;
```

Chapter 5

Evaluation

Experiments have been carried out to evaluate the effectiveness and scalability of our approach. These experiments are MapReduce WordCount and approximate π computation. We also use a Reactive map to visualize the spatial coordination on the web browser.

5.1 Environment

We setup the AKKA cluster in our lab. We have one master node and two computation nodes. All of the nodes are Mac with Intel i5 2.7G processor, 750GB Santa disk. We use Java 1.7 and AKKA 2.3.9 in our experiments. The performance is monitored by a web-based tool- AppDynamics agent.

5.2 Effectiveness of CMM

Calculating the value of π is a computationally intensive application, i.e, CPU bound. To demonstrate the effectiveness of our CMM in the context of distributed computing, we implement an algorithm to estimate the value of π by using AKKA Remote Actors and our CMM to scale out on multiple machines in a cluster.

The algorithm we used is based on the computation of the following Gregory-Leibniz series:

$$\sum_{n=1}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots = \frac{\pi}{4}.$$

For this algorithm, we use a master actor to divide the series into equal parts for each present worker actor. When each worker actor has processed its chunk, it sends a result back to the master. The master aggregates the total result.

Figure 5.1 shows how to compute π in a cluster approach.

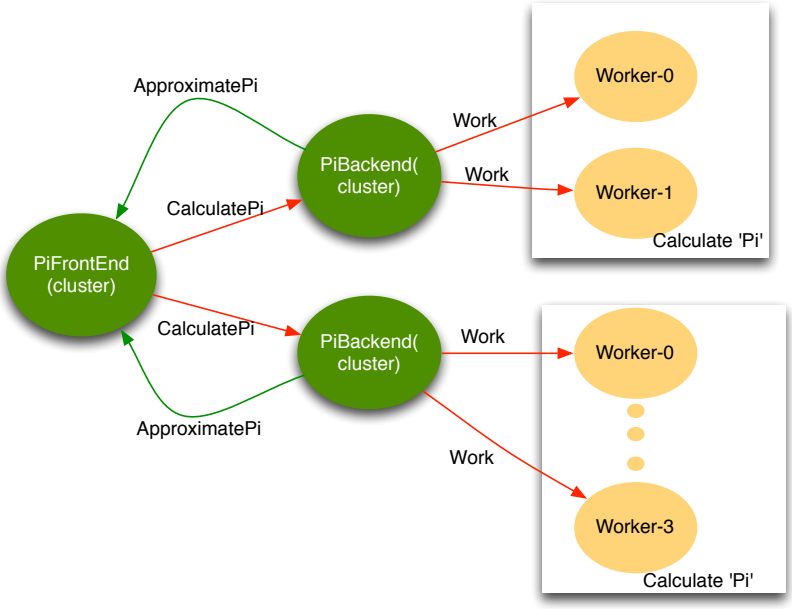

Figure 5.1: Compute π in Cluster Approach

This computation requires far less communication between the nodes and shows the validity of the chosen approach. Using CMM, the program will separate worker actor to multiple machines in the cluster.

Figure 5.2 illustrates the effectiveness of our CMM. That is, using CMM to compute π in a cluster approach can improve the performance by reduce the computation time. When the number of actors is increased, the computation time will decrease accordingly.

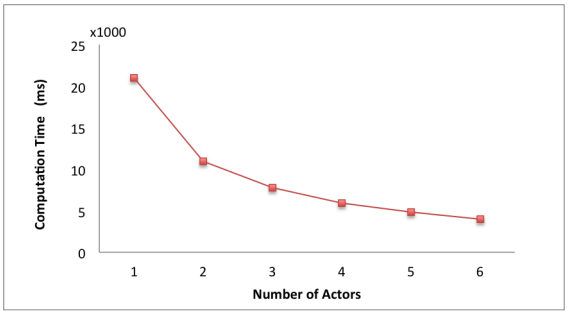

Figure 5.2: Result: compute π in Cluster Approach

5.3 Performance Analysis

We analyze the performance of our CMM using a WordCount application on the distributed system.The implementation of this WordCount application is based on the MapReduce algorithm.

5.3.1 Single Application

In these experiments, we use AKKA actor system and CMM to implement a MapReduce application, WordCount. The application counts the number of occurrences of each word in a file. We implement the application as follows. A dataActor is responsible for maintaining the files, and send the files to other actors which require them. The actual computation is carried out by a number of mapActors and reduceActors in the same way as Map-Reduce.

We first run the computation in the original AKKA platform with 3 physical nodes. In this case actors are randomly distributed on 3 nodes. We then run the same computation on CMM, with one static spatial constraint, which is *collocate(mapActor, dataActor)*.

In this way, the computations are collocated with the data that they require. Figure 5.3 shows the results of the experiments with different file sizes (See Table 5.1). When the file size increases, we can achieve an order of magnitude of performance gain using CMM, with no added programming complexity. Only a static constraint is needed.

