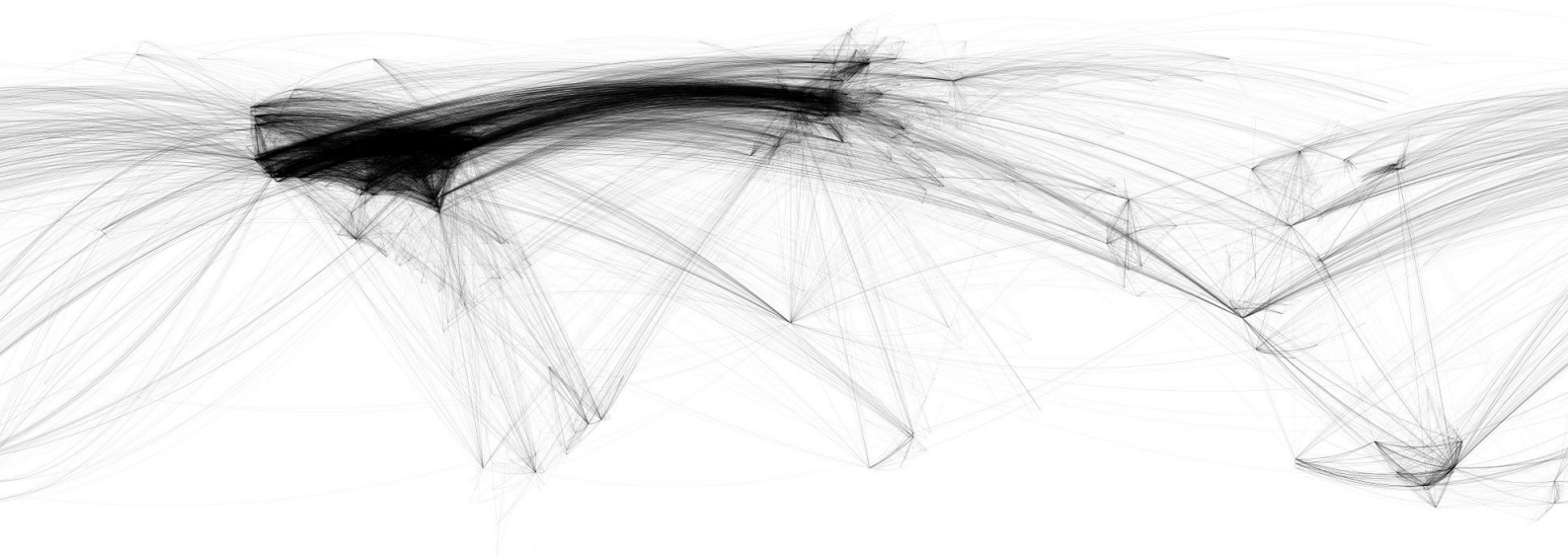
Optimal Deployment of Distributed Systems



Martijn Moraal

Master's Thesis Computer Science
Radboud University Nijmegen
Thesis No. 619, November 2009

Supervisors
prof.dr. F.W. Vaandrager (Radboud University)
dr. L.D. Michel (University of Connecticut)
dr. E. Marchiori (reader, Radboud University)

Radboud University Nijmegen

University of Connecticut

Abstract

As our world becomes more interconnected, there is an increasingly important role for distributed computer system. Designing these systems is not an easy task. Progress has been made in this regard with the development of formal specification languages and verification tools. One area that is usually not addressed is the deployment of a system. This is unfortunate as the deployment can be critical to the performance. Placing components on slow, unreliable hosts will severely hinder the system, while grouping components on the fastest hosts creates single points of failure.

This thesis investigates this deployment problem. The problem itself is a combinatorial optimization problem; a type of problem that is challenging computationally, with many instances being NP-hard. Many different techniques exist to solve these problems, however, what works best for a specific problem is difficult to predict. Mixed integer programming and constraint programming have been considered for the deployment problem in the past. This thesis extends on this work by investigating the use of constraint-based local search and hybrid methods for solving the problem.

The deployment problem of two distributed system architectures in particular is considered, and constraint-based local search and hybrid methods are developed to solve them. The developed methods perform very well. They strike different trade-offs between time that is spend searching for a deployment, and guarantees that are made on the quality of the obtained solution. The new methods are not superior to the existing methods in every way. Instead, they provide users more freedom in choosing the tools that are best for a specific task.

Preface

This master's thesis is the culmination and conclusion of my work as a student of computer science at the Radboud University Nijmegen. The research contained herein was performed at the department of Computer Science & Engineering of the University of Connecticut, CT, USA. Being able to perform this research in such an unfamiliar environment was an experience that was both intellectually stimulating and culturally enriching. An opportunity for which I am very grateful.

This thesis would not have been possible without the support of some people, and I would like to thank them. First of all I want to thank my advisor at the Radboud University, Frits Vaandrager, for giving me the opportunity to perform this research and for his support, ideas and feedback. Secondly I would like to thank my external advisor, Laurent Michel, for welcoming me at the University of Connecticut and allowing me to perform my Master's research there, and for his enthusiastic ideas, support, and help whenever I got stuck. Last but not least I want to thank Elaine Sonderegger, for her ideas, the insightful comments and her support whenever the work got frustrating or felt hopeless.

Martijn Moraal
Nijmegen, November 2009

Contents

1 Introduction

As our world becomes more and more interconnected, there is an increasingly important role for distributed computer systems. With the internet as the most prominent example it is not difficult to see the dominant role these systems play in our current society. That role is continuing to grow as new systems are continuously introduced, as wireless sensor networks develop, or as cloud computing emerges.

Distributed computing systems are appealing for a number of reasons. In the case of the internet it allows us to efficiently communicate and share information with each other, while wireless sensor networks enable the use of many autonomous devices to cooperatively monitor physical or environmental conditions. In general distributed networks enable the use of many, spatially separated, resources to work together to achieve a goal or perform a task, that each of the individual participants could not do on their own. For instance, a distributed system can allow multiple devices to co-operate by sharing cpu resources or storage space; it can enable more flexible access to, and more efficient sharing of data, by centrally storing it; or it can provide robustness and fault tolerance by eliminating single points of failure.

To achieve these tasks some underlying services and algorithms are needed. For fault tolerant data storage and sharing, for instance, some replication of the data is usually required. Storing the data on a single device would create a single point of failure and would place a great resource strain on that particular device. Replication of the data, however, raises the issue of maintaining atomicity, e.g. the service will need to guarantee that the same data is stored at each replication point. Designing systems that effectively and efficiently deal with these issues is not an easy task. Systems often consist of heterogeneous computing devices and are characterized by concurrent and asynchronous operations. These complexities make them hard to reason about. This is further complicated when the underlying logical network structure is dynamic, with hosts continuously joining and leaving, often without warning.

Some of these difficulties can be tamed using formal specification frameworks. Frameworks such as the Timed I/O Automata framework [17] and the associated Tempo Language and Toolkit [19] are specifically aimed to aid in the design of distributed systems and prove their correctness. Creating a correct specification is, however, only part of the problem. When a distributed system has been specified, and after rigorous analysis has been deemed correct, it will still need to be implemented and deployed. Implementing the system consists of generating the software components so that their behavior corresponds to that of the model. The deployment then deals with mapping the software components onto a distributed computing platform.

This deployment typically has a large impact on the performance of the system. For instance, non-uniform communication costs between the various devices in the network give significance to the placement of the different components, as placing a critical resource on a device with a slow communication speed to the rest of the network will significantly impact the overall performance of the system. Co-locating components on the same host allows for fast and cheap communication, but

it also hinders reliability and robustness by creating single points of failure. Distributing the software components among the hosts in the network allows more efficient use of processing and storage resources and improves fault tolerance. However, it also induces communication delays between the various components. The performance of the system can be optimized by placing the components in such a way that the communication delays between the components are minimal. Achieving such an optimal deployment, however, is a very challenging task. It is this task of creating optimal deployments that this thesis focuses on.

1.1 Optimal Deployment

The problem of deploying a specified distributed system onto a network structure typically consists of mapping the components of the system onto the hosts of the network. This mapping is subject to certain constraints. For example, certain components will need to be separated to achieve fault tolerance, while others will need to be placed together to function properly. Furthermore, not all hosts might be able to support all components, or bandwidth constraints on the links between the hosts might limit the amount of network traffic that can take place between them. Other properties might need to be optimized. For instance, the goal of a certain deployment might be to minimize the communication delays, or the total amount of network traffic.

Determining such an optimal deployment is a *combinatorial optimization* problem. Combinatorial optimization problems are concerned with the efficient allocation of limited resources to meet desired objectives. They are discrete problems with a set of discrete resources to allocate and a discrete set of solutions. Constraints on the resources reduce the total set of possible solutions, however, for most problems there is still a great number of feasible solutions. An overall objective determines which of these feasible solutions is the best.

Combinatorial optimization problems are generally extremely challenging computationally. Typically they are NP-hard, and thus cannot be solved exactly in polynomial time (unless $P = NP$). From a theoretical perspective it is difficult to get much traction on these problems. These problems are however a reality and are in fact ubiquitous in our society. Companies continuously face the problem of how to assign their limited resources, such as machines, vehicles and personnel, to perform certain activities. Typical examples are the assignment of airline crews to flights so that the operating costs are minimized [11], or the placement of warehouses so that transportation costs are minimized [33].

Many approaches have been developed to tackle these kinds of problems, and many tools have been designed to aid in the process. However, it is often difficult to predict what will and what will not work. It is unlikely that a single approach will be effective on all problems, or even on all instances of a single problem. Some problems are better solved using mathematical programming, some are more amenable to solutions by constraint programming, while local search is more effective on others. This difficulty in predicting which methods will yield the best results makes solving optimization problems a highly experimental endeavor.

1.2 Problem Definition

Not all distributed systems have the same deployment problem. The specifics of *what* needs to be deployed and *how* it needs to be deployed will vary. However, some generalities can be established. Typically there will be a set of software components with associated communication frequencies, and a set of hosts with associated communication delays. The assignment of components to hosts is restricted by constraints on sets of components or hosts which must or may not be assigned in certain ways. The objective of the deployment is to minimize some objective function which is defined in terms of certain properties of the components and the hosts.

In this thesis the deployment problem of two distributed system architectures in particular will be considered. These problems are interesting themselves and worth solving. However, they are also representative of more general sets of problems. The first system, Eventually-Serializable Data Services, provides a scalable fault-tolerant distributed data service. It operates in a static network setting, and its deployment problem is an *offline* problem that only needs to be solved once. The second system, Reconfigurable Atomic Memory for Basic Objects, provides shared atomic memory in a dynamic network setting. In contrast to the first system its deployment problem is an *online* problem that needs to be solved continuously while the services is running. Both types of systems and their associated deployment problems will be described briefly below.

This thesis will investigate the use of combinatorial optimization techniques to solve these problems. In particular, techniques from the field of *constraint programming* will be used. Constraint programming is very much a field in development and as such is constantly looking for new applications to test and extend its repertoire of techniques. The results of this thesis provide relevant feedback to the constraint programming community in this regard.

1.2.1 Eventually-Serializable Data Services

Eventually-Serializable Data Services (ESDS) [12] is a distributed algorithm for providing scalable fault-tolerant distributed data services. Providing distributed and concurrent access to data objects is one of the fundamental concerns of distributed systems. The simplest implementation of such a system uses a single centralized data object that can be accessed from multiple locations by multiple clients. There are however two large problems with this simple approach: it has a single point of failure, and due to congestion it does not scale well as the number of clients increases.

To solve these issues most distributed data services use some form of replication. The data object is replicated at multiple locations, each of which can be accessed independently. This approach eliminates the single point of failure and scales very well, since the number of replicas can be increased and the load can be balanced among them. It however also introduces a new problem: the problem of maintaining consistency of the replicated data object. The issue of consistency can be solved in many different ways, but it typically comes at a high performance cost. ESDS reduces this performance cost by trading off the immediate consistency guarantees, while ensuring the long-term consistency of the data.

An ESDS system consists of three different types of software components, *clients*, *front-ends*, and *replicas*. The deployment problem consists of assigning each component to a particular host in a network. This assignment is subject to certain constraints. For instance, no two replicas can be assigned to the same host to ensure fault tolerance. The goal of a deployment is to minimize the total communication delays. The deployment problem is an *offline* problem; it only needs to be solved once, before the system is actually deployed. The terms in which the ESDS deployment problem is stated are very general, and the deployment of many other distributed systems can be stated in similar terms.

1.2.2 Reconfigurable Atomic Memory for Basic Objects

The ESDS systems requires a static network to function properly. It allows for the failure of some hosts, but it does not allow for continuous and large changes in the underlying network. Providing consistent shared objects in such dynamic networks is another fundamental problem in distributed computing. Reconfigurable Atomic Memory for Basic Objects (RAMBO) [13, 20] was developed for this purpose. It makes guarantees about the consistency and availability of shared objects in a setting where hosts can continuously join, leave, or fail. RAMBO achieves this by replicating the shared object among hosts grouped in *read-* and *write-quorums*. The quorums enable the system to maintain memory consistency in the presence of small and transient changes.

The number of changes in the network that the quorum system is able to handle is, however, still limited. As hosts continuously leave and new ones join, there may at some point be none of the original hosts left. To be able to deal with these larger and more permanent changes RAMBO supports dynamic *reconfiguration*. This reconfiguration modifies the assignment of replicated objects to host and the associated quorum assignments. What exactly the new configuration should look like is a problem that RAMBO does not solve. It is this deployment problem that is considered in this thesis.

In the ESDS case a deployment can be computed before the system is actually deployed, the problem is therefore inherently an *offline* problem. In the RAMBO case, however, the deployment will need to be continuously adjusted to allow the system to keep functioning in the dynamic network setting it is designed for. The deployment problem is therefore very much an *online* problem, which means there is a tighter limit on the time available to compute the deployment. The RAMBO deployment problem deals with finding an assignment of quorum members, who maintain a copy of the data object, to hosts. This assignment is subject to the constraint that no two members can be placed on the same host. The goal is twofold, firstly the fault tolerance of the system has to be ensured so the most reliable hosts need to be chosen, secondly the performance needs to be optimized so that the communication delays are minimized.

1.3 Approach

This thesis builds upon research that has been previously done in this area. Previous work on the problems has considered the use of mathematical programming and constraint programming as solving techniques. The results have, however, left something to be desired. The problems proved extremely challenging computationally and solutions could often not be obtained within reasonable time limits. A brief overview of the related work done on these problems will be given below. This thesis investigates the use of constraint-based local search to tackle the problems. Constraint-based local search has the ability to often deliver good solutions very quickly. This performance, however, comes at the cost of completeness. In addition to constraint-based local search this thesis will investigate hybrid methods which combine multiple techniques.

Mastering techniques for combinatorial optimization is quite challenging. Fortunately, many tools have been designed to aid in the process and automate many of the low-level details, leaving only the high-level solving algorithm to be specified. This thesis uses the combinatorial optimization tool COMET [1, 15, 23].

The results presented herein are novel algorithms for solving the ESDS and RAMBO deployment problems. The developed constraint-based local search and hybrid algorithms offer sophisticated integration of several techniques. They perform very well, and are able to deliver solutions for realistic instances of these problems, in reasonable time. The developed methods strike different trade-offs between time that is spend searching for a deployment, and guarantees that are made on the quality of the obtained solution. The new methods are not superior to the existing methods in every way. Instead, they provide users more freedom in choosing the tools that are best for a specific task.

Results of this research have been published in two conference papers. The results that are achieved on the ESDS deployment problem were presented at CPAIOR '09 in *"Bandwidth-Limited Optimal Deployment of Eventually-Serializable Data Services"* [25], while the results that are obtained on the RAMBO deployment problem were presented at CP '09 in *"Online Selection of Quorum Systems for RAMBO Reconfiguration"* [26]. The papers present both related work that was done on the constraint programming method for these problems, and the constraint-based local search and hybrid methods that have been developed as part of the research presented in this thesis. Chapters 4 and 5 of this thesis provide an expansion on the relevant sections of these papers.