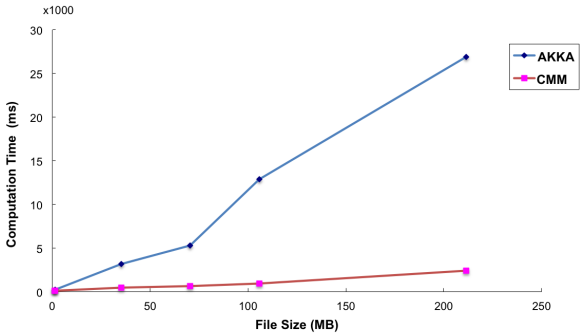

Figure 5.3: Performance Comparison- WordCount

File	book1	book2	book3	book4	book5
Data(MB)	1.5	35.2	70.5	105.7	211.4

Table 5.1: File Size for WordCount Application

5.3.2 Multiple Applications

To verify our CMM in parallel environment, we run two WordCount applications in parallel on AKKA, CMM and CMM with load balancing. Table 5.2 shows the setting of constraints. Let WordCount1 read book i, WordCount2 read book j, and then measure the maximum computation time. Using the constraints in Table 5.2. In the third case, we add the load balance control by separating the mapActors in WordCount1 from the mapActors in WordCount2.

Setting	**Constraints**
AKKA	dispatch dataActor, mapActor and reduceActor randomly
CMM	`collocate` mapActor and dataActor for each WordCount
CMM + Load balancing	besides collocation, `separate` two mapActors in two applications

Table 5.2: Constraints in Three Settings

Overall, for two WordCount application run in parallel, using CMM and load balance method, the developer can get the better performance in their project. Figure 5.4 Shows the performance is better in with load balance case.

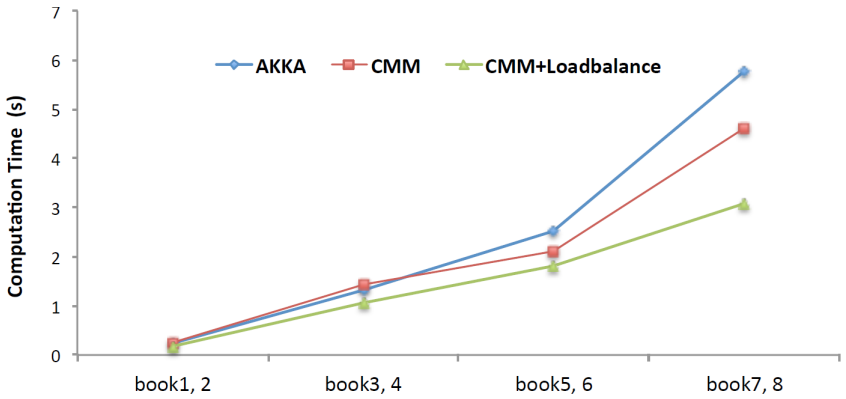

Figure 5.4: Performance Comparison: Two WordCount Applications

5.4 Scalability Analysis

In the experiments presented in section 5.3, only a small number of constraints are used. To evaluate the scalability of CMM, we have carried out another set of experiments.

5.4.1 Number of Constraints

In a standard CSP, the solving time will be exponential when the number of nodes is decreased and the number of actors is increased. For CMM, it only handles two kinds of constraints: separation and collocation. The complexity of collocation is lower than separation in the practical case.

In these experiments, we randomly generate a number of constraints and uses CMM to generate and enforce a deployment plan. We then measure CSP solving time of CMM, and the results are shown in Figure 5.5, which demonstrates that CMM scales well when the number of constraints increases. This scalability is impacted by the number of separations and the number of collocations.

Therefore, We can get the conclusion that our CMM works well up to a reasonable number of actors and quotas in nodes.

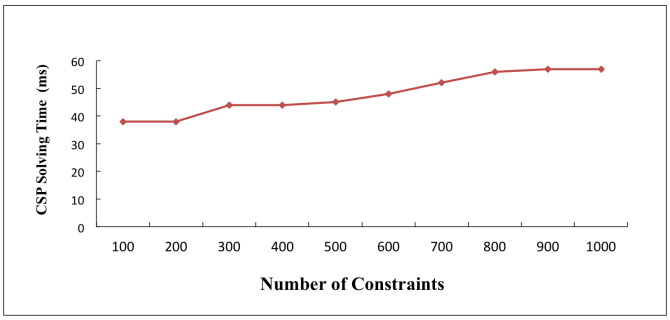

Figure 5.5: Scalability of CMM

5.5 Code Complexity Comparison

Hard-coding long-lived spatial relationships of multiple computations using low-level mobility primitives increases programming complexity. For example, if the developer want to implement the separate, collocate and migrate the actors by their own code, they need to write about two hundred lines for each migration or manipulation.

Our CMM provides support for actor mobility and enables the dynamic adaption of the system configuration during runtime with minimal (zero-programming) effort. We demonstrate this code complexity by using CMM in WordCount application. Figure 5.6 shows the code to use our CMM. If the programmers use CMM in their own project, they do not need to deal with the migration manually. Using the APIs of CMM, they can implement the feature and get the assignment with merely 40 lines.