1.3.1 Related Work

The deployment problem was first considered in [4, 5, 6, 7], the results at the time were, however, not satisfactory. The approach consisted of forms of mathematical programming, and many instances of the problem could not be solved in reasonable time constraints. The Eventually-Serializable Data Services deployment problem was revisited in [22], where two methods were used, *mixed integer programming* (a form of mathematical programming) and constraint programming. The results were much more satisfactory. The constraint programming approach was shown to be vastly superior to the mixed integer approach, for this particular problem, and many instances could be solved in very reasonable time. The ESDS deployment problem was then extended to include bandwidth constraints in [27], with less dramatic results.

The RAMBO deployment problem was considered in [26], and a constraint programming solution was developed for it. This solution was able to solve instances of the problem, however, some of instances took a considerable amount of time. This left room for improvement, especially given the tight time constraints in which the problem must be solved due to the *online* nature of the problem.

1.3.2 Comet

COMET is an award-winning tool for solving complex combinatorial optimization problems in areas such as resource allocation and scheduling. One of its main innovations is Constraint-based Local Search, a computational paradigm that combines the ideas of constraint programming and local search. Few combinatorial optimization tools support the concept of constraint-based local search, so the choice of COMET for this thesis seems a logical one. What makes COMET even more appealing is its ability to support multiple solving methods. COMET features both a constraint-based local search engine, a constraint-programming solver, and mathematical programming solver. A highly desirable feature, since this thesis will compare the use of these method on the particular problems.

Comet is an object-oriented language and provides the traditional expressivity of constraint programming with a very rich languages for expressing search algorithms. It adheres to the vision that a combinatorial application is best described as a model and a search component. The model specifies what the solutions are and their overall quality in terms of constraints and objectives. The search expresses how to find solutions. As such it allows the specification of both the model and the search component at a high abstraction level, achieving high reusability while maintaining high performance.

1.3.3 Document Structure

Chapter 1 of this thesis gives an introduction into the problem this thesis attempts to solve and the methods which are used. Chapter 2 gives a brief introduction into constraint programming, which forms the basis of many of the techniques that are used in this thesis. Chapter 3 explains the concept of constraint-based local search, and as such builds upon the principles of constraint programming explained in the previous chapter. Chapter 4 explains the Eventually-Serializable Data Services deployment problem, and describes the methods that were developed to solve it. Chapter 5 details the Reconfigurable Atomic Memory for Basic Objects deployment problem, and the methods used to solve this problem. And finally, chapter 6 presents some conclusions.

2 Constraint Programming

This chapter will give a short overview of constraint programming (CP). Parts of this chapter are based on more complete introductions in [3, 24, 28, 31].

2.1 What is Constraint Programming?

Constraint programming is a programming paradigm where relations between variables are stated in the form of constraints. Constraints essentially are relations over (sets of) unknowns. They are present in nearly all problems we encounter in our day to day lives. When we say "Let's plan a meeting tomorrow in the afternoon" we *constrain* the unknown time of the meeting to a specific day (tomorrow) and a specific time interval (12:00 am to 6:00 pm). Usually constraints come in sets rather than alone, and they are rarely independent. The time of the meeting, for instance, is most likely further constrained by any prior arrangements of the members of the meeting, making them unavailable at certain time intervals.

Constraint programming has been applied to many different areas, such as computer graphics, software engineering, databases, circuit design, finance, and of course combinatorial optimization. Although the fundamental principals are the same, it is not surprising that the specifics of how constraint programming is applied can be of a very different nature, given the diversity of the application domains. This chapter will focus on the use of constraint programming to solve combinatorial optimization problems.

Constraints have a number of important characteristics that make them a natural and transparent way of describing problems. The first important feature of constraints is that they are *declarative*. They specify what relation must hold without specifying a computational procedure to enforce that relation. Another key characteristic is that they are *additive*. The order of imposition of the constraints does not matter, all that matters at the end is that the conjunction of constraints is in effect. Constraints will typically only specify partial information. They do not need to uniquely specify the value of their variables; there can be multiple values that satisfy a constraint or possibly even none (leaving the problem unsolvable). Usually there will be multiple solutions to set of constraints. However, there is no guarantee that there is any solution at all, even if all individual constraints have *local* solutions.

Example: The N-Queens Problem A toy problem that can be stated using constraints is the N-queens problem. This problem will be used throughout this chapter as en example to illustrate the various aspects of constraint programming. The problem consists of placing N queens on a NxN chessboard, with the constraint that no two queens should threaten each other, i.e. they cannot be placed on the same column, row or diagonal. Figure 2.1 shows a possible solution to the 5-queens problem

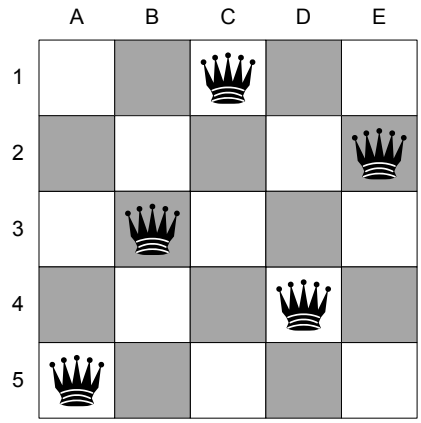


Figure 2.1: A Solution to the 5-Queens Problem

2.2 Constrain Satisfaction Problems

Problems for constraint programming are usually stated in the form of a Constraint Satisfaction Problem (CSP). A CSP is composed of a finite set of variables, each of which is associated with a finite domain, and a set of constraints that restricts the values the variables can simultaneously take. Although most practical problems can be stated in these terms, some problems may require variables with infinite domains or a dynamically changing set of variables. These problems require a different set of specialized techniques to solve. Although these problems are important, they are beyond the scope of this thesis and will not be further discussed.

More formally, a constraint satisfaction problem is defined as:

- A set of variables $X = \{X_1, ..., X_n\}$.
- For each variable $X_i \in X$ a nonempty domain D_{X_i} of possible values.
- A set of constraints $C = \{C_1, ..., C_m\}$. A constraint $C_i \in C$ is a formula in some appropriate (predicate) logic, that involves some subset of the variables and specifies the allowable combinations of values for that subset.

A solution to a constraint satisfaction problem is an assignment of a values to variables, $\{X_1 = v_1, ..., X_n = v_n\}$, with $v_i \in D_{X_i}$. If the solution satisfies all constraints $C_i \in C$ it is called *feasible*. There may be many feasible solutions to a particular problem, and the goal of an algorithm may be to find:

- One feasible solution, with no preference as to which one.
- All feasible solutions.
- An *optimal* feasible solution, which maximizes (or minimizes) some objective function defined in terms of (a subset of) the variables. These problems are usually referred to as Constraint Optimization Problems (COP).

This formal definition may seem limited, but many complex problems can be specified as CSP's in a natural and transparent way.

Example: The N-Queens Problem To formalize the N-queens problem a set of variables, a set of domains and a set of constraints have to be identified. One possible formalization could be to use variables $X = \{Q_1, ..., Q_N\}$ to each denote the location of queen Q_i on the NxN board, with $D_{Q_i} = \{1...N^2\}$. However, by using a slightly different formalization the size of the problem can be significantly reduced. Since no two queens can be placed on a single column each column will contain exactly one queen. Each queen can therefore be associated with a column, leaving only the row in that column to be assigned. This can be formalized by using each variable $Q_i \in X$ to denote the *row* at which the queen in column i is located, thereby reducing the domain of each $Q_i \in X$ to $D_{Q_i} = \{1...N\}$.

This representation ensures that no two queens can be on the same column, this constraint therefore does not have to be formalized. The constraint that no two queens can be on the same row can be expressed as follows:

$$C_1 : \forall i, j \in N : i \neq j \Rightarrow Q_i \neq Q_j$$

The requirement that no two queens can be placed on the same diagonal can be expressed with the following two constraints:

$$C_2 : \forall i, j \in N : i \neq j \Rightarrow Q_i - i \neq Q_j - j$$

$$C_3 : \forall i, j \in N : i \neq j \Rightarrow Q_i + i \neq Q_j + j$$

2.3 Solving Constraint Satisfaction Problems

From a theoretical perspective it is trivial to find a solution to a CSP. One can simply generate all possible assignments of values to variables and test for each assignment whether it satisfies the set of constraints. This naive approach may work for very simple problems but for larger problems it will quickly take an enormous amount of time. Moreover, the typical application domain of CP consists of NP-hard problems. For these problems the naive method is infeasible and more efficient methods will have to be used. Research in the area of constraint programming therefore concentrates on developing algorithms which solve problems faster and more effectively.

Backtracking Search The basis of most solving algorithms is a backtrack search algorithm. This is a depth-first search that picks one variable at a time and chooses a value for this variable. The choice for a variable or value is called a *choice point* and the assignment of a value to a variable is called *labeling*. If labeling the current variable with the chosen value violates certain constraints then the algorithm backtracks to the previous choice point, either choosing a new value or if none are available going back to the previous variable. This process continues until all the variables are labeled and a solution has thus been found, or until all combinations of labels have been tried and have failed, in which case the problem is unsolvable.

This simple backtracking algorithm in itself is still quite naive, it makes minimal use of the constraints to limit and direct the search. One important technique to make the search more efficient is the propagation of the consequences of an assignment on the other variables through the constraints. Another method of enhancing the search is by using heuristics to make smart decisions on which variables and values to select.

Constraint Propagation Constraint propagation is the general term for propagating the implications of a constraint on one variable onto other variables. For instance, when a queen is assigned a certain row in the N-queens problem we then know that row cannot be assigned to any other queen, because of the constraint that no two queens can be assigned to the same row. Whenever a row is assigned that row can therefore be removed from the domain of all remaining variables. Of course, the same holds for the diagonals.

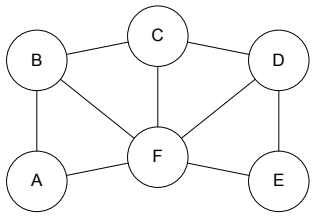


Figure 2.2: Graph Coloring Problem

	A	B	C	D	E	F
Initial	RGB	RGB	RGB	RGB	RGB	RGB
A = R	**R**	G B	R G B	R G B	R G B	G B
C = G	**R**	B	**G**	R B	R G B	B
E = B	**R**	B	**G**	R	**B**	

Figure 2.3: Constraint Propagation in the Graph Coloring Problem

Figure 2.2 shows the graph coloring problem. The problem here is to color the vertices of the graph *red*, *green* or *blue* in such a way that no two neighboring vertices share the same color. Figure 2.3 shows how the labeling of a variable affects the domain of the other variables. Each variable starts with a domain consisting of *red*, *green* and *blue*. When vertex A is labeled *red*, this color is removed from the domain of the neighboring vertices through constraint propagation. When in the subsequent step vertex C is labeled *green*, vertex B and F have their domain reduced to a single value, eliminating the need to branch on them completely. When vertex E is then labeled *blue* there a no legal values left for vertex F.

The propagation of the effect of assignments through the constraints quickly eliminates choices that are inconsistent with the current assignment, thereby greatly reducing the number of branching options. Whenever a variables has no values left in its domain there is no possible solution with the current assignments, eliminating the need to explore that subtree any further. In the graph coloring example an inconsistency is in fact already introduced with the second assignment. Figure 2.3 shows that after vertex C has been labeled *green* the only value left for both vertex B and F is *blue*, however, they cannot both have the same color. Detecting this inconsistency will cause the algorithm to backtrack immediately without even considering the third assignment.

Variable and Value Ordering Another important element of the search is the order in which variables and values are chosen for labeling. The order in which variables are picked can make a significant difference. For instance, in the graph coloring example after the assignments $A = red$ and $B = green$ there is only one possible value left for F. Therefore it makes sense to choose F for the next assignment. The assignment $F = blue$ in fact forces all further assignments. Choosing the variable with the smallest remaining domain is called the *minimum-remaining-values* heuristic. Another heuristic is to choose the variable that is involved in the largest number of constraints, this is called the *degree heuristic*. In the graph coloring example it makes sense to choose F first because labeling it will cause the largest reduction in the domains of the remaining variables. In fact, after F has been assigned you can chose any consistent color at each choice point and arrive at the solution without any backtracking.

Choosing which value to pick can be done using the *least-constraining-value* heuristic. This heuristic selects the value that rules out the fewest choices for the neighboring variables. For instance, assume the assignments $A = red$ and $B = green$ have been produced and now C has been selected for labeling. Here it makes sense to choose red instead of blue, since choosing blue would leave no legal values for F.

```
1 int N = 5;
2 range Size = 1..N;
3 Solver<CP> m();
4
5 var<CP>{int} queen[i in Size](m, Size);
6
7 solve<m> {
8    m.post(alldifferent(queen));
9    m.post(alldifferent(all(i in Size) queen[i] + i));
10   m.post(alldifferent(all(i in Size) queen[i] - i));
11 } using {
12   forall(i in Size) by (queen[i].getSize())
13      tryall<m>(v in Size : queen[i].memberOf(v))
14         label(queen[i], v);
15 }
```

Figure 2.4: The N-Queens Problem in COMET

2.4 Constraint-Programming Languages

Several constraint-programming languages and systems have been developed for solving combinatorial optimization problems. These constraint-programming platforms are typically characterized by two main features:

- An expressive language, offering both a rich constraint language and the ability to specify search procedures.
- A computational model for solving combinatorial optimization, which focuses on using constraints and feasibility information to reduce the search space.

The computational model typically employs the various constraint propagation techniques and handles the backtracking, while the choice for variables and values is left to a user specified search procedure. This ability to specify search procedures is critical to obtain reasonable efficiency on complex combinatorial problems. It allows the use of many different sophisticated heuristics, that can employ problem specific knowledge to explore the search tree more efficiently. The constraint propagation techniques, on the other hand, are problem independent and can therefore be handled by the computational model as efficiently as possible.

Example: The N-Queens Problem Figure 2.4 shows the N-queens problem in the constraint-programming language COMET. It shows the typical structure of constraint programs. First the data declarations on lines 1–3, the declaration of the decision variables on line 5, the statement of the constraints on lines 8–10, and finally the search procedure on lines 12–14. There are N decision variables $queen[i]$ $i \in N$, each representing the row at which the queen in column i is located. Each decision variable has domain $1..N$. The `alldifferent` constraint on line 8 requires each each decision variable to have a unique value, different from all others. This ensures that no two queens can be placed on the same row. The constraints on line 9 and 10 require the same for each diagonal.

The idea of the search procedure is to consider each decision variable, ordered by their domain size and assign them a value out of this domain in a nondeterministic way. It is important to note the difference between the selection of *variables* and *values*. The selection of variables is done through the `forall` instruction, which executes for each value i in *Size*. While values are selected using the `tryall` instruction, which specifies a choice point with a number of alternatives and just one of them must be selected.

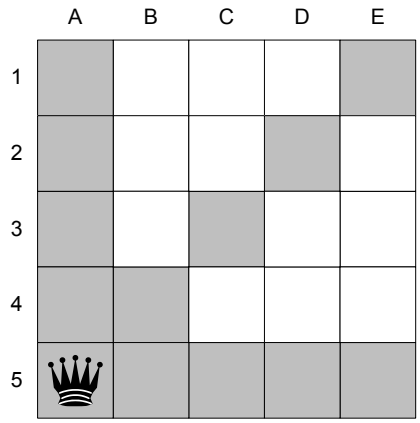


Figure 2.5: First Step in the 5-Queens Problem

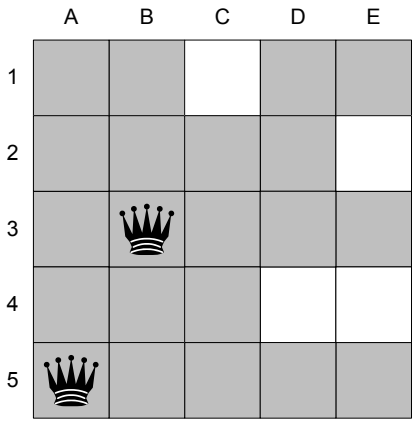


Figure 2.6: Second Step in the 5-Queens Problem

Figure 2.5 and 2.6 show the first two steps of the program. Initially all domains are equal so a queen is selected randomly, here $queen[1]$ is selected and assigned the value of 1. All inconsistent values are then removed from the domains of the remaining queens as shown in figure 2.5. The remaining queens now all have 3 values remaining in their domains, so one is selected randomly again. In this case $queen[2]$ is selected and assigned to row 3. This move leaves only one remaining legal value for $queen[3]$ and $queen[4]$, as shown in figure 2.6. These values are immediately assigned, after which only one legal value is left for $queen[5]$. A solution to the problem has thus been found with only two choices and without backtracking.

3 Constraint-Based Local Search

This chapter provides an introduction into constraint-based local search. A more complete overview can be found in [2, 15, 23, 24], on which parts of this chapter are based.

3.1 Local Search

Local search takes a fundamentally different approach to solving combinatorial optimization problems than the systematic tree search of constraint programming. Compared to Constraint Programming it sacrifices quality guarantees for performance. Local search is, in contrast to constraint programming, not *complete*. There is no guarantee that an optimal, or even high-quality solution will be found. However, on many problems local search algorithms are able to find optimal or near-optimal solutions within very reasonable time constraints. In essence local search explores a graph, moving from solutions to neighboring solutions in the hope of improving the value of the objective function.

Unlike constraint programming, where a solution is build up one variable at a time, local search starts with a (usually randomly generated) solution and moves to neighboring solutions in the hope of improving a function f. This function f measures the quality of a given solution. Typically it consists of either an objective function, in the case of optimization problems, or a measure of distance from the current solution to a *feasible* solution, in the case of satisfiability problems. If the problem is a combination of an optimization and a satisfiability problem, f may be defined as a combination of both an objective function and a satisfiability measure.

Figure 3.1 shows a simple generic local search template. The search starts from an initial solution (line 2), and performs a predefined number of iterations (line 4). The main operation of the algorithm, the move from a solution to one of its neighboring solutions, is performed on line 5. The set of neighboring solution is called the *neighborhood*, and is denoted by $N(s)$. The neighboring solutions may either be *legal* or they may be *forbidden*. The L operator identifies the set of legal neighbors and the S operator selects one of them. The variable $s*$ is used to track the best solution found so far. It is updated each time a solution is found which is satisfiable and which represents an improvement in f over the previously found best solution (line 6–8). When the predefined number of iterations have been performed the best solution that has been encountered is returned (line 8).

Defining the neighborhood and selecting which solution to move to are the two main issues faced when designing a local search algorithm, and most research focuses on these two areas. Local search is particularly appropriate for problems where using systematic searches, such as used in constraint programming, is not feasible. For example, large-scale problems that involve thousands of decision variables, or online optimization problems where a (good) solution must be found within strict time constraints. Local search is the current best approach for many practical problems, such as the traveling tournament problem, vehicle routing, frequency allocation, and many resource-allocation and scheduling problems.

```
1 function LocalSearch {
2     s := GenerateInitialSolution();
3     s* := s;
4     for k:= 1 to MaxIterations do
5         s := S(L(N(s), s), s)
6         if satisfiable(s) and f(s) < f(s*) then
7             s* := s;
8     return s*
9 }
```

Figure 3.1: Basic Local-Search Template

3.2 Constraint-Based Local Search

Constraint-based local search (CBLS) is a relatively recent addition to the field of combinatorial optimization. Historically most research in this area has focused on systematic search and has largely ignored local search. However, the 1990s witnessed significant progress in solving satisfiability problems by local search, and saw the first modeling language for local search being introduced. At the beginning of the 21st century, combinatorial constraints were recognized as beneficial in local search and the idea of constraint-based local search emerged.

Constraint-based local search uses constraints to describe and control local search. Much like in constraint programming, problems are typically separated into a declarative component which describes the problem through the use of constraints, and a search component which details the search procedure.

3.2.1 Modeling

Problems in constraint-based local search are modeled in much the same way as they are in constraint programming. They are stated as Constraint Satisfaction Problems (CSP) or Constraint Optimization Problems (COP). Although the constraints are the same, the way in which they are used is not. Unlike in constraint programming, they are not used to prune the search space, instead they are used to maintain a number of properties which in turn can be used to guide the local search.

Differentiable Objects In constraint-based local search constraints are used to guide the search and determine where to move next. This is done by maintaining certain properties about them, such as their satisfiability, their violation degree, and how much each of its underlying variables contribute to the violations. Local search algorithms typically evaluate the properties of many different neighboring solutions before selecting one. To create an efficient algorithm it is therefore essential to be able to quickly and efficiently calculate these properties.

A neighboring solution typically only differs slightly from the current solution, therefore it is not needed to calculate all properties completely every time. Instead, they can be maintained *incrementally*, so that only the difference needs to be calculated. This concept of incremental variables is one of the key elements for efficient local search algorithms. Incremental variables can be used to construct *differentiable objects*. These object maintain properties about themselves and can be queried to evaluate the effect of local moves.

For constraint-based local search it is of course essential to have constraints as differentiable objects. But the concept can be extended to other types of objects, such as functions. This is essential for optimization problems, where the effect of local moves on the objective function needs to be evaluated. Similar the the additive property of constraints in CP the constraints in constraint-based local search are *compositional*. They can be stated in any order, and can be added at any time.

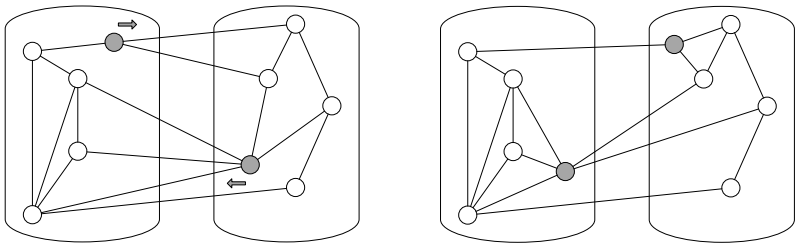


Figure 3.2: A Move in the Graph Partitioning Problem

Furthermore, they can be combined with each other, and with functions, into constraint systems or higher level functions. These in turn can also be used as differentiable objects, so that the effects of moves on a set of constraints or on a combination of constraints and an objective function can be evaluated.

3.2.2 Searching

The neighborhood graph and how it is explored is defined by the search procedure. As mentioned before, the main operation of local search is moving from one solution to a neighboring solution. What such a move consists of defines the neighborhood. Typically a move consists of a simple reassignment of a value to a variable, but other moves are possible, such as multiple reassignments, swapping the value of two or more variables, or even more complex structures. A search procedure can even use multiple different moves, and thus neighborhoods, and alternate between them in search of better solutions.

Figure 3.2 depicts a move in the graph partitioning problem. This problem consists of dividing a graph into evenly balanced partitions so that the number of connections between the two partitions is minimal. The initial solution, shown on the left side of figure 3.2, has 6 connections between the two partitions. When two vertices are swapped only 4 connections remain in the new solution. The move here consists of swapping two vertices between the two partitions. By using this particular move the balancing constraint is implicitly maintained, as long as it is satisfied in the initial solution. Another possible move could have been the reassignment of a vertex to a partition, in this case the balancing constraint could be broken. For some problems allowing such infeasible solutions might give an advantage. However, this means the search will not only have to be directed towards solutions that minimize the number of connections but also towards feasible solutions. There are many different ways to accomplish this. For instance, a feasibility measure could be included in the objective function.

Not all of the neighboring solutions may be *legal*: some might be *forbidden*. For instance, an algorithm might not allow moves that worsen the function f, i.e. moves that introduce more constraint violations or have a worse objective value than the current solution. However, disallowing such moves may cause the neighborhood to become *disconnected*. Connectivity is a fundamental property of a neighborhood, and one of the main issues any local search algorithm faces. A neighborhood is *connected* if a path exist from any solution s to an optimal solution $s*$. To be able to find an optimal solution it is essential that this property holds. However, there is a fundamental conflict between the desire to select the best neighboring solution, and allowing enough moves to keep the neighborhood connected. To be able to find the global optimum you need to select improving solutions, but at the same time this may cause the neighborhood to become disconnected, as you get stuck in local optima. Balancing between the two can be challenging. A commonly used technique to battle this connectedness issue is to incorporate a *diversification* step in the algorithm. The idea is to move to a

```
1  int N = 8;
2  range Size = 1..N;
3  Solver<LS> m();
4  UniformDistribution distr(Size);
5
6  var{int} queen[i in Size](m, Size) := distr.get();
7
8  ConstraintSystem S(m);
9  S.post(alldifferent(queen));
10 S.post(alldifferent(all(i in Size) queen[i] + i));
11 S.post(alldifferent(all(i in Size) queen[i] - i));
12 m.close();
13
14 while (S.violations() > 0)
15    selectMax(q in Size)(S.violations(queen[q]))
16       selectMin(v in Size)(S.getAssignDelta(queen[q],v))
17          queen[q] := v;
```

Figure 3.3: The N-Queens Problem in COMET

random solution when no improvements have been found for a certain amount of time. This enables the algorithm to move to areas of the search space which might otherwise have been disconnected.

The selection of which neighboring solution to move to, out of the legal neighbors, is closely tied with the connectedness issue. A first idea might be to evaluate all of them and select the best one. This is known as a *greedy* algorithm. While this may work well for some problems, it has a high risk of getting stuck in local optima, one of the main problems of any local search algorithm. To counter this, some nondeterminism is usually included in the algorithm. This can be as simple as randomly selecting a neighbor out of the three best neighboring solutions, or it can be more complex. In the *simulated annealing* metaheuristic, for instance, a neighbor is selected randomly and the algorithm will move to this neighbor if it is of a better quality than the current solution. However, even if the solution is of worse quality, the move is still performed with a certain probability, based on the difference in quality between the current solution and the selected neighbor (the bigger the difference the lower the probability of performing the move).

The search procedure of a constraint-based local search algorithm implements heuristics and metaheuristics. A lot of research has focused on developing effective heuristics and metaheuristics. However, which methods will yield the best results remains very problem dependent. Problem specific knowledge can often be incorporated to produce more efficient algorithms. What exactly will or will not work remains hard to predict, making the development of effective local search algorithms somewhat of an art form.

3.3 Example: The N-Queens Problem

Figure 3.3 shows a constraint-based local search program, in COMET, for the N-queens problem, which was earlier described in chapter 2. The CBLS program shows many similarities to the earlier presented constraint programming solution. The data and decision variable declarations are the same, however, in the CBLS program the decision variables are initialized with a random value. This random initialization is necessary because, unlike CP, CBLS starts with a solution.

Lines 8–12 declare the constraints. It important to note that, although the syntax for declaring them is slightly different, these are the very same constraints as were used in the constraint programming example of chapter 2. Line 8 declares a constraint system S, and the constraints are added to S. Constraint systems are differentiable object similar to constraints. The allows the effects of local

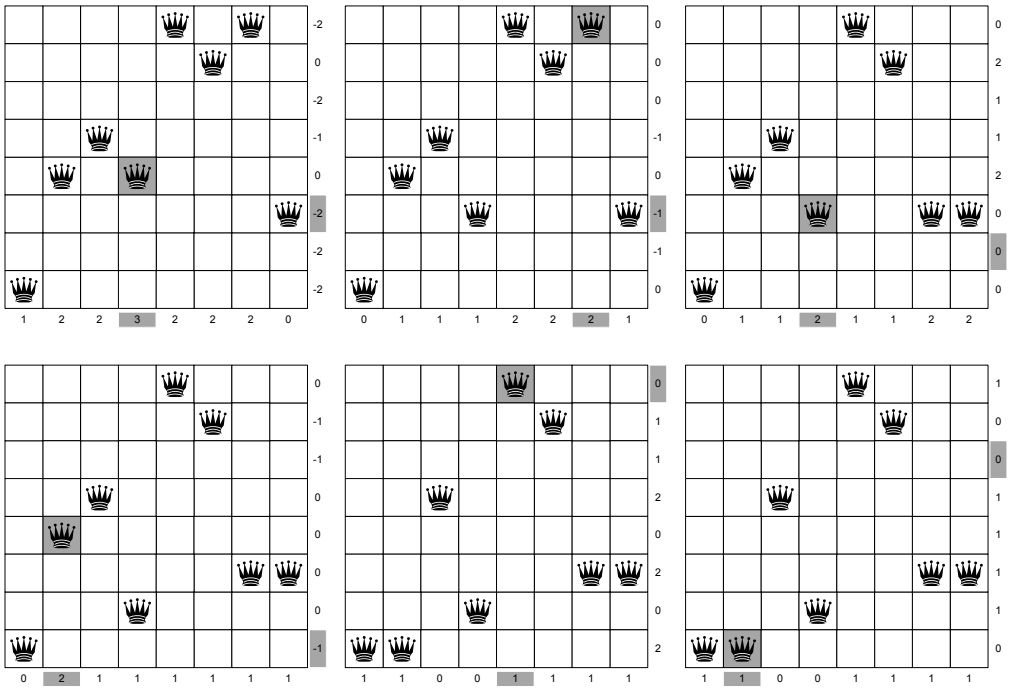


Figure 3.4: Six Steps of the Constraint-Based Local Search Algorithm for the 8-Queens Problem

moves on the set of constraints as a whole to be evaluated. The search procedure therefore does not have to deal with the constraints individually, in fact constraints could be added or removed without the need to modify the search procedure. Also note that, like in constraint programming, this declarative part only specifies what the constraints are and not how to maintain the properties or how to enforce them.

The search strategy for the N-queens problem is shown on lines 14–17. It iterates until the violation degree of the constraint system is zero, meaning that all constraints are satisfied. At each iteration line 15 selects the queen which contributes to the most violations. This is determined using the instruction `S.violations(queen[q])`, which returns the number of violations that queen q is involved in. When a queen is selected, line 16 selects the value for this queen which will cause the largest decrease in the total amount of violations of the constraint system. The instruction `S.getAssignDelta(queen[q],v)` evaluates the effect of the assignment of value v to variable $queen[q]$, on the constraint system S. It reports the decrease (or increase) in the amount of violations of the constraint system as a result of that move. When both a queen and a value have been selected, line 17 performs the assignment and thereby executes the actual move.

Figure 3.4 shows six steps of the algorithm for the 8-queens problem. Each board shows at the bottom of each column the number of violations that the queen in that column is involved in. The gray tab indicates which queen is selected to be moved. The numbers to the right of each row display the gain that will result from moving the selected queen to that row. The gray tab indicates which row will be selected to move the selected queen to. In the first step $queen[4]$ is selected because this queen is involved in the most violations. Moving this queen to row 1, 3, 6, 7 or 8 will reduce the violations by two. Out of these, one is randomly selected, in this case 6. At the second step queens 5, 6 and 7 all are involved in two violations, $queen[7]$ is selected and moved position 6. Note that at step 5 the selected queen is not moved, its current position is one of the most optimal locally.

4 Deployment of Eventually-Serializable Data Services

This chapter presents the Eventually-Serializable Data Services Deployment Problem (ESDSDP). First Eventually-Serializable Data Services is described in section 4.1 and the deployment problem is described in section 4.2. Section 4.3 discusses the constraint programming algorithm that was developed in [27]. Sections 4.4 and 4.5 present a constraint-based local search algorithm and a hybrid CP-CBLS algorithm which were developed for the problem. The models were extensively tested on various benchmarks, described in section 4.6, and the results are discussed and compared to the existing CP model in section 4.7.

4.1 Eventually-Serializable Data Services

Data replication is a fundamental technique in distributed systems: it improves availability, increases throughput, and eliminates single points of failure. There are however extra communication costs associated with data replication in order to maintain consistency among the replicas. Eventually-Serializable Data Services (ESDS) [12] were developed to reduce these extra communication costs. ESDS reduces communication costs by allowing users to selectively relax the immediate consistency requirements in exchange for improved performance, while still ensuring the long-term consistency of the data. It maintains the requested operations in a partial order that gravitates over time towards a total order, while providing clear and unambiguous guarantees about the immediate and long-term behavior of the system.