5.6 Demonstration

To visualize the spatial coordination provided by our CMM, we modify a reactive map application by using Akka and CMM to provide horizontally scalable and resilient message passing and data management.

This application is an example of how to implement the tenets of the Reactive Manifesto using the Typesafe Reactive Platform [21]. It uses Play, combined with the latest in client side technologies to implement a reactive user interface. The application shows the location of every

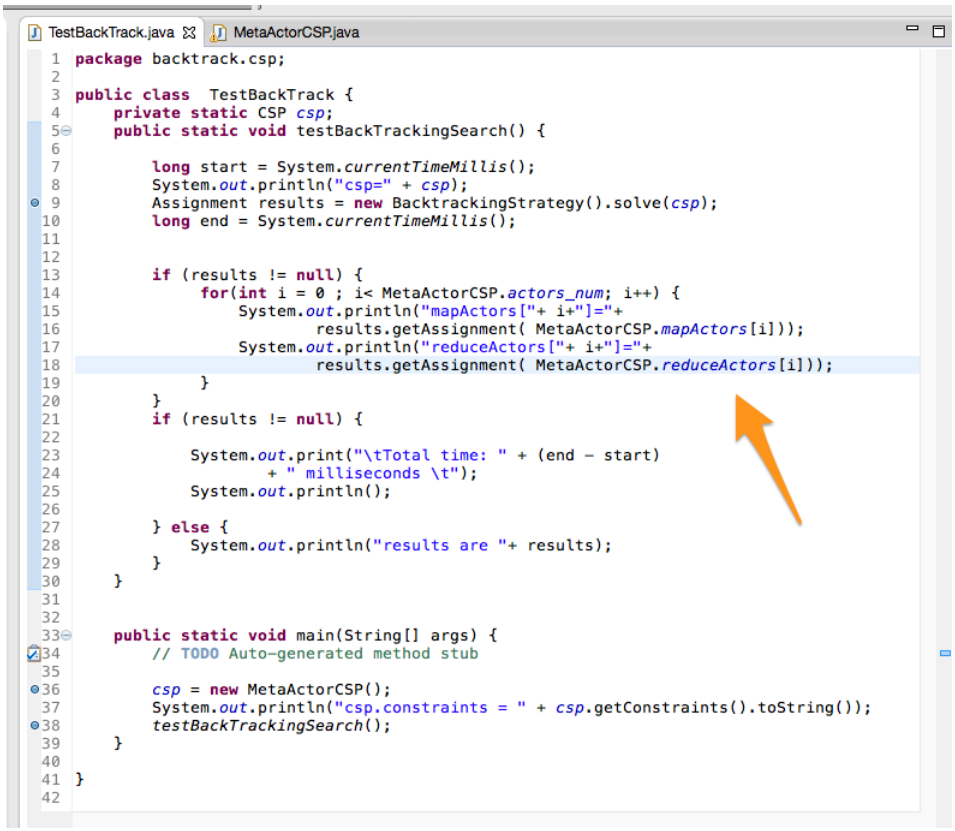

Figure 5.6: Code Complexity of CMM for WordCount

user currently connected on a map, updated in real time [22].

In Figure 5.7, it visualizes the movements of actors. For demonstration, we can specify the input parameters- location constraints and group information in Web-UI and separate or collocate the actors in movement.

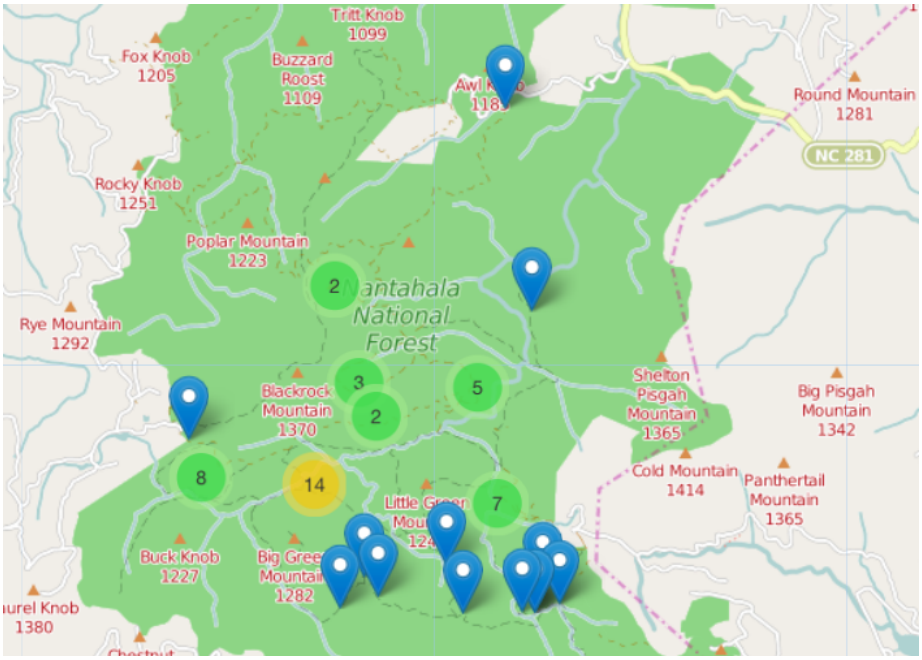

Figure 5.7: Demonstration: Reactive Map

Chapter 6

Case Study

In this case study, we develop a job scheduler plug-in for Hadoop YARN cluster based on our CMM. This CMM scheduler takes into account both location constraints and the dependency between map and reduce tasks and provide an effective scheduling strategy. We evaluate the performance of our CMM scheduler using a big data application and two benchmarks, namely Terasort and Random Write/Read.