An Eventually-Serializable Data Service consists of three types of components: *clients*, *front-ends*, and *replicas*. The clients access the data by issuing *requests* and receiving *responses* from the data service. With each operation a client requests it may specify a *prev* set, which contains operations that must be done before the requested operation. The client may also specify a requested operation to be *strict*. A *strict* operation is required to be stable at the time of response, i.e., all operations that precede it must be totally ordered. Operations that are not *strict* may return a result faster, but these operations may be reordered even after a response has been returned. The reordering, however, must always adhere to the client-specified constraints.

Clients issue requests for operations on shared data and receive responses returning the results of those operations. They do not communicate with replicas directly, instead they communicate with front ends. The front-ends keep track of pending requests from the clients and handle the communication with the replicas. They simply relay the requests they receive from the clients to one or more replicas, and relay the responses they receive from the replicas back to the clients. The replicas each maintain a complete copy of the shared data and they *gossip* all operations they have received and processed to all other replicas. Each replica maintains three sets of operations: *pending*,

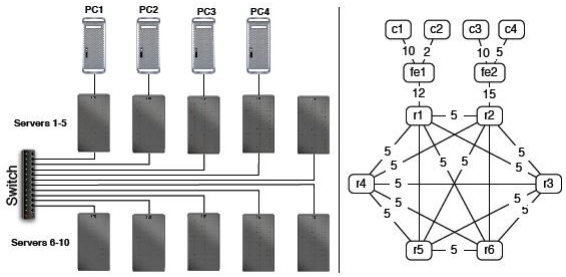


Figure 4.1: A Simple ESDS Deployment Problem

done, and *stable*. An operation is *pending* when the replica has not processed it yet, *done* when the operation has been processed by that replica, and *stable* when that replica knows that it is done at every replica. Initially the operations are in a partial order, but they gravitate towards a total order over time, as they become stable.

ESDS is well-suited for systems which need fault tolerance and a good response time, but do not need immediate consistency of updates. Such systems are for example naming and directory services. A directory service must be robust and have good response times for name lookup and translation requests. Requests to the system are dominated by queries, with infrequent update requests. However, it is unnecessary for the updates to be atomic in all cases. For such a system ESDS offers redundancy to ensure fault tolerance, replication to provide fast responses to queries, and lazy propagation of information for updates.

4.2 Modeling Optimal ESDS Deployments

The deployment of an Eventually-Serializable Data Service deals with the mapping of the *clients*, *front-ends*, and *replicas* of the ESDS onto a set of hosts in a distributed computer network. This deployment is *optimized* by minimizing the total network traffic in the distributed computer network. Finding the optimal deployment can be challenging because of non-uniform communication costs induced by the actual network interconnect, and because of widely varying communication patterns of the various software components. Furthermore, there are certain constraints which put restrictions on what is considered a *feasible* deployment. Three types of constraints limit the possible deployment of components onto hosts. Firstly, to achieve fault tolerance no more than one replica should be placed on a single host. Secondly, some components may require to be co-located on the same host. And finally, not all hosts may be able to support all components, i.e., some hosts may only be able to support clients while others are only able to support replicas.

Typical ESDS instances involve a handful of front-ends, a few replicas, and a few clients. They may not be particularly large as the (potentially numerous) actual users are *external* to the system and simply forward their requests to the *internal* clients. A simple ESDS deployment problem is depicted in figure 4.1. The left side of the figure shows the target distributed computer network and the right side depicts the abstract implementation of the ESDS.

The target network consists of 10 heavy-duty servers interconnected via a switch, and 4 light servers connected via direct connections to the first four heavy-duty servers. For simplicity the cost of sending messages in this network is defined as the number of network hops. For instance, sending a message from $Server_5$ to $Server_9$ *costs* 1 hop (the switch is regarded as a direct connection) and sending a message from PC_1 to PC_2 *costs* 3 hops. The ESDS consists of 4 clients $(c_1, ..., c_4)$, 2 front-

ends (fe_1,fe_2), and 6 replicas ($s_1, ..., s_6$). The connections and associated numbers in the ESDS model of figure 4.1 depict the communication patterns and frequencies of the various components of the ESDS. The four clients send their requests to the two front-ends, the front-ends forward the requests to two replicas, and the replicas constantly gossip updates to each other. The deployment of the software components onto the hosts of the network is subject to the following constraints: the first three client modules (c_1, c_2, c_3) must be hosted on the light servers ($PC_1, ..., PC_4$) while the remaining components (c_4,fe_1,fe_2,$s_1, ..., s_6$) must run on the ten heavy-duty servers ($server_1, ..., server_{10}$). Furthermore, the replicas ($s_1, ..., s_6$) must each be hosted on distinct servers to achieve fault tolerance. The deployment problem consists of finding an assignment of software components to hosts, so that the constraints are satisfied and the total *cost* of network traffic is minimized.

A model for the ESDS deployment was formulated in [4, 6, 22]. The same approach will be followed here. The input data of the deployment model consists of the following:

- The set of software components C.
- The set of hosts N.
- The subset of hosts to which a component can be assigned is denoted by booleans $s_{c,n}$ equal to *true* when component c can be assigned to host n.
- The network cost is directly derived from its topology and expressed with a matrix h where $h_{i,j}$ is the minimum number of hops required to send a message from host i to host j. Note that $h_{i,i} = 0$ (local messages are free).
- The communication frequency. In the following, $f_{a,b}$ denotes the average frequency of messages sent from component a to component b.
- The separation set *Sep* which specifies that the components in each $S \in Sep$ must be hosted on different servers.
- The co-location set *Col* which specifies that the components in each $S \in Col$ must be hosted on the same server.

A decision variable x_c is associated with each software component $c \in C$, which denotes the host $n \in N$ on which component c is deployed.

An optimal deployment minimizes the following:

$$\sum_{a \in C} \sum_{b \in C} f_{a,b} \cdot h_{x_a, x_b}$$

The deployment is subject to the following constraints. Firstly, a components may only be assigned to a host that supports it:

$$\forall c \in C : x_c \in \{i \in N | s_{c,i} = 1\}$$

Secondly, for each separation constraint $S \in Sep$, all components in that constraint are required to be deployed onto different hosts:

$$\forall S \in Sep : \forall i, j \in S : i \neq j \Rightarrow x_i \neq x_j$$

Finally, for each co-location constraint $S \in Col$, all components in that constraint are required to be deployed onto the same host:

$$\forall S \in Col : \forall i, j \in S : x_i = x_j$$

4.2.1 Bandwidth extension

The connections between the hosts in a target network may have bandwidth restrictions due to either limitations of the physical channel or because of QoS guarantees. For this reason the ESDS deployment model was extended with bandwidth constraints in [27]. In this extended model a *connection* between a set of hosts may have a bandwidth constraint. A *connection* in this case is a network interconnect between a set of k machines. This can be, for example, a dedicated point-to-point link between a pair of hosts, or a switched 802.11-wired Ethernet subnet. A deployment platform then reduces to a set of *connections* with some hosts (e.g., routers or machines with several network cards) appearing in several connections to establish bridges.

Formally, a deployment platform is a hypergraph $H = (X, E)$ where X is the set of hosts and E is a set of hyperedges, i.e. $E \subseteq \mathscr{P}(X)\backslash\{\emptyset\}$ where $\mathscr{P}(X)$ is the power-set of X. Each hyperedge either has a specific bandwidth limit, or is bandwidth-unlimited. In the extended model a *connection* c is represented using two properties: a hyper edge, denoted by the set of vertices (or hosts) $c.nSet$ where $c.nSet \subseteq N$; and the bandwidth capacity, denoted by $c.bw$ where $c.bw = \infty$ if there is no bandwidth limitation.

Each pair of hosts i and j is connected by one or more paths, where a path is an ordered set of connections from the source host i to the destination host j. In the ESDS deployment model without the bandwidth extensions it was sufficient to only consider the shortest paths between any pair of hosts. In the extended model, however, it might be needed to consider multiple paths as bandwidth limitations might render the shortest path unusable. For this reason the two-dimensional matrix h which captured the communication cost between a pair of hosts i and j, is replaced by a matrix which captures the communication costs from host i to host j along a specific path p.

Not all possible paths have to be considered, however. It suffices to only consider for each pair of hosts i and j the shortest bandwidth-unlimited path from host i to host j, and any shorter bandwidth-limited path from i to j. Any other path can safely be ignored as it will be dominated by the shortest bandwidth-unlimited path; the shortest bandwidth-unlimited path can always be used instead without incurring a cost increase.

To reflect these added bandwidth limitations the ESDS deployment model was extended as follows [27]. First, the input data is extended with:

- The set of connections *Conn*.
- The set of paths P. In the following, $P_{i,j}$ denotes the set of paths from host i to host j for all $i, j \in N$. Since not all paths need to be considered the set consists of only the shortest bandwidth-unlimited path from i to j (if one exists), and all shorter bandwidth-limited paths. If $i = j$, $P_{i,j}$ contains a single bandwidth-unlimited path of zero length.
- The communication cost of each path $p \in P$, denoted by h_p.
- A boolean matrix *hasC*, where $hasC_{p,c}$ is *true* if path p uses connection c.

Furthermore, the model is extended with the decision variable $path_{a,b}$ for each communicating pair of components $a, b \in C$. This decision variable denotes the path $p \in P_{a,b}$ that is used to send messages from host a to b. Note that each pair of components uses a single directed path to send all messages. However, $path_{a,b}$ and $path_{b,a}$ may be different.

An optimal deployment now minimizes:

$$\sum_{a \in C} \sum_{b \in C} f_{a,b} \cdot h_{path_{a,b}}$$

In addition to the supporting, separation and co-location constraints of the original model a bandwidth constraint is added. For each $c \in Conn$ with $c.bw < \infty$ the total used bandwidth must be less or equal to the available bandwidth:

$$\sum_{a \in C} \sum_{b \in C} f_{a,b} \cdot hasC_{path_{a,b},c} \leq c.bw$$

4.3 Constraint Programming Model

A constraint programming (CP) model for the bandwidth-limited ESDS deployment problem was presented in [27]. This model is shown in figure 4.2. Lines 2–11 show the data declarations, and the decision variables are declared on lines 13–14. To simplify the implementation some details of the CP program diverge slightly from the model described in section 4.2. Instead of the one-dimensional array h a three-dimensional matrix is used, where $h_{i,j,p}$ denotes the length of path p from host i to host j. Similarly, the two-dimensional matrix *hasC* is transformed into a four-dimensional matrix, where $hasC_{i,j,p,c}$ denotes whether path p from host i to host j uses connection c. Analogous to the model of section 4.2, the decision variable $x[c]$ is used to specify the host onto which component c is deployed, with its domain computed from the support matrix s. Finally, variable $path[c1, c2]$ specifies the path used to send messages from component $c1$ to component $c2$, expressed as the rank of the selected path in the set $P[x[c1], x[c2]]$.

Lines 16–20 specify the objective function, which minimizes communication costs. The CP formulation uses the three-dimensional matrix h, which is indexed not only by variables for the two hosts but also by the variable for the particular communication path used between them. The value of the objective function is the summation of the communication frequencies, for each pair of hosts, multiplied by the length of the path between those hosts.

Lines 21–26 contain the co-location and separation constraints. The constraint on lines 27-28 ensures that a path assigned to a pair of hosts is an actual path between those hosts. This constraint is needed because the $path[c1, c2]$ variable only denotes the rank of the selected path in the $P[x[c1], x[c2]]$ set. Its type is therefore int, and its range needs to be reduced to the amount of elements in that set. Lines 29–30 are the bandwidth constraints: for each connection $c \in Conn$, the bandwidth $c.bw$ must be greater than or equal to the sum of the communication frequencies of all pairs of components with c in their chosen path. Note that for the sake of simplicity a bandwidth-unlimited connection is denoted with $c.bw = 0$. The `onDomains` annotation on the various constraints indicate that arc-consistency must be enforced.

The search procedure, depicted in lines 32–46, operates in two phases. In the first phase (lines 32–41) all the components are assigned to hosts, beginning with the components that communicate most heavily. The search must *estimate* the communication cost between components a and b's potential deployment sites along *any* given path. Line 33 picks the first site k for component b, and the `tryall` instruction on line 34 considers the sites for component a in increasing order of path length based on an estimation equal to the shortest path one could take between the choice n and the selection k. The second phase (lines 42–46) labels the path variables, backtracking as needed over the initial component assignments.

```
1  Solver<CP> cp();
2  range C = ...; // The components
3  range N = ...; // The host nodes
4  int[,] s = ...; // The supports matrix
5  int[,] f = ...; // The frequency matrix
6  int[,,] h = ...; // The hops matrix
7  set{set{int}} Sep = ...; // The separation sets
8  set{set{int}} Col = ...; // The co−location sets
9  set{connection} Conn = ...; // The connections
10 set{connection}[,] P = ...; // The paths matrix
11 int[,,,] hasC = ...; // The path/connection matrix
12
13 var<CP>{N} x[c in C](cp, setof(n in N) (s[c,n] == 1));
14 var<CP>{int} path [a in C, b in C](cp,0..max(i in N, j in N) P[i,j].getSize()−1);
15
16 var<CP>{int} obj(cp, 0..1000);
17
18 minimize<cp> obj
19 subject to {
20     cp.post(obj == sum(a in C, b in C: f[a,b] != 0) f[a,b] ∗ h[x[a],x[b],path[a,b]], onDomains);
21     forall(S in Col)
22         select(c1 in S)
23             forall (c2 in S: c1 != c2)
24                 cp.post(x[c1] == x[c2], onDomains);
25     forall(S in Sep)
26         cp.post(alldifferent(all(c in S) x[c]), onDomains);
27     forall (a in C, b in C : f[a,b] != 0)
28         cp.post (path[a,b] < P[x[a],x[b]].getSize(), onDomains);
29     forall (c in Conn: c.bw > 0)
30         cp.post (c.bw >= sum (a in C, b in C: f[a,b] != 0) hasC[x[a], x[b], path[a,b], c] ∗ f[a,b], onDomains);
31 } using {
32     while (sum(k in C) x[k].bound() < C.getSize()) {
33         selectMax(a in C: !x[a].bound(), b in C)(f[a,b]) {
34             int k = min(k in N: x[b].memberOf(k)) k;
35             tryall<cp>(n in N: x[a].memberOf(n))
36                     by (min (i in 0..P[n,k].getSize()−1) h[n,k,i])
37                 cp.post(x[a] == n);
38             onFailure
39                 cp.post(x[a] != n);
40         }
41     }
42     forall (a in C,b in C: f[a,b] != 0 && !path[a,b].bound())
43         tryall<cp> (i in 0..P[x[a],x[b]].getSize()−1) by (h[x[a], x[b], i])
44             cp.post(path[a,b] == i);
45         onFailure
46             cp.post(path[a,b] != i);
47 }
```

Figure 4.2: A CP program for the Bandwidth-Limited ESDS Deployment Problem in COMET

```
1  class LSModel extends Model {
2     Solver<LS>                        ls;
3
4     WeightedConstraintSystem<LS>  S;
5     FunctionSum<LS>                   O;
6     Function<LS>                      C;
7
8     var{int}[,]                       path;
9     var{int}[]                        x;
10
11    LSModel() : Model();
12    void stateModel();
13    var{int} mkWeight();
14 }
```

Figure 4.3: The ESDS Deployment Problem in COMET

4.4 Constraint-Based Local Search Model

The Constraint-Based Local Search program has the same input variables as the ESDS deployment model described in section 4.2 As usual, it consists of a *model* component which defines the constraints and the objective, and a *search* component which defines the search procedure. Both components will be described in detail in this section.

4.4.1 The Model

Figure 4.3 shows the class for the model component of the CBLS program. The class extends the *Model* class (not shown here), which contains the same input data as the CP model of the previous section. The *LSModel* class contains a solver (line 2), a constraint system (line 4), and objectives (lines 5 and 6). A weighted constraint system is used here because all constraints will be given an individual weight, which can be dynamically adjusted during the search. These weights are used to guide the search towards feasible solutions. Two objectives are used, one for the communication cost function, and the other for the combination of the cost function and the constraint satisfaction. The model further contains decision variables for the component-to-host assignment x (line 8), and the path assignment *path* (line 9).

The constructor for the *LSModel* class is shown in figure 4.4. The important part here is the initialization of the two decision variables. The decision variable x (line 7) is defined as an array where each element corresponds to a component of the ESDS model. The value of that element corresponds to the host onto which that component is currently assigned. Each variable in the array receives a domain corresponding with the subset of nodes onto which that component can be deployed. This domain restriction acts as the supports constraint. The same restriction could also have been achieved using a constraint, however, restricting the domain instead is slightly more efficient and thus faster. The *path* decision variable is defined as a two-dimensional matrix (line 8), where each $path[a, b]$ element corresponds with the chosen path from component a to component b. Here each element of the array receives a domain corresponding with the maximum amount of paths available between any two hosts. The domain of each element of the *path* matrix can not be restricted in the same way as was done for the x decision variable. The *path* variable deals with the assignment of paths between *components*, wile the set of paths P deals with the available paths between *hosts*. Which paths are available between a pair of components therefore changes as the deployment of those components onto hosts changes. Since the domain can not be restricted, a constraint will have to be formulated to ensure that the chosen path between a pair of components

```
1 LSModel::LSModel() : Model() {
2    ls = new Solver<LS>();
3
4    S = new WeightedConstraintSystem<LS>(ls);
5    O = new FunctionSum<LS>(ls);
6
7    x = new var{int}[c in C] = new var{int}(ls, setof(n in N) (s[c,n]==1));
8    path = new var{int}[a in C, b in C](ls, 0..max(i in N, j in N) P[i,j].getSize()-1);
9
10   forall (c in C)
11      select(n in setof(n in N) (s[c,n]==1))
12         x[c] := n;
13 }
```

Figure 4.4: Initialization of the ESDS Deployment Problem in COMET

is an actual path between the hosts onto which these components are deployed.

The decision variable x is initialized with random values, on lines 10–12, to provide a nondeterministic starting point for the search. The *path* decision variable uses the default initialization of 0. A random assignment does not give a clear advantage here due to the very limited number of available paths between a pair of hosts. Most component pairs will only have one possible path, leaving 0 the only value which satisfies the constraints.

Figure 4.5 depicts the constraints for the CBLS program of the ESDS deployment problem. It first states the separation constraint on line 2–3, using an `alldifferent` constraint for each element in the *Sep* set. Each individual constraint receives its own weight. These weights act as a multiplication factor for the number of violations of the constraint. They are used to increase or decrease the importance that is placed on a constraint and its violation degree, their exact role will be further explained in section 4.4.2. The weights are created using the `mkWeight` function which is depicted in lines 29–32. They have an initial value of 1, which can be adjusted dynamically throughout the search. Lines 5–8 state the co-location constraint. For each element in the *Col* set, the smallest element in that set is chosen and all other elements in the set are required to be deployed on the same location. Again each constraint is given its own unique dynamic weight. Lines 10–13 depict the constraints that require the assigned path between any pair of components to be in the valid range of paths that are available between the hosts onto which the components are deployed. The final constraint, the bandwidth constraint, is stated on lines 15–19. For each connection with a bandwidth limitation the total amount of traffic on that connection is required to be less or equal to the stated bandwidth for that connection.

The objective functions are stated on lines 21–24. The first objective O is the communication cost function (line 21–22). It is defined as the sum of the communication frequency between each pair of components, multiplied by the length of the chosen path between those components. The second objective C is defined as a combination of the feasibility constraint set S and the communication cost function O.

4.4.2 The Search

The class for the search component of the ESDS deployment problem is depicted in figure 4.6. It extends the in the previous section described *LSModel* class and contains variables that deal with various aspects of the search. The function `search` (not shown) is a wrapper function that calls the `stateModel` function of the *LSModel* class and the `searchProcedure` function. The actual search is performed in the `searchProcedure` function, which is illustrated in figure 4.7. The various aspects of this search procedure are described below.

```
1  void LSModel::stateModel() {
2      forall(s in Sep)
3          S.post(alldifferent(all(c in s) x[c]), mkWeight());
4
5      forall(s in Col)
6          selectMin(c1 in s) c1
7              forall (c2 in s : c1 != c2)
8                  S.post(istrue(x[c1] == x[c2]), mkWeight());
9
10     ConstraintSystem<LS> S2(ls);
11     forall (c1 in C, c2 in C : f[c1, c2] != 0)
12         S2.post(path[c1, c2] < P[x[c1],x[c2]].getSize()-1);
13     S.post(S2, mkWeight());
14
15     ConstraintSystem<LS> S3(ls);
16     forall(c in Conn : c.bw > 0)
17         S3.post(c.bw >= sum(c1 in C, c2 in C : f[c1, c2] != 0)
18             (hasC[x[c1], x[c2], path[c1, c2], c] * f[c1, c2]));
19     S.post(S3, mkWeight());
20
21     forall(c1 in C, c2 in C : f[c1, c2] != 0)
22         O.post(f[c1, c2] * h[x[c1], x[c2], path[c1, c2]]);
23
24     C = S + O;
25
26     m.close();
27 }
28
29 var{int} LSModel::mkWeight() {
30     var{int} w(ls) := 1;
31     return w;
32 }
```

Figure 4.5: Constraints of the CBLS Model in COMET

```
1  class LSSearchProcedure extends LSModel {
2     int                    startTime;
3
4     float                  objChance;
5     float                  divChance;
6
7     int                    it;
8     int                    stableIt;
9     int                    rounds;
10
11    int                    bestValue;
12    Solution               bestSolution;
13    boolean                feasible;
14
15    int                    tabuLen;
16    int                    tabuInc;
17    dict{int -> int}       tabu;
18
19    UniformDistribution    ud;
20    ZeroOneDistribution    zo;
21
22    LSSearchProcedure() : LSModel();
23    void search(DeploymentSolution sol);
24    void searchProcedure(DeploymentSolution sol);
25    void updateBest(DeploymentSolution sol);
26    void updateWeights();
27    void diversify();
28 }
```

Figure 4.6: Search Component of the CBLS Model in COMET

Two Neighborhood Search The search procedure explores two different neighborhoods. These two neighborhoods correspond to the two objectives of the search: minimizing the communication costs and satisfying the constraints. Both objectives can be captured in one objective function, and in fact are in the objective function C. However, it is difficult to strike a good balance between the two and ensure that the right amount of resources is spend on both, when a single neighborhood is used to optimize this single objective. Using different neighborhoods for both objectives gives more control in balancing them. Furthermore, it allows the use of different heuristics and meta-heuristics for both objectives. Both these elements make the two neighborhood approach more efficient for this problem than a single neighborhood approach would have been.

During each iteration of the search procedure one neighborhood is selected (line 3) with probability *objChance* (which has a default value of 70%). The first neighborhood (lines 4–10) optimizes the objective function. The key idea here is to select the variable that is able to cause the biggest decrease in the objective function, and assign that variable the value that achieves this biggest decrease. To do this, first all variables appearing in the objective function O are selected on line 4. Out of these variables the variable which is able to achieve the largest decrease and which is *non-tabu* is selected (line 5). If multiple variables share this largest decrease (and are non-tabu), one is selected randomly. When a variable has been selected, a new value is chosen for this variable, out of its domain. The value that is choosen is the one that achieves the largest decrease in the objective function O (line 6). And finally, the variable is assigned this value (line 7), and is made *tabu* for *tabuLen* iterations.

The second neighborhood (lines 12–20), aims at reducing the amount of violations in the constraint system S. This is done by first selecting the constraint in S with the highest violation degree (line 12). This violation degree is defined as the number of violations of the constraint, multiplied by its weight. On line 14 the variables that appear in this constraint are gathered, and out of these the variable which contributes to the most violations is chosen (line 15). A new value is then selected for this variable so that the objective C is minimized (line 16). And finally, on line 17 the variable is assigned this new value. Note that the second neighborhood does not use a tabu list.

Tabu Search Tabu search [14] is a popular and effective metaheuristic. It is used in the exploration of the first neighborhood. The main idea behind it is to prevent cycles and to get out of local optima by marking previously visited solutions as *tabu*, i.e. forbidden. Tabu search encompasses a great variety of techniques. What is used here is a simple list of variables which are *tabu*. This tabu list is represented by a simple dictionary that records for each variable the iteration number from which point onwards it may be reassigned. Initially this number is 0, meaning the variable may be reassigned. The key idea here is to prevent the search from reassigning the same variable over and over, and instead force the reassignment of many different variables in order to explore a greater portion of the solution space.

Only the first neighborhood uses the tabu list. In this neighborhood the only variables considered for reassignment are the non-tabu ones. This means that the tabu entry for those variables needs to be an iteration number that is less or equal to the current iteration number. Once a variable is reassigned its tabu entry is updated with the current iteration number plus *tabuLen*, thus making the variable *tabu* for *tabuLen* iterations. This *tabuLen* value has to be chosen carefully, too high and not enough moves will be allowed and the solution space may become disconnected. Too low and it will lose its effectiveness and the search may still get stuck in cycles or local optima. Which value is optimal is dependent on a great number of factors, making it hard to predict and unlikely that a single value will be optimal for all instances.

A solution for this problem is to dynamically adjust the value based on information gathered during the search. In [8] a method is described to adjust the value during the search based on the frequency of re-occurrence of solutions. Of course, there is a trade-off here, as there is a large

```
void LSSearchProcedure::searchProcedure(DeploymentSolution sol) {
   while ((System.getCPUTime() − startTime) < 10000) {
      if (zo.get() <= objChance) {
         var{int}[] ox = O.getVariables();
         selectMax(i in ox.getRange() : tabu{ox[i].getId()} <= it)(C.decrease(ox[i])) {
            selectMin(v in ox[i].getDomain())(O.getAssignDelta(ox[i], v)) {
               ox[i] := v;
               tabu{ox[i].getId()} = it + tabuLen;
            }
         }
      } else {
         selectMax(k in S.getRange())(S.getConstraint(k).violations()) {
            Constraint<LS> cls = S.getConstraint(k);
            var{int}[] cx = cls.getVariables();
            selectMax(i in cx.getRange())(cls.violations(cx[i])) {
               selectMin(v in cx[i].getDomain())(C.getAssignDelta(cx[i],v)) {
                  cx[i] := v;
               }
            }
         }
      }
      it++;
      stableIt++;

      if (!feasible && S.violations() == 0)
         feasible = true;

      if (S.violations() == 0 && O.value() < bestValue)
         updateBest(sol);

      if (stableIterations >= 100)
         updateWeights();

      if (rounds >= 200)
         diversify();
   }
   bestSolution.restore();
}
```

Figure 4.7: Search Procedure of the CBLS Model in COMET

increase in memory cost to store the previously visited solutions. A simpler solution is used for the ESDS deployment problem. After a certain number of iterations of the search algorithm the *tabuLen* value is updated based on whether a feasible solution has been found during those iterations. If a feasible solution has been found the value is increased, if no feasible solution has been found it is decreased. The rationale behind this is that experimental results showed that a too large value results in no feasible solutions being found anymore, and that the optimal *tabuLen* value lies right next to this boundary.

The initial value is defined using the following statement.

```
tabuLen = (C.getSize() + sum(c1 in C, c2 in C : f[c1, c2] != 0) 1)/2;
```

This value is equal to half the number of variables that appear in the objective function. This encapsulate the most important factor that influences the *tabuLen* value. Although this initial value will not be equally good for all instances, it has shown good empirical behavior. In general the value will be increased in the course of the search, thus pushing the search into new areas of the solution space and away from previously found local optima.

```
1  void LSSearchProcedure::updateWeights() {
2      with atomic(m) {
3          forall(k in S.getRange() : S.getConstraint(k).violations() > 0) {
4              var{int} wk = S.getWeight(k);
5              wk := wk + S.getConstraint(k).violations();
6          }
7      }
8      stableIt = 0;
9      rounds++;
10 }
```

Figure 4.8: Guided Local Search in the CBLS Model

Guided Local Search As mentioned, each constraint has its own dynamic weight. These weights are used to guide the search towards feasible solutions. This technique is called Guided Local Search [32] and is another well know metaheuristic. Constraints that prove hard to satisfy will have their weight increased so that the search will select them more often for reassignment, in order to satisfy them. Constraints that are easily satisfied will have their weights unaltered. The adjustment of the weight variables is done in the `updateWeight` function shown in figure 4.8. This function is called after the solution has not been improved for at least 100 iterations. The function selects all constraints that are not satisfied and increases their weights by the degree to which they are violated.

Tracking the Best Solution On lines 28-29 of figure 4.7 the new solution is compared to the best solution found so far. Only feasible solutions are considered here. If the new solution has a better objective value then the previously found best solution, then the best solution is updated using the `updateBest` function (not shown here). This function simply signals the *DeploymentSolution* object to store the new solution and updates the various variables to reflect this.

Diversification One of the main challenges any local search algorithm faces is the tendency to get stuck in local optima. When a local optimum has been located it can be hard to get away from, since all surrounding solutions will worsen the objective. A commonly used technique to combat this behavior is to carry out a *diversification* step when no improving solutions have been found for a certain number of iterations. In this diversification step (some) of the decision variables are reset to random values, so that a new area of the search space can be explored.

The search procedure performs such a diversification step, shown in figure 4.9, after updating the constraint weights 200 times (200 rounds). During this step the previously described update of the *tabuLen* variable is carried out (lines 2-5) and all the constraint weights are reset (lines 16-19). The diversification (lines 10-15) consists of selecting all the variables that appear in the objective function and reassigning them with probability *divChance* to a random value in their domain (line 13). This in essence acts as a reset, allowing the search to start from a new random starting point in order to explore a new area of the solution space, possibly disconnected from the previously explored area.

4.4.3 Co-Location Preprocessing

Initial testing showed the CBLS program to have problems with one constraint in particular: the co-location constraint. The co-location constraint is represented by an equality constraint for each element in each co-location set. Because of this formulation, whenever a component in a co-location set is moved, all equality constraints are violated. This induces a *bump* in the optimization value,

```
1  void LSSearchProcedure::diversify() {
2      if (feasible)
3          tabuLen += tabuInc;
4      else
5          tabuLen −= tabuInc;
6
7      feasible = false;
8
9      with atomic(m) {
10         var{int}[] av = C.getVariables();
11         forall(k in av.getRange()) {
12             if (zo.get() < divChance) {
13                 av[k] := av[k].getDomain().getLow() + ud.get(av[k].getDomain().getSize());
14             }
15         }
16         forall(k in S.getRange()) {
17             var{int} wk = S.getWeight(k);
18             wk := 1;
19         }
20     }
21     rounds = 0;
22 }
```

Figure 4.9: Diversification in the CBLS Model

i.e. the amount of violations of the system is increased by the number of elements in the co-location set. Because such moves break a potentially large number of constraints they are very undesirable. The algorithm therefore has difficulty moving a set of components that are required to be co-located, from one node to another.

To solve this problem an alternative representation was developed. This representation replaces each set of components that are required to be co-located, with a single meta-component and changes all constraints and variables which act on the components in the set to now act on the meta-component which represents them. This avoids a large collection of equality constraints and eliminates the problem since all the co-located components can now be moved as one, in a single transition.

The original formulation can be recast into the new formulation with the meta-components using simple preprocessing. Both the CP and CBLS models do not need to be altered for this at all. Post-processing can then recast the solution in terms of the new formulation to the initial formulation, by simply replacing the meta-components by the sets of components which they represent.

4.5 Hybrid Model

The key difference between constraint programming and constraint-based local search is that CP is complete (it will always find the optimal solution), while CBLS sacrifices this completeness for performance (it might not find the optimal solution, but it has a high chance of finding a high quality solution fast). A hybrid of these two methods that has the best of both worlds, i.e. the performance of CBLS with the completeness of CP, seems contradictory. It is precisely the need of searching the whole solution space that makes CP relatively slow, and the sacrifice of this completeness that gives CBLS its performance advantage. However, elements of CBLS can be used to augment the CP search, and possibly speed it up. The key idea is that when the optimal solution is know, or even just a "good" solution, then this information can be used as an upper bound in the search. This allows the search algorithm to quickly discard subtrees of which it can determine that all solutions located in

it will exceed the upper bound. Of course the CP algorithm already uses this technique. Whenever a new solution is located the upper bound is tightened so that the search space can be pruned more efficiently. The key advantage of CBLS is its ability to locate high quality solutions very fast, often faster than the CP algorithm.