6.1 Apache Hadoop YARN

In this section, we introduce Apache Hadoop YARN architecture and its job scheduling.

6.1.1 Hadoop YARN Architecture

Figure 6.1 shows the architecture of Apache Hadoop YARN. A Hadoop YARN system consists of multiple worker nodes and one master node. In the master node, a centralized ResourceManager (RM) routine manages the resources. On each worker node, a NodeManager (NM) routine monitors the system status on that node [23].

An ApplicationMaster (App Mstr) then generates resource requests, and negotiates resources from the scheduler of ResourceManager. On each worker node, the ApplicationMaster works with

the NodeManager to execute and monitor the map and reduce tasks for each job.

The ResourceManager, NodeManagers and ApplicationMasters form the data processing framework. The ResourceManager is the key component that allocates resources to the applications in the system [24]. It consists of three components: Scheduler, ApplicationsManager and Resource Tracker, as shown in Figure 6.1. The functionablities of each component is described as follows,

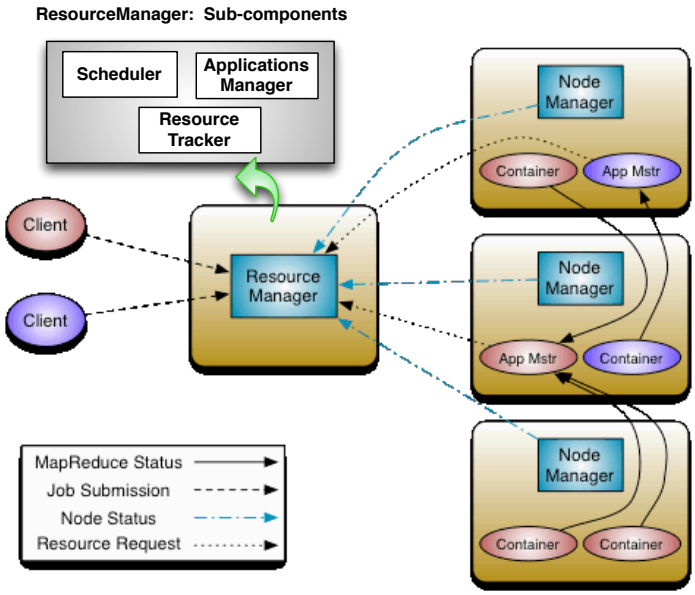

Figure 6.1: Apache Hadoop YARN Architecture

- Scheduler

  Allocates resources to different application with the constraints(capacities and hierarchical queues).

- ApplicationsManager

  Accepts job submissions, negotiate resources with the ResourceManager, work with the NodeManager to execute the tasks for the user applications.

- Resource Tracker

  Contains setting such as the maximum number of ApplicationMaster retries, etc.

The Scheduler is responsible for partitioning the cluster resources among various queues, applications etc. It can employ different resource allocation policies as plug-ins. Various MapReduce schedulers such as the Capacity-Scheduler and the Fair Scheduler are examples of the plug-in.

Furthermore, Hadoop YARN manages the system resources in a fine-grain. Specifically, each NodeManager needs to report the available memory and CPU cores of its worker node to the ResourceManager/Scheduler, and each ApplicationMaster needs to specify the resource demands for its tasks. The scheduler in Hadoop YARN will then allocate available resources to the waiting tasks based on a particular scheduling policy.

6.1.2 Hadoop YARN Job Scheduling

In a Hadoop YARN cluster, job scheduling is performed by the scheduler in the master node, which distributes work to a number of worker nodes. Tasks are assigned by the master node in response to heartbeats received from the worker nodes every few seconds.

Scheduling of MapReduce jobs in Hadoop proceeds as follows:

- At first, worker nodes notify the master about the availability of free resources.
- Second, the master will assign a pending task to a worker node with free resources.

Scheduler is a pluggable scheduling component of YARN. Depending on the use case and user needs, administrators may select either a simple FIFO, capacity, or fair scheduler. The various scheduler options are described briefly in this section.

Hadoop YARN has three typical Schedulers:

- `FIFO scheduler`

  in which the scheduler maintains an FIFO waiting queue of jobs sorted by arrival time. When resource becomes available, the master simply selects the next job in the waiting queue for execution. Typically, FIFO schedules have no sense of job priority. The FIFO scheduler is practical for small workloads, but can cause issues when large shared clusters are used [25].