In essence this turns the problem of finding the optimal solution into proving the optimality of a given solution, which can be a considerably easier problem in some instances. This idea of hybridization of CP and CBLS can be implemented in two ways: either sequentially, or in parallel. Both methods were investigated and are explained below.

4.5.1 Sequential Hybrid

A sequential hybrid runs a CBLS and CP model sequentially. First the CBLS model is run for a certain amount of time. When it is done, the resulting solution is passed on to the CP model and is used as an upper bound. This upper bound allows the CP search algorithm to discard subtrees that are provably devoid of any solutions with an objective value lower than the upper bound. Of course there is a trade off between the time spend on the CBLS component and time gained in the CP component. There is only a net gain compared to the "pure" CP model if the gain achieved in the CP component is greater than the time spend on the CBLS component.

Implementing this hybridization is straightforward, the CBLS part of the hybrid model is identical to the "pure" CBLS model and the CP part only has one added constraint. This constraint requires the objective to be less than the solution found by the CBLS algorithm.

4.5.2 Parallel Hybrid

A parallel hybrid runs the CBLS and CP models in parallel. Every time the CBLS model finds a new solution the CP model is notified, and the upper bound is tightened. Here there is no trade off between the run time of the CBLS model and the gain in the CP model, since both models are now run in parallel. Worst case scenario is that the CBLS model does not find any solution and/or it is not able to cause any improvement in the CP model. The CP model then still takes the same amount of time as it would have when the CBLS model had not been run at all. This is, however, under the assumption that the two models truly run in parallel and are not competing for the same cpu resources. This of course means a dual cpu/core setup is needed. Furthermore, the parallel run will induce some overhead costs which will slow down the search somewhat, but the delay caused by this will be minimal.

With COMET, the implementation of this parallel hybrid is also straightforward. The models will have to be run in two separate threads, and there needs to be some communication between the two threads, so that each model is notified when a new solution has been found by the other. This communication is achieved very easily in COMET through *events*. The instruction

```
1 c.updateSolution(solution);
```

is added to the CBLS model, and is called every time a new solution is found. The following snippet is added to the CP model to update the upper bound each time a new solution is received from the CBLS component.

```
1 whenever c@updateSolution(Solution s)
2     if (s.getObjectiveValue().compare(cp.getObjective().getPrimalBound()) < 0)
3         cp.setPrimalBound(s.getObjectiveValue());
```

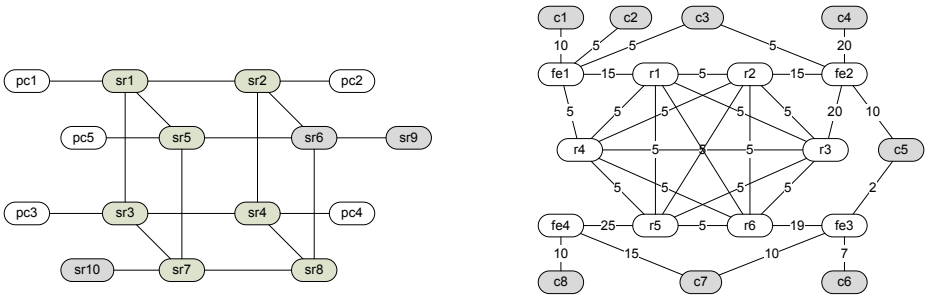


Figure 4.10: Instance HYPER8: Deploying ESDS to a network with many equivalent paths.

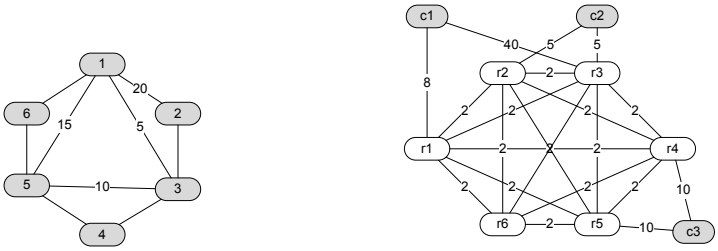


Figure 4.11: Instance RING6: Deploying to a tightly coupled, bandwidth-limited network.

4.6 Benchmarks

To test the performance of a combinatorial optimization algorithm it is essential to use realistic benchmarks. The problem with state-of-the-art systems like ESDS, however, is that not much information is available on actual network configurations being used. The benchmarks described here are synthetic benchmarks that are representative of realistic instances. One of the main characteristics of the used benchmarks is the presence of non-uniform communication costs. While this may not be present in small networks, consisting of machines connected in a small local area network. It is likely to be the case for networks with high redundancy and response time requirements. To guarantee those requirements such networks need to consist of machines located at multiple geographical locations, typically inducing non-uniform communication costs. The communication costs of the benchmark networks is represented by the number of hops. This may seem an unusual choice, but the hops merely represent a measure for distance. They do not necessarily represent actual machines, but can also be regarded as routers or simply long distances.

The benchmarks that are used here are the same ones that were used in [27]. They fall into three categories: variants of the simple ESDS deployment problem depicted in figure 4.1, variants of the HYPER8 ESDS deployment problem shown in figure 4.10, and variants of the RING6 deployment problem shown in figure 4.11.

The HYPER8 and RING6 benchmarks are particularly interesting because they are simple representations of networks with many equivalent alternative paths and networks with tightly coupled hosts. To model the capabilities of the communication infrastructure of a distributed system more realistically, all the benchmarks include, in addition to the components shown, one extra software module between each pair of replicas (components $r_1, \ldots, r_6$). These extra components are "drivers" that manage the communication channels and are required to be co-located with their sending replicas.

			Without co-location pp			With co-location pp		
Benchmark		*Opt*	T_{end}	T_{opt}	$\#Chpt$	T_{end}	T_{opt}	$\#Chpt$
SIM2BW1	μ	214	0,32	0,01	168	0,09	0,05	159
	σ		0,01	0,00	6	0,01	0,02	48
SIM2BW2	μ	234	0,48	0,03	161	0,27	0,09	354
	σ		0,02	0,00	7	0,05	0,07	123
RING4	μ	54	0,39	0,36	393	0,09	0,08	120
	σ		0,09	0,09	185	0,04	0,04	60
RING5	μ	88	10,43	9,95	22146	0,64	0,57	893
	σ		3,83	3,83	9003	0,36	0,36	651
RING6	μ	120	107,72	105,49	244168	9,27	8,98	21730
	σ		30,25	30,25	76673	5,87	5,88	15833
HYP8BW1	μ	522	131,79	18,93	39032	46,87	11,93	32258
	σ		5,78	6,83	2637	1,85	7,85	2204
HYP8BW4	μ	526	2032,19	1329,03	105106	365,97	141,92	47822
	σ		313,03	284,50	20540	64,05	94,18	36567

Table 4.1: Experimental Results for the CP Models

The benchmarks are as follows:

- `SIM2BW1` is a variant of figure 4.1 with a bandwidth limit of 5 on the connection between PC_2 and r_2.
- `SIM2BW2` is a variant of figure 4.1 with a bandwidth limit of 5 on the connection between PC_2 and r_2 and a bandwidth limit of 10 on the connection between PC_3 and r_3.
- `RING4` is a variant of `RING6` with only four gossiping replicas (and no messages from $c3$ to $r5$).
- `RING5` is a variant of `RING6` with only five gossiping replicas.
- `RING6` is illustrated in figure 4.11.
- `HYP8BW1` is a variant of `HYPER8` with a bandwidth limit of 10 on the connection between PC_1 and r_1.
- `HYP8BW4` is a variant of `HYPER8` with four bandwidth-limited connections.

4.7 Experimental Results

4.7.1 Constraint Programming Model

Table 4.1 reports the results for the CP model with COMET 1.1 (executing on an Intel Core 2 at 2.16Ghz with 1 gigabytes of RAM). The column *Opt* reports the objective value of the optimal solution for each benchmark. The time to find the optimum and prove optimality is given by column T_{end}. Column T_{opt} reports the time to find to optimum. And finally, column $\#Chpt$ reports the number of *choice points* that were needed to find the optimum and prove optimality. All results are in seconds, and report the average and standard deviation over 50 runs of the algorithm. Both the results for the representation without co-location preprocessing and the representation with co-location preprocessing are given.

		Without co-location pp			With co-location pp		
Benchmark		*Best*	T_{end}	T_{best}	*Best*	T_{end}	T_{best}
SIM2BW1	μ	214,00	3,55	0,17	214,00	2,78	0,03
	σ	0,00	0,03	0,10	0,00	0,04	0,00
SIM2BW2	μ	234,00	4,08	0,32	234,00	3,40	0,04
	σ	0,00	0,03	0,18	0,00	0,01	0,01
RING4	μ	54,00	5,43	1,00	54,00	4,38	0,67
	σ	0,00	0,03	0,81	0,00	0,03	0,48
RING5	μ	93,00	7,37	2,31	88,00	5,99	1,41
	σ	4,86	0,08	2,29	0,00	0,03	0,65
RING6	μ	128,55	11,58	7,00	120,86	7,96	4,66
	σ	6,99	0,15	2,60	1,14	0,04	1,96
HYP8BW1	μ	535,70	4,67	2,57	523,12	2,68	0,62
	σ	12,57	0,37	1,42	1,81	0,02	0,65
HYP8BW4	μ	554,24	12,44	6,26	542,92	7,64	2,39
	σ	19,50	0,47	3,71	16,20	0,47	2,18

Table 4.2: Experimental Results for the LS Models

The results show that the easiest benchmarks to solve are `SIM2BW1` and `SIM2BW2`, followed by the `RING` variants. The hardest benchmarks to solve are the `HYP8` variants. All benchmarks benefit from the co-location preprocessing. The improvement appears to be related to the fraction of components that can be combined. The largest improvement is for `RING6` where 45 components are reduced to 8, and the smallest improvement is for `HYP8BW1` and `HYP8BW4` where 54 components are reduced to 18.

It is also interesting to note that the algorithm finds the optimum relatively quickly for the `SIM2BW1`, `SIM2BW2`, and `HYP8BW1` benchmarks, where the majority of the total run-time of the algorithm is spend on the optimality proof. While for the `RING` benchmarks the optimum is not found until very late, after which the optimality proof only takes a small amount of time.

4.7.2 Constraint-Based Local Search Model

Table 4.2 reports the results for the CBLS models with COMET 1.1 (executing on an Intel Core 2 at 2.16Ghz with 1 gigabytes of RAM). The *Best* column gives the quality of the best solution found by the algorithm. The total run-time of the algorithm is reported in the T_{end} column. The column T_{opt} reports the time at which the best solution was found. All values are in seconds, and report the average and standard deviation over 50 runs of the algorithm. Again, both the results for the model with and without co-location preprocessing is presented.

The experimental results indicate that CBLS delivers high-quality solutions in a few seconds. The elimination of the co-location constraints is beneficial in several respects. First, it reduces the running time significantly (both to termination and to the best solution), and it has a positive impact on the average best solution found. Second, all the standard deviations improved significantly, indicating that the algorithm is more robust. As the results show, the algorithm was able to find the optimal solution on all 50 runs for the `SIM2BW1`, `SIM2BW2` and `RING4` benchmarks. And when co-location preprocessing was used the optimum is also found every time for the `RING5` benchmark, while getting within 1% of the optimum on average on the `RING6` benchmark.

The hardest benchmark to solve is clearly `HYP8BW4`, however, even here the average best solution found is still within 5% of the optimum. Compared to the constraint programming model, the CBLS model is able to find its best solution faster on all but the `RING4` and `RING5` benchmarks.

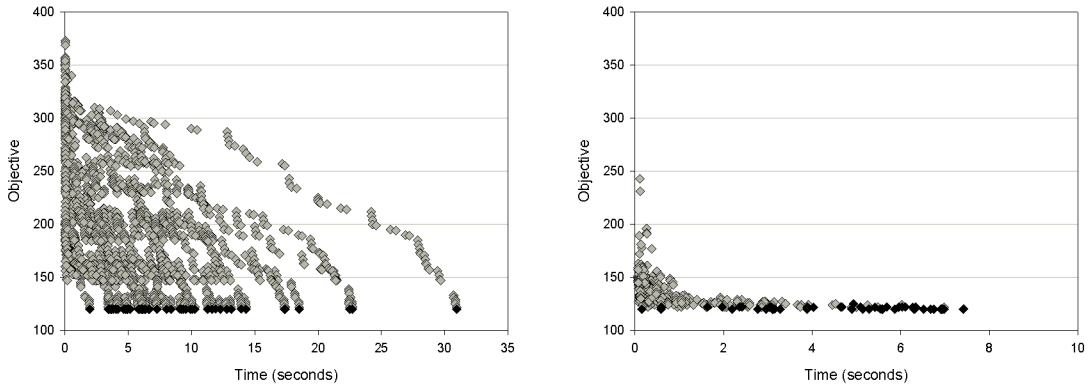


Figure 4.12: Evolution of the objective over time of the RING6 benchmark using the CP model (left) and CBLS model (right).

The graphs in figure 4.12 and 4.13 show the benefit of the constraint-based local search model over the constraint programming model. The graphs show the evolution of the objective over time. Each point in the graph represents a solution found by the algorithm, which is an improvement over the previously found best solution. Each graph combines 50 runs of the algorithm. The black points represent the best solution found during a run, while the gray points represent intermediary solutions found during the run.

The curve in the graphs of the CBLS models is clearly steeper than the curve of the CP models. This means the CBLS model is able to find high quality solutions much faster. This is most dramatic on the RING6 benchmark, shown in figure 4.12 (note the difference on the time scale for both graphs). Here the CBLS model finds a solution with an objective value of 150 or less on all runs of the algorithm in under 1 second, while the CP model is able to do this on hardly any of its 50 runs.

Figure 4.13 shows that the CBLS model has more problems with the HYP8BW4 benchmark, but the algorithm still finds high-quality solutions significantly faster than the CP model (again, note the difference in time scale for both graphs). Even though the CBLS algorithm is not able to find the optimal solution on many of its runs, it shows a clear advantages over the CP model in being able to find high-quality solutions faster. This shows that the CP algorithm could potentially benefit from the CBLS algorithm in a hybrid setting.

Another interesting characteristic of the CBLS graph of figure 4.13 are the spikes near the 2, 4 and 6 second mark. These are caused by the diversification step, and clearly show that the algorithm benefits greatly from this. They show that after a certain amount of time the algorithm no longer finds many new improvements, when the diversification is then carried out (essentially restarting the algorithm from a new random starting point) new and better solution are often found very quickly.

4.7.3 Sequential Hybrid Model

Table 4.3 reports the results for the sequential hybrid model and compares them to the results for the CP model, both obtained using COMET 1.1 (on an Intel Core 2 at 2.16Ghz with 1 gigabytes of RAM). The column *Opt* again reports the optimum for each benchmark. Column T_{end} reports the time to find the optimum and prove optimality. Column T_{opt} reports the time to find to optimum. And column $\#Chpt$ reports the number of *choice points*. All results are in seconds, and report the average and standard deviation over 50 runs of the algorithm. Only the results with co-location

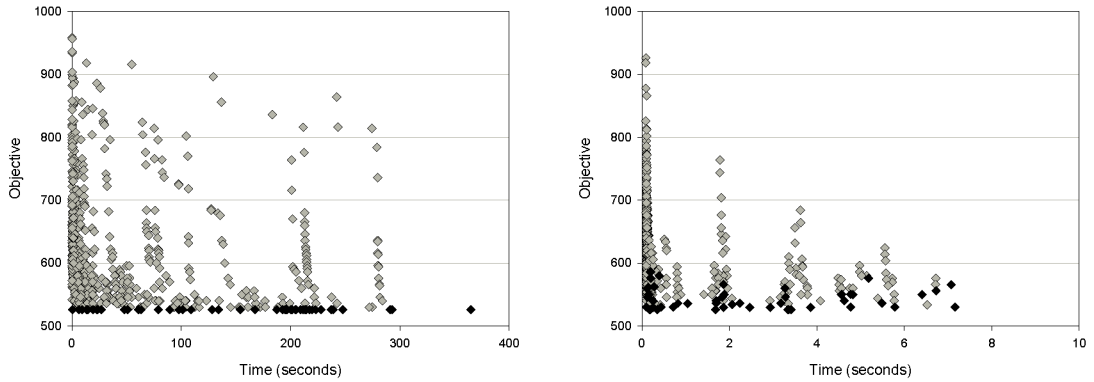


Figure 4.13: Evolution of the objective over time of the HYP8BW4 benchmark using the CP model (left) and CBLS model (right).

preprocessing were used here.