- `Fair Scheduler`

  which ensures that all submitted MapReduce jobs get an equal opportunity for executing their tasks. Under the Fair scheduler, when a single application is running, that application may request the entire cluster. If additional applications are submitted, resources that are free are assigned fairly to the new applications so that each application gets roughly the same amount of resources [25].

- `Capacity Scheduler`

  which puts jobs into multiple queues in accordance with the conditions, and allocates certain system capacity for each queue. The Capacity scheduler permits sharing a cluster while giving each user or group certain minimum capacity guarantees. The Capacity scheduler works best when the workloads are well known which helps in assigning the minimum capacity [25].

6.1.3 Limitation of FIFO and Fair Schedulers

In Hadoop YARN, the reduce tasks of a MapReduce job consist of two main stages, shuffle and reduce. In the shuffle stage, the output of each map task of the job is transferred to the worker nodes which host the reduce tasks, while computation in the reduce stage starts when all the input data are ready. Therefore, the execution time of a reduce task are dependent on several map-related factors, such as the execution times of all map tasks and the size of the intermediate output data.

Under FIFO and Fair schedulers, when multiple jobs are running concurrently in the cluster, their reduce tasks are launched and thus occupy most of the resources, which may dramatically delay the execution of map phases [26].

As FIFO and Fair Scheduler do not take into account the dependency between map and reduce tasks, our work attempts to cover these aspects and puts more emphasis on meeting location constraints in the shared cluster environment.

Suppose there are two jobs: `Job i` and `Job j` run on a Hadoop YARN cluster. When `Job i`'s map tasks run first and reduce tasks come along, these reduce tasks have impact on the remain map tasks in `Job i`, or `Job j`'s map tasks. With our CMM scheduler, if all the resources are used

by current map and reduce tasks, the new reduce task will be held by our scheduler until one of map tasks is completed and Nodemanager notify the available resource to our job scheduler.

Using CMM which is described in the previous chapter, we implement our scheduler to solve this problem by considering the impacts of dependency between map and reduce task, and thus achieves the better completion time (makespan) of a set of MapReduce jobs.

6.2 CMM Scheduler

In this section, we describe the design, implementation and configuration of our CMM Scheduler for Hadoop YARN cluster. It is developed as a plug-in module. Note the configuration file of Hadoop needs to be modified to use this scheduler.

6.2.1 Design

The CMM scheduler, as a conditional constraint scheduler, is designed based on the following ideas.

- Resource awareness

  The CMM scheduler supports CPU and memory resources. It takes into account the memory and CPU requirements of submitted applications and support their dynamic needs.

- Locality

  The CMM scheduler supports specifying the locality of computation as well as node by applications. It is good at automatically allocating resources on not only preferred nodes, but also nodes that are close to the preferred ones (in a lesser distance). With this idea, it ensures that the developers does not have to worry about locality.

- Load balancing

  The third interesting idea is the dynamic nature of the map and reduce tasks. These are executed as YARN Containers, and their numbers will change as the application runs. This feature provides much better cluster utilization. To improve utilization, CMM scheduler

also enforces strict location constraints to avoid two applications from overwhelming a single node. For this case, we separate two applications into two nodes with available resources respectively, to satisfy the location constraints, improve the performance and achieve load balancing.

Base on these ideas, the design goals of our scheduler are twofold. First, the scheduler should be able to give users immediate feedback on whether the job can be completed with the given location constraints. If constraints can be met, proceed with the execution. Second, the scheduler should maximize the number of jobs that can be run in the cluster while satisfying the location requirements of all jobs.

6.2.2 Implementation

We take into account the dependency between map and reduce task, exploit the CMM strategy to derive the better task assignment in the runtime. That is, our CMM scheduler considers the relationship between tasks for determining the appropriate task to execute.

Our CMM scheduler has two main components: Init_TaskAssignment and Dynamic_TaskAssignment. When the cluster is started, all ApplicationMasters have submitted the resource requests for their MapReduce tasks to CMM scheduler. Init_TaskAssignment is to assign the first batch of tasks for execution while the rest of tasks remain pending in the system queue.

The Dynamic_TaskAssignment is the key component in our CMM scheduler. It selects a set of tasks for being served on a worker node which has newly released resources.

The scenario is as follows,

(1) When old tasks are finished and the corresponding resources are released, NodeManager will notify CMM scheduler through heartbeat messages.

(2) Then CMM scheduler will execute Dynamic_TaskAssignment to select one or more task from the pending queue.

(3) And assign them to the worker node where the available resources are located.

6.2.3 Configuration

We compile the source code of our CMM scheduler into a single Java jar file. To enable our CMM scheduler in Hadoop YARN, we place its JAR file on Hadoop's classpath, by copying it to the lib directory. Then set the mapred.jobtracker.taskScheduler property to:
org.apache.hadoop.mapred.CMMScheduler

The CMM scheduler will work without further configuration.