As the results show the sequential hybrid performs worse on the SIM2BW1, SIM2BW1, RING4 and RING5 benchmarks. This is not surprising, due to the already short run-time of the CP model for these benchmarks the run-time of the hybrid will be dominated by the CBLS component. Even though the CBLS model may find the optimal early on it will continue to run for its fixed amount of iterations. The time to the optimum in fact shows that the hybrid finds the optimum slightly faster on the SIM2BW1 and SIM2BW1 benchmarks. The hybrid shows an improvement in both the total run-time and the time to optimum for the RING6, HYP8BW1 and HYP8BW2 benchmarks, although the difference is not that great.

It is also interesting to note the difference in the number of choice points for the sequential hybrid and CP models. Since the choice points are only present in the CP component of the hybrid, they give a good indication of how much the CP component is able to benefit from the CBLS component. Overall, the number of choice points is significantly lower for the hybrid models. The most dramatic are perhaps benchmark RING4, where the number of choice points is reduced from 120 to only 4, and benchmark RING4, where the number is reduced from 21730 to 78.

4.7.4 Parallel Hybrid Model

Table 4.4 reports the results for the parallel hybrid model and compares them to the results for the CP model, both obtained using COMET 1.1 (on an Intel Core 2 at 2.16Ghz with 1 gigabytes of RAM). The column *Opt* again reports the optimum for each benchmark. Column T_{end} reports the time to find the optimum and prove optimality. Column T_{opt} reports the time to find to optimum. And column $\#Chpt$ reports the number of *choice points*. All results are in seconds, and report the average and standard deviation over 50 runs of the algorithm with co-location preprocessing being used.

All benchmarks except for SIM2BW1 and SIM2BW2 show an improvement of the total run-time, compared to the CP model. The time to optimal is equal or better for all benchmarks. The most dramatic improvement can be found in benchmark RING6, which has its run-time reduced from 9,27 seconds to 0,95 seconds. Although the improvement in runtime is not as dramatic for benchmarks HYP8BW1 and HYP8BW2, they do show a significant reduction in time to optimum, 11,93 to 1,74 and 141,92 to 61,87 respectively.

			CP			Sequential Hybrid		
Benchmark		Opt	T_{end}	T_{opt}	$\#Chpt$	T_{end}	T_{opt}	$\#Chpt$
SIM2BW1	μ	214	0,09	0,05	159	2,71	0,03	71
	σ		0,01	0,02	48	0,01	0,00	5
SIM2BW2	μ	234	0,27	0,09	354	3,70	0,04	244
	σ		0,05	0,07	123	0,04	0,01	8
RING4	μ	54	0,09	0,08	120	4,50	0,73	4
	σ		0,04	0,04	60	0,03	0,48	3
RING5	μ	88	0,64	0,57	893	6,33	1,33	21
	σ		0,36	0,36	651	0,06	0,75	2
RING6	μ	120	9,27	8,98	21730	8,57	2,84	78
	σ		5,87	5,88	15833	0,10	2,67	7
HYP8BW1	μ	522	46,87	11,93	32258	45,79	3,25	29677
	σ		1,85	7,85	2204	4,42	5,71	443
HYP8BW4	μ	526	365,97	141,92	47822	343,96	91,69	30114
	σ		64,05	94,18	36567	50,33	76,26	3709

Table 4.3: Experimental Results for the Sequential Hybrid Models

			CP			Parallel Hybrid		
Benchmark		Opt	T_{end}	T_{opt}	$\#Chpt$	T_{end}	T_{opt}	$\#Chpt$
SIM2BW1	μ	214	0,09	0,05	159	0,10	0,05	124
	σ		0,01	0,02	48	0,01	0,01	20
SIM2BW2	μ	234	0,27	0,09	354	0,30	0,06	277
	σ		0,05	0,07	123	0,02	0,01	14
RING4	μ	54	0,09	0,08	120	0,08	0,08	86
	σ		0,04	0,04	60	0,02	0,02	23
RING5	μ	88	0,64	0,57	893	0,29	0,21	197
	σ		0,36	0,36	651	0,06	0,06	89
RING6	μ	120	9,27	8,98	21730	0,95	0,63	840
	σ		5,87	5,88	15833	0,26	0,26	572
HYP8BW1	μ	522	46,87	11,93	32258	42,52	1,74	29696
	σ		1,85	7,85	2204	3,63	1,72	514
HYP8BW4	μ	526	365,97	141,92	47822	322,72	61,97	28577
	σ		64,05	94,18	36567	31,90	70,26	2496

Table 4.4: Experimental Results for the Parallel Hybrid Models

5 RAMBO Deployment

This chapter presents the RAMBO deployment problem and algorithms to solve it. First RAMBO is described in section 5.1 and the deployment problem is described in section 5.2. Section 5.3 discusses the constraint programming algorithm that was developed in [26]. Section 5.4 presents a hybrid CBLS/CP algorithm that was developed for this problem, and section 5.5 presents a parallel composition of the CP and hybrid algorithms. The models were extensively tested on various benchmarks, described in section 5.6, and the results are discussed and compared to the existing CP model in section 5.7.

5.1 RAMBO

Providing consistent shared objects in dynamic networked systems is one of the fundamental problems in distributed computing. Shared object systems must be resilient to failures and guarantee consistency despite the dynamically changing collections of hosts that maintain object replicas. RAMBO, which stands for Reconfigurable Atomic Memory for Basic Objects [13, 20], is a formally specified distributed algorithm designed to offer a solution for this problem. The algorithm provides availability and consistency guarantees in a dynamic network setting, where hosts can continuously join, leave or fail. To achieve this, RAMBO uses *reconfigurable quorum systems*. Availability and fault tolerance are ensured by replicating the objects at hosts that are members of a quorum, while atomicity is ensured through the use of sets of read- and write-quorums that intersect each other.

The quorum *configurations* enable the algorithm to maintain memory consistency in the presence of small and transient changes. In order to accommodate larger and more permanent changes, the algorithm supports dynamic *reconfiguration*, by which the quorum configurations are modified. New configurations may be installed at any time, however, it is still important to install "good" configurations. A poorly crafted configuration may affect RAMBO's performance in negative ways. It may deteriorate the response time if the quorum members are over-burdened and slow to respond, or it may weaken fault-tolerance when failure-prone hosts are chosen to maintain object replicas. In response, RAMBO may need to perform additional reconfigurations, in the hope of installing a better quorum system, possibly thrashing between ill-chosen configurations.

RAMBO supports three activities, all concurrently: reading and writing objects, introducing new configurations, and removing obsolete configurations. Atomicity is guaranteed in all executions. At any time multiple quorum configurations can be active. Each of these configurations consists of a set of *members*, a set of *read-quorums*, and a set of *write-quorums*. Members are hosts for the replicated object, while quorums are subsets of members, with the requirement that every read-quorum has a non-empty intersection with every write-quorum. The algorithm performs read and write operations using a two-phase strategy. The first phase gathers information from at least one read-quorum of every active configurations, and the second phase propagates information to at least

one write-quorum of every active configuration. The information propagates among the participants by means of background gossip. Because every read-quorum and write-quorum intersect, atomicity of the data object is maintained.

The production of a new configuration is an activity that occurs *online* while RAMBO is running. Any member of the latest configuration may propose a new configuration at any time; different proposals are reconciled by an execution of consensus among the members of the latest configuration. The algorithm removes old configurations when their use is no longer necessary for maintaining consistency. This is done by "writing" the information about the new configuration and the latest object value to the new configuration. Exactly *when* to reconfigure and *what* new configuration to choose is not specified by the RAMBO algorithm itself, but is left as decisions for an external service. Given the dynamic nature of the deployment environments, it is neither feasible nor desirable to pre-specify future configurations. The decision of what configuration to choose next ought to be made dynamically in response to external stimuli and observations about the performance of the service.

Optimization techniques can be used to design and deploy sensible configurations that positively affect the performance of read and write operations, while increasing the likelihood that the configuration will be long-lived. Participant ought to be able, based on historical observations, to propose well-designed quorum configurations which are optimized with respect to relevant criteria, such as being composed of members who have been communicating with low latency, and consisting of quorums that will be well-balanced with respect to read and write operation loads. This chapter investigates the use of combinatorial optimization techniques to solve the problem of *what* configuration to reconfigure to. The problem of *when* to reconfigure is not addressed, this is considered to be an application-level decision made on the basis of the observations about the performance of the service and the suspected failures of object replicas.

5.2 Modeling RAMBO Configuration Selection

Designing a configuration can roughly be divided into two tasks: creating an abstract quorum system, and mapping that abstract quorum system to a subset of hosts participating in the service. Although the creation of an "optimal" abstract quorum system in itself is an interesting problem, it is not so much a problem that needs to be solved *online* while the RAMBO service is running. Indeed, the addition, departure or failure of some nodes is unlikely to cause significant changes in the "optimality" of a given quorum system. In contrast, mapping the abstract quorum system to the hosts of the network is very much an online problem. The departure or failure of a host directly impacts the mapping, as the host may need to be replaced by another. Furthermore, changes in the underlying network setting can cause changes in communication delays between the hosts, greatly impacting the "optimality" of a certain mapping.

The model described in this section therefore focuses on the problem of mapping an abstract quorum system onto a set of hosts in a network. It assumes that the participants have a specification of such an abstract quorum system, consisting of the members, the read-quorums, and the write-quorums, at their disposal. The problem which then needs to be solved is to find an assignment of members to hosts participating in the service, in such a way that communication delays for read and write operations are minimized.

Figure 5.1 shows an abstract quorum system and physical network structure. The left side of the figure shows the quorum system, with the horizontal groups, (m_1, m_2, m_3) and (m_4, m_5, m_6), representing the read quorums, while the vertical groups, (m_1, m_4), (m_2, m_5), (m_3, m_6), represent the write quorums. The right side of the figure shows the representation of a physical network structure. The numbers associated with each host denote the average frequency of read and write operations for that host. The deployment problem consists of assigning each member $m_1, ..., m_6$ to a unique host in the network, in such a way that the resulting configuration is "optimal".

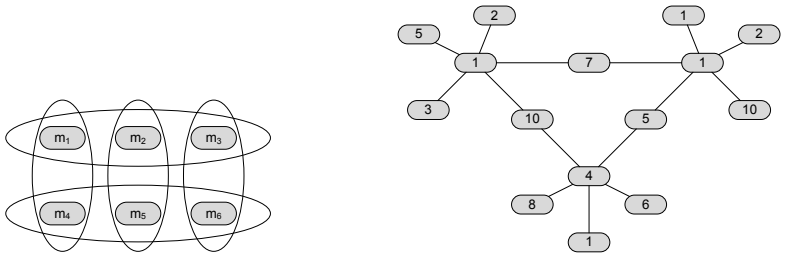


Figure 5.1: Abstract Quorum System and Physical Network Structure

Hosts in a dynamic network setting generally have no knowledge of the underlying network, particularly as nodes join and leave. This leaves them very little basis on which to estimate the reliability of other nodes in the network. The model described in this section assumes that the best available estimate of the network connections and host performance is the measurement of average round trip message delays. In particular, a measurement of round trip delay from host h_1 to host h_2 and back to h_1 can be used as an assessment of:

- The communication "distances" between h_1 and h_2.
- The communication loads on h_1 and h_2.
- The processing loads at h_1 and h_2.
- The likelihood that host h_2 is failing, or even failed.

The hosts can record delay measurements as running averages for the recent past, reasonably assuming that the measurements are quickly impacted by failures or slowdowns. Each host also measures the average frequency of read and write operations it initiates. The hosts share gathered operation frequencies and messaging delays by adding them to the gossip messages of RAMBO. The overall guiding principle is that observations of current behaviors are the best available predictors of future behaviors.

To implement this the RAMBO algorithm requires only minimal changes. The sole modification is the addition of local observations about system performance piggy-backed onto the gossip messages. Upon receipt of a gossip message, a host updates its local knowledge using the piggy-backed information and then delivers the message to RAMBO. Any participant can compute a new configuration using only the local knowledge about the behavior of the participants and submit it directly to the RAMBO reconfiguration service.

Once a mapping is computed, the resulting configurations may be augmented to include information recommending the best read and write quorums for each host to use, where use of best quorums will result in minimum delays for read and write operations and the most balanced load on replica hosts (until failures occur in those best quorums). Of course the use of this information is optional and a host can always use other quorums if it does not observe good responses from the recommended quorums.

A more formal model for the RAMBO deployment problem was described in [26]. The input data of this model consists of the following:

- The set of hosts H.
- For every host $h \in H$, the average frequency f_h of its read and write requests.
- For every pair of hosts $h_1, h_2 \in H$, the average round trip delay d_{h_1,h_2} of messages from h_1 to h_2.
- The abstract configuration c to be deployed on H, where c consists of:
  - The set of members M, each of which maintains a replica of the data.
  - The set of read quorums $R \subseteq \mathscr{P}(M)$.
  - The set of write quorums $W \subseteq \mathscr{P}(M)$.
- The load-balancing factor α, restricting the spread of loads on the configuration members, assuming each host first tries to contact its fastest read and write quorums.

For each member $m \in M$ a decision variable x_m with domain H denotes onto which host $h \in H$ member m is deployed. Each host $h \in H$ is also associated with two decision variables $readQ_h$ and $writeQ_h$ (with domains R and W) denoting, respectively, one read and one write quorum from the set of read (write) quorums with minimum average delay to h. Finally, an auxiliary variable $load_m$ represents the load of a configuration member m which is induced by the traffic between the hosts and their chosen read/write quorums.

An optimal deployment minimizes:

$$\sum_{h \in H} f_h \cdot \left(\min_{q \in R} \left(\max_{m \in q} d_{h,x_m} \right) + \min_{q \in W} \left(\max_{m \in q} d_{h,x_m} \right) \right)$$

Each term in the summation captures the time it takes in RAMBO for a host h to execute the read/write phase of the protocol. A read requires the client to contact all the members of at least one read quorum, before it can proceed to the write phase of the round of communication. Similarly, the write phase requires the client to contact all the members of at least one write quorum to update the data item. The $\max_{m \in q} d_{h,x_m}$ term reflects the time it takes to contact all the members of quorum q. Since one must wait for all members to respond, the wait time is equal to the response time of the slowest member. The outer $\min_{q \in R}$ term reflects the fact that RAMBO must hear back from one quorum before it proceeds, therefore a reply from just the fastest quorum suffices.

A configuration is subject to the following constraints. First, all configuration members must be deployed on separate hosts:

$$\forall m, m' \in M : m \neq m' \Rightarrow x_m \neq x_{m'}$$

An implementation of RAMBO can use different strategies when contacting the read and write quorums. A conforming implementation might simply contact all the read quorums in parallel. Naturally, this does not affect the value of the objective, but it induces more traffic and work on the members of the quorum system. Another strategy for RAMBO is to contact what it currently perceives as the fastest read quorum first and fall back on the other read quorums if it does not receive a timely response. It could even select a read quorum uniformly at random. These strategies strike different trade-offs

between the traffic they induce and the workload uniformity. The model described below captures the greedy strategy, namely, RAMBO contacts its closest quorum first, and the model assumes that this quorum replies (unless a failure has occurred).

The model uses the $readQ_h$ and $writeQ_h$ variables of host h to capture which read and write quorum RAMBO contacts. Since the quorums with minimal delay are contacted, the value for $readQ_h$ and $writeQ_h$ is restricted to these optimal quorums. Note however that several quorums might deliver the same minimal delay, so this restriction alone does not determine the ideal read and write quorum. More formally:

$$readQ_h = r \Rightarrow \max_{m \in r} d_{h,x_m} = \min_{q \in R} \left(\max_{m \in q} d_{h,x_m} \right)$$

$$writeQ_h = w \Rightarrow \max_{m \in w} d_{h,x_m} = \min_{q \in W} \left(\max_{m \in q} d_{h,x_m} \right)$$

The load on a configuration member m depends on which read and write quorums are chosen among those that induce minimal delays. It is defined as follows:

$$load_m = \sum_{h \in H} \left(\sum_{m \in readQ_h} f_h + \sum_{m \in writeQ_h} f_h \right)$$

Finally, the load-balancing constraint requires the maximum load on a member to be within a factor α of the minimum load. In this α characterizes the width of the band between the maximum and minimum loads. A lower value of α results in smaller band and a more restrictive constraint.

$$\max_{m \in M} load_m \leq \alpha \cdot \left(\min_{m \in M} load_m \right)$$

5.3 Constraint Programming Model

A constraint programming model for the RAMBO deployment problem was presented in [26]. This model is shown in figure 5.2. The data declarations on lines 2–8 correspond to the input data of the RAMBO model of section 5.2. Line 9 declares an additional input used for breaking symmetries among the members of the quorum configuration. Lines 10–15 define derived data. Specifically, $nbrQ[m]$ is the number of quorums in which m appears, and $degree[h]$ is the number of neighbors of host h in the logical network graph. RQ and WQ are the index sets of the read and write quorums, respectively. The auxiliary matrices $readQC$ and $writeQC$ are encodings of the quorum membership, e.g., $readQC[i, j] = true \Leftrightarrow j \in R[i]$.

Lines 17–22 declare the decision variables. Variable $x[m]$ specifies the host of configuration member m. Variables $readD[h, r]$ and $writeD[h, w]$ are the communication delays for host h to access read quorum r and write quorum w. The variables $readQ[h]$ and $writeQ[h]$ represent the read and write quorum selections for host h. Finally, the variable $load[m]$ represents the communication load on configuration member m, given the current deployment and quorum selections.

Line 25 specifies the objective function, which minimizes the total communication delay over all operations. Line 27 specifies the fault tolerance requirement, namely, all members of the configuration must be deployed to distinct hosts. Lines 28–29 break the variable symmetries among the configuration members, similar to the method described in [29]. Lines 30–40 constrain the auxiliary delay variables and quorum selection variables needed in the load-balancing constraint. The constraints on lines 31 and 33 capture the delays incurred by host h to use a read (write) quorum.

```
1  Solver<CP> cp();
2  range M = ...; // The members of the quorum configuration
3  set{int}[] R = ...; // An array storing all the read quorums in the configuration
4  set{int}[] W = ...; // An array storing all the write quorums in the configuration
5  range H = ...; // The host nodes
6  int[] f = ...; // The frequency matrix
7  int[,] d = ...; // The delays matrix
8  int alpha = ...; // The load factor
9  set{tuple{int low; int high}} Order = ...; // The order of quorum members
10 int nbrQ[m in M] = .... ; // The number of quorums for each member
11 int degree[H] = ...; // The degree of a host (number of neighbors)
12 range RQ = R.getRange();
13 range WQ = W.getRange();
14 boolean readQC[RQ, M] = ...;
15 boolean writeQC[WQ, M] = ...;
16
17 var<CP>{int} x[M](cp, H);
18 var<CP>{int} readD[H, RQ](cp, 0..10000);
19 var<CP>{int} writeD[H, WQ](cp, 0..10000);
20 var<CP>{int} readQ[H](cp, RQ);
21 var<CP>{int} writeQ[H](cp, WQ);
22 var<CP>{int} load[M](cp, 0..10000);
23
24 minimize <cp>
25    sum(h in H) f[h] * (min(r in RQ) readD[h, r] + min(w in WQ) writeD[h, w])
26 subject to {
27    cp.post(alldifferent(x), onDomains);
28    forall (o in Order)
29       cp.post(x[o.low] < x[o.high]);
30    forall (h in H, r in RQ)
31       cp.post(readD[h, r] == max(m in R[r]) d[h, x[m]]);
32    forall (h in H, w in WQ)
33       cp.post(writeD[h, w] == max(m in W[w]) d[h, x[m]]);
34    forall (h in H) {
35       cp.post(readD[h, readQ[h]] == min(r in RQ) readD[h, r]);
36       cp.post(writeD[h, writeQ[h]] == min(w in WQ) writeD[h, w]);
37    }
38    forall (m in M)
39       cp.post(load[m] == sum(h in H) f[h] * (readQC[readQ[h], m] + writeQC[writeQ[h], m]));
40    cp.post(max(m in M) load[m] <= alpha * min(m in M) load[m]);
41 } using {
42    while (sum(k in M) x[k].bound() <M.getSize())
43       selectMax(m in M: !x[m].bound()) (nbrQ[m])
44          tryall<cp>(h in H : x[m].memberOf(h)) by (-degree[h])
45             cp.label(x[m], h);
46          onFailure
47             cp.diff(x[m], h);
48    once<cp> {
49       forall (h in H : !readQ[h].bound() || !writeQ[h].bound()) by (-f[h]) {
50          label(readQ[h]);
51          label(writeQ[h]);
52       }
53    }
54 }
```

Figure 5.2: Constraint Programming Model in COMET

Lines 34–37 require the quorums assigned to host h, namely $readQ[h]$ and $writeQ[h]$, to be among the quorums with minimum delay for that host. Lines 38–39 specify the communication load on m as the sum of the operation frequencies of each host for which m is a member of its assigned read and/or write quorum. Line 40 is the load-balancing constraint and requires the load on the most heavily loaded configuration member to be no more than *alpha* times the load on the most lightly loaded configuration member.

The search procedure operates in two phases. The rational for using two phases is that the problem can be decomposed into two more or less independent sub-problems. Only the assignment of configuration members to hosts impacts the objective; the decision variables which assign read and write quorums to hosts do not appear in the objective function. Optimizing the objective function can therefore be done without assigning read and write quorums. This assignment is only relevant for determining whether an assignment of quorum members to hosts satisfies the load balancing constraint. Furthermore, the load balancing constraint provides little to no guidance on the assignment of quorum members. The problem can therefore be effectively decomposed into a first phase which optimizes the objective function by finding an optimal assignment of quorum members to hosts, and a second phase which determines for each solution of the first phase whether there is an assignment of read and write quorums to hosts that satisfies the load balancing constraint.

The first phase is shown on lines 42–47. The variable selection heuristic first focuses on variables that appear in many quorums. The value selection heuristic first considers hosts that have many neighbors 'close by' as these would be ideal locations for quorum members. The second phase is depicted on line 48–53. Since the second phase does not affect the objective, only one assignment of read and write quorums that satisfies the load balancing constraint is needed. This explains the `once<cp>` annotation enclosing the second phase call. The phase two procedure considers the most "talkative" hosts first (by decreasing frequencies) and attempts to assign one of the remaining legal (minimal) quorums from its domain.

5.4 Hybrid CBLS/CP Master-Slave Algorithm

Determining a new configuration for RAMBO is, unarguably, an online problem. It must be solved quickly so a new proposal can be submitted by RAMBO for consensus with the other participants. While a constraint programming approach is appealing based on its ability to prove optimality, it might not scale nicely or may take a long time to establish optimality. For this reason a constraint-based local search solution is considered, based on the assumption that local search can deliver high-quality solutions in short-order, a highly desirable property in an online setting.

A first natural attempt uses a search procedure with a neighborhood structure that re-assigns either the deployment or the quorum selection variables. Unfortunately, this *direct* approach is unsuccessful as load-balancing provides little to no guidance on the quorum selection until the deployment is fixed. The recognition of this difficulty suggests a second approach where a master local search is first used to find a deployment that minimizes the communication volume. For each candidate solution produced in the master, a slave model focuses on finding a quorum selection. This decomposes the problem in a similar fashion as the two phase search strategy of the CP model. Note that the slave does not affect the objective. Instead, it handles the feasibility of the load-balancing constraint. Finding a feasible quorum selection is typically quite hard in its own right and to solve this sub-problem a constraint programming model is used. This master-slave approach is reminiscent of Benders decompositions [9].

As usual, the CBLS/CP program is divided into a model and a search component, which will both be described in detail in this section.

```
1  void RAMBO::stateModel() {
2      x = new var{int}[h in H](ls, H) := h;
3
4      UniformDistribution ud(H);
5      forall (h1 in H)
6          select (h2 in H)
7              x[h1] :=: x[h2];
8
9      readD = new var{int}[h in H, r in RQ](ls) <- max (m in R[r]) d[h, x[m]];
10     bestReadQ = new var{set{int}}[h in H](ls) <- argMin (r in RQ) readD[h, r];
11     readQMax = new var{int}[r in RQ](ls) <- sum(h in H : member(r, bestReadQ[h])) f[h];
12     readQMin = new var{int}[r in RQ](ls) <- sum(h in H : card(bestReadQ[h]) == 1 &&
13                                                          member(q, bestReadQ[h])) f[h];
14     ...
15
16     loadMax = new var{int}[m in M](ls) <- sum (r in RQ : member(m, R[r])) readQMax[r] +
17                                           sum (w in WQ : member(m, W[w])) writeQMax[w];
18     loadMin = new var{int}[m in M](ls) <- sum (r in RQ : member(m, R[r])) readQMin[r] +
19                                           sum (w in WQ : member(m, W[w])) writeQMin[w];
20
21     S = new ConstraintSystem<LS>(ls);
22     S.post (max (m in M) loadMax[m] <= alpha * min (m in M) loadMax[m]);
23
24     O = new FunctionSum<LS>(ls);
25     forall (h in H)
26         O.post(f[h] * (min (r in RQ) (max (m in R[r]) d[h, x[m]]) + min (w in WQ) (max (m in W[w]) d[h, x[m]])));
27
28     violations = new var{int}(ls, 0..10000) := 10000;
29     weight = new var{float}(ls) := 0.01;
30     obj = new var{float}(ls) <- sqrt(O.value() ^ 2 + (weight*violations) ^ 2);
31
32     ls.close();
33 }
```

Figure 5.3: Constraint-Based Local Search Model in COMET

5.4.1 The Model

Figure 5.3 shows the model component of the CBLS program, in COMET. It receives the same input data as the model described in section 5.2 and the CP program of section 5.3. The input variables are not shown here for brevity reasons. Line 2 declares the deployment decision variable x. The range of x is extended from the range of *members* to the range of *hosts*. Although a deployment only requires the assignment of hosts to members, it is useful to extend the number of deployment variables, so that a neighborhood structure based on swaps can be used. With the number of x variables equal to the number of hosts it is possible to create an initial assignment where each host is assigned to precisely one x variable. The variables $x[i]$ with $i \notin M$ do not represent an actual deployment, however, with only the use of swaps as moves, the constraint that requires each member to be deployed on a unique host is now implicitly enforced. The initial assignment is created on line 2, while lines 4–7 perform a series of random swaps to create a non-deterministic starting point.

Lines 9-13 declare invariants which maintain a number of key properties related to the read quorums: the invariant $readD[h, r]$ maintains the communication delay from host h to access read quorum r, $BestReadQ[h]$ maintains the set of read quorums with a minimal delay to host h, while invariants $readQMax[r]$ and $readQMin[h, r]$ maintain the maximum and minimum load, respectively, on read quorum r given the current assignment. Determining the actual load on a quorum requires the assignment of read and write quorums to each host. Since this assignment is not always available

it is useful to maintain these maximum and minimum loads. The maximum load is defined as the load quorum r would have when all hosts for which r is among the quorums with minimal delay (which represent the only legal choices) would choose r as their read quorum. Similarly, the minimum load is defined as the load quorum r would have when all hosts which have an option of not choosing r will indeed not choose r as their read quorum. Similar invariants are declared for the write quorums, but are not shown here for brevity reasons.

The maximum and minimum loads for read and write quorums are used by the $loadMax[m]$ and $loadMin[m]$ invariants to determine the maximum and minimum load on configuration members. The true load, $load[m]$, on configuration member m satisfies:

$$loadMin[m] \leq load[m] \leq loadMax[m]$$

The constraint stated on line 22 serves as an *estimation* of the actual load-balancing constraint. This estimation compares the maximum loads on configuration members. The actual load-balancing constraint requires the assignment of read and write quorums to hosts. Determining this assignment is a costly process, therefore the estimation will be used whenever possible. Note that the constraint is neither an upper bound nor an lower bound, it is merely an approximation. Satisfaction of the estimation constraint does not guarantee satisfaction of the true load-balancing constraint, and violation of the estimation constraint does not guarantee violation of the true constraint.

Lines 24–26 declare the objective function O as the sum of frequencies multiplied by the communication delay. A variable representing the number of violations is declared on line 28, and a weight is declared on line 29. These variables are used in the objective function *obj*, which combines the earlier defined objective, with the amount of violations. The emphasis that is placed on the violations can be varied using the weight.

5.4.2 The Search

As mentioned, the search algorithm is decomposed into a master and a slave search. The master attempts to find an assignment of hosts to members which minimizes the communication volume. For each assignment produced by the master, the slave attempts to find an assignment of read and write quorums to hosts, so that the load-balancing constraint is satisfied.

Simulated Annealing The CBLS master is shown in figure 5.4. The search algorithm is based on the *simulated annealing* metaheuristic. Simulated annealing (SA) is inspired by the physical process of annealing in metallurgy, which described the manner in which a metal cools and freezes into a minimum energy crystalline structure. It was first applied to combinatorial optimization in [18, 21].

In simulated annealing some "bad" moves, which deteriorate the objective, are accepted, in order to allow the exploration of a greater portion of the search space. A move is accepted with probability:

$$e^{-\Delta O/T}$$

Where ΔO is the difference in the objective function caused by the move, and T is a *temperature*. If ΔO is negative (the objective was improved) the move is always accepted, if ΔO is small the probability that the move is accepted is high, while that probability is low if ΔO is large. Similarly, if the temperature T is high, "bad" moves have a high probability of being accepted, while that probability becomes smaller as the temperature is lowered. By starting with a high temperature the algorithm allows the exploration of a great portion of the search space, without getting stuck in local minima, while converging onto a (global) minimum as the temperature cools down. In essence,

```
1  void RAMBO::search() {
2      int iterations, stableIterations, rounds = 0;
3      int bestFeasible, bestInfeasible, bf, bi = System.getMAXINT();
4      float temp = 15.0;
5      ExponentialDistribution distr();
6
7      while (iterations < 50000) {
8          select (m in M) {
9              select (n in H : n != m) {
10                 float delta = lookahead(ls, obj) makeMove(m, n); − obj;
11
12                 if (distr.accept(−delta/temp)) {
13                     makeMove(m, n);
14
15                     if (violations == 0)
16                         bf = min(obj, bestFeasible);
17                     else
18                         bi = min(obj, bestInfeasible);
19                 }
20             }
21         }
22         iterations++;
23         stableIterations++;
24         temp = temp∗0.9995;
25         weight := weight + 0.01;
26
27         if (bf < bestFeasible || bi < bestInfeasible) {
28             bestFeasible = bf;
29             bestInfeasible = bi;
30             stableIterations = 0;
31             if (violations == 0)
32                 updateBest();
33         }
34
35         if (stableIterations >= 1000)
36             reheat();
37
38         if (rounds >= 10)
39             diversify();
40     }
41 }
```

Figure 5.4: CBLS Master Search in COMET

the method starts out as a random walk where almost all neighboring solutions are accepted, and smoothly transitions more and more into a hill-climbing heuristic where only improving solutions are accepted.

Moves are usually randomly selected in simulated annealing. In the algorithm of figure 5.4 a move is randomly selected on lines 8–9, and the move is evaluated on line 10. The instruction `lookahead(ls, obj