6.3 Evaluation

We then evaluate the scheduler using one classical MapReduce application -WordCount [27] and two Hadoop YARN benchmark applications. We investigate both the correctness (whether the constrains are satisfied) and the efficiency (whether a solution can be generated in a timely manner) [28].

For each applications, we have not considered map and reduce task start-up overhead in our experiments. We also do not consider the failure rate of the cluster in the experiments. We evaluate the task assignment behavior of Constraint Scheduler for map and reduce location constraints.

6.3.1 Experimental Setup

We set up a Hadoop YARN cluster in our DSR lab. Our experiments are carried out in this cluster. Each node of Hadoop YARN cluster is an iMac PC with 2.7GHz quad-core Intel Core i5 processor 8GB 1600Mhz DDR3. Total volume of disk size is 450GB.

Except changing the algorithm of the scheduler, all other configuration parameters were default values. The HDFS block size was 64 MB. The measured network capacity were
- download Speed: 84815 kbps (10601.9 KB/sec transfer rate)
- upload Speed: 105947 kbps (13243.4 KB/sec transfer rate).

6.3.2 Dataset

We use a real dataset for our WordCount on this Hadoop YARN cluster. This dataset consists of reviews from Amazon. The data span a period of 18 years, including 35 million reviews up to March 2013. Reviews include product and user information, ratings, and a plaintext review [29].

6.4 Benchmarks

Hadoop YARN provide several benchmarks to validate its performance. We utilize two benchmarks to evaluate our CMM scheduler, compared with FIFO and Fair schedulers.

6.4.1 Terasort

A full TeraSort benchmark run consists of the following three steps:

- Generating the input data via `TeraGen`.
- Running the actual `TeraSort` on the input data.
- Validating the sorted output data via `TeraValidate`.

Figure 6.2 shows the basic data flow.

6.4.2 RandomWrite and RandomRead

RandomWrite/RandomRead benchmark writes/reads 10 GB (by default) of random data/host to DFS using Map/Reduce [30]. Each map takes a single file name as input and writes random BytesWritable keys and values to the DFS sequence file. The maps do not emit any output and the reduce phase is not used. The specifics of the generated data are configurable.

Table 6.1 shows the configuration variables :

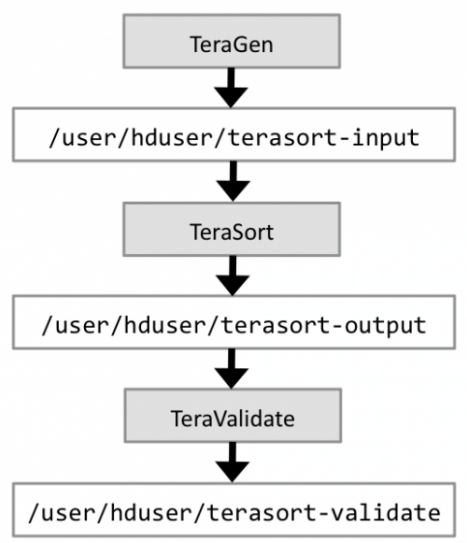

Figure 6.2: Terasort Basic Data Flow

Name	Default Value	Description
maps_per_host	10	Number of maps/host
bytes_per_map	1073741824	Number of bytes written/map
min_key	10	minimum size of the key in bytes
max_key	1000	maximum size of the key in bytes
min_value	0	minimum size of the value
max_value	20000	maximum size of the value

Table 6.1: Random Write and Read Configuration

6.5 Results

To verify the correctness and efficiency of our CMM scheduler, we compare it with other job schedulers in Hadoop YARN. Specifically, we run two WordCount jobs: i, j concurrently using each scheduler. In CMM scheduler, the constraints are separating map tasks in job i from map tasks in job j when resources in worker nodes are available for two jobs. If resource utilization is high, the tasks are held in the queue, until resources are released by completed tasks. Figure 6.3 shows the WordCount application completion time among four schedulers. Our CMM scheduler outperforms both FIFO and Fair Schedulers. Its performance is slightly better than the capacity scheduler.

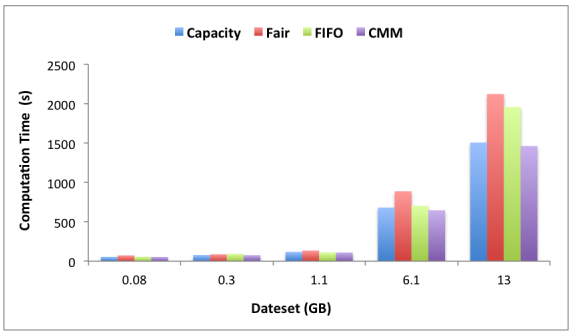

Figure 6.3: Performance Comparison of Four Schedulers

Using two Terasort jobs: i and j , CMM scheduler separates map tasks in job i from map tasks in job j. Figure 6.4 shows Terasort benchmark completion time comparison for four schedulers. Our CMM scheduler outperforms FIFO, Fair and Capacity schedulers.

Using two RandomWrite/Read jobs: i and j, CMM scheduler separates map tasks in job i from map tasks in job j. Figure 6.5 shows RandomWrite/Read benchmark completion time comparison in four schedulers. For I/O bound application, our CMM scheduler outperforms FIFO and Capacity Scheduler.

(The data in these three figures are collected from our cluster with Hadoop YARN 2.6.0.)

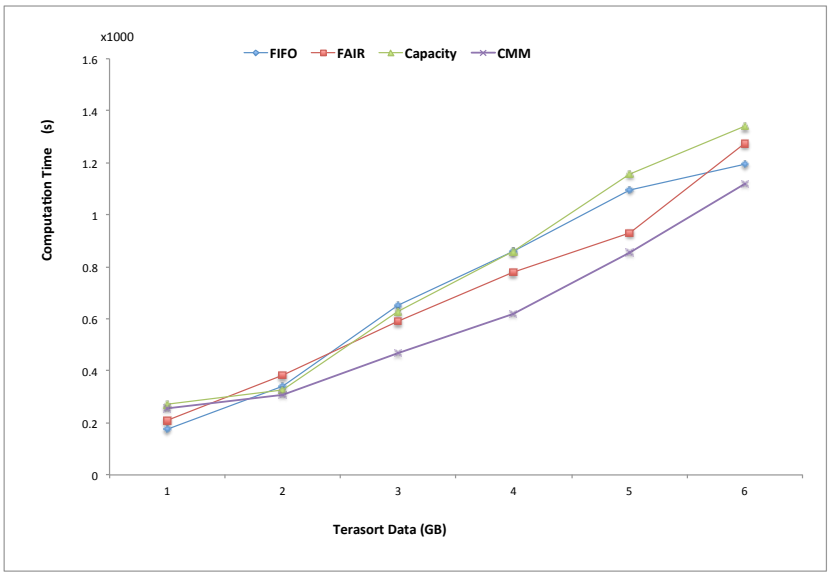

Figure 6.4: Terasort benchmark

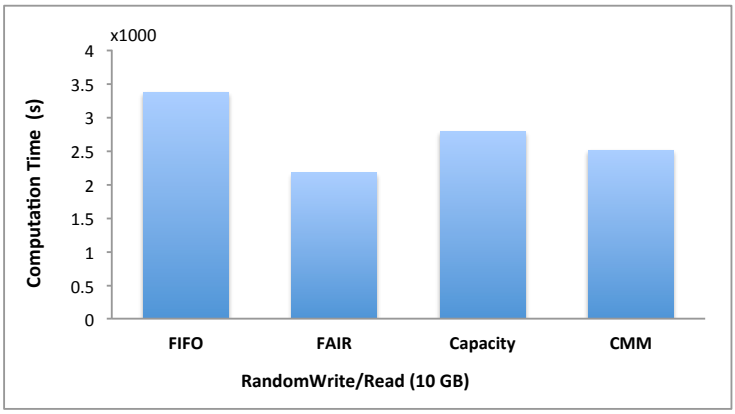

Figure 6.5: Random Write/Read benchmark

6.6 Summary

In this case study, we address the constraints between map and reduce task in Hadoop YARN cluster and design a CMM job scheduler for Hadoop YARN. Using WordCount application and two benchmarks to evaluate the efficiency and resource utilization of CMM scheduler.

The results show CMM scheduler can efficiently schedule map and reduce tasks and thus improve the performance. In addition, the performance of MapReduce jobs can be further improved by taking into consideration the dependency between map and reduce tasks when multiple jobs are competing for resources in Hadoop YARN cluster.

In our future work, we will consider the aspects that may affect on the performance of scheduling, including data distributions in a heterogeneous MapReduce system, iterative MapReduce jobs under time constraints, the system resources such as CPU and memory, network bandwidth, and map and reduce task execution time estimation.

Chapter 7

Conclusion and Future Work

The Actor model supports mobility of computations; however, its implementations often require users to explicitly specify the destination of actor migration, which unnecessarily limits the flexibility actor systems.

In this thesis, we present a Constraint Management Middleware (CMM), which enables users to program relative location relationships among actors, and enforces these location constraints at runtime. CMM on AKKA actor systems shows good scalability, as well as potentials of enhancing programmability and performance of actor systems. In a case study, we integrate CMM into Hadoop YARN system to schedule map and reduce tasks by enforcing spatial constraints dynamically according to the resource availability. Experimental results show that the CMM based scheduler outperforms Hadoop's default schedulers, FIFO and Fair scheduler.

Work is ongoing to extend CMM by accommodating other types of constraints, such as real-time constraints. In addition, we will use the extended CMM to solve problems in mobile clouds and other environments. Our CMM has been developed by adapting open-source technologies, and we plan to release our work as open source as well.

Bibliography

[1] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems.* Cambridge, MA, USA: MIT Press, 1986.

[2] M. Barthélemy, "Spatial networks," *Physics Reports*, vol. 499, no. 1, pp. 1–101, 2011.

[3] O. Holder, I. Ben-Shaul, and H. Gazit, "Dynamic layout of distributed applications in fargo," in *Proceedings of the 21st international conference on Software engineering.* ACM, 1999, pp. 163–173.

[4] H. Chen, H. Kang, G. Jiang, K. Yoshihira, and A. Saxena, "Vcae: A virtualization and consolidation analysis engine for large scale data centers," in *Self-Adaptive and Self-Organizing Systems (SASO), 2010 4th IEEE International Conference on.* IEEE, 2010, pp. 1–10.

[5] N. Jamali and A. Keela, "Maintaining spatial relationships in uncertain environments," in *Self-Adaptive and Self-Organizing Systems (SASO), 2012 IEEE Sixth International Conference on.* IEEE, 2012, pp. 227–228.

[6] C. Demetrescu, I. Finocchi, and A. Ribichini, "Reactive imperative programming with dataflow constraints," in *ACM SIGPLAN Notices*, vol. 46, no. 10. ACM, 2011, pp. 407–426.

[7] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of constraint programming.* Elsevier, 2006.

[8] M. Yokoo, *Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-agent Systems*, 1st ed. Springer Publishing Company, Incorporated, 2012.

[9] L. Climent, R. J. Wallace, M. A. Salido, and F. Barber, "An algorithm for finding robust and stable solutions for constraint satisfaction problems with discrete and ordered domains," in *Tools with Artificial Intelligence (ICTAI), 2012 IEEE 24th International Conference on*, vol. 1. IEEE, 2012, pp. 874–879.

[10] C. F. F. Costa Filho, D. A. R. Rocha, M. G. F. Costa, and W. C. de Albuquerque Pereira, "Using constraint satisfaction problem approach to solve human resource allocation problems in cooperative health services," *Expert Systems with Applications*, vol. 39, no. 1, pp. 385–394, 2012.

[11] Typesafe. Akka: homepage. [Online]. Available: http://akka.io/

[12] S. Mohindra, D. Hook, A. Prout, A.-H. Sanh, A. Tran, and C. Yee, "Big data analysis using distributed actors framework."

[13] H. Schöneberg, H.-G. Schmidt, and W. Höhn, "A scalable, distributed and dynamic workflow system for digitization processes," in *Proceedings of the 13th ACM/IEEE-CS Joint Conference on Digital Libraries*, ser. JCDL '13. New York, NY, USA: ACM, 2013, pp. 359–362. [Online]. Available: http://doi.acm.org/10.1145/2467696.2467729

[14] E. Motesnitsalis, "Using akka platform in unidentified falling object detection on the lhc." Tech. Rep., 2013.

[15] P. P. Jayaraman, C. Perera, D. Georgakopoulos, and A. Zaslavsky, "Efficient opportunistic sensing using mobile collaborative platform mosden," in *Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom), 2013 9th International Conference Conference on*. IEEE, 2013, pp. 77–86.

[16] P.-A. M. R. Cherix, "A distributed computing framework for mobile devices based on scala actors." 2012.

[17] K. G. Munish, *Akka Essentials*. Birmingham, UK: Packt Publishing, 2012.

[18] X. Zhao and N. Jamali, "Load balancing non-uniform parallel computations," in *ACM SIGPLAN Notices: Proceedings of the 3rd International ACM SIGPLAN Workshop on Programming Based on Actors, Agents and Decentralized Control (AGERE! at SPLASH 2013)*, Indianapolis, IN, 2013, pp. 1–12.

[19] S. Russell, "Artificial intelligence: A modern approach author: Stuart russell, peter norvig, publisher: Prentice hall pa," 2009.

[20] N. Jamali and A. Keela, "Maintaining spatial relationships in uncertain environments," in *Proceedings of the 2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems*, ser. SASO '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 227–228. [Online]. Available: http://dx.doi.org/10.1109/SASO.2012.39

[21] J. Allen, *Effective Akka*. O'Reilly Media, Inc., 2013.

[22] TypeSafe. Akka: Reative maps. [Online]. Available: http://akka.io/

[23] V. Henning and J. Reichelt, "Mendeley-a last. fm for research?" in *eScience, 2008. eScience'08. IEEE Fourth International Conference on*. IEEE, 2008, pp. 327–328.

[24] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[25] A. C. Murthy, V. K. Vavilapalli, D. Eadline, J. Niemiec, and J. Markham, *Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2*. Pearson Education, 2013.

[26] A. Verma, L. Cherkasova, and R. H. Campbell, "Two sides of a coin: Optimizing the schedule of mapreduce jobs to minimize their makespan and improve cluster performance," in *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '12.

Washington, DC, USA: IEEE Computer Society, 2012, pp. 11–18. [Online]. Available: http://dx.doi.org/10.1109/MASCOTS.2012.12

[27] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[28] T. White, *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[29] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: understanding rating dimensions with review text," in *Proceedings of the 7th ACM conference on Recommender systems*. ACM, 2013, pp. 165–172.

[30] M. Kontagora and H. Gonzalez-Velez, "Benchmarking a mapreduce environment on a full virtualisation platform," in *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*. IEEE, 2010, pp. 433–438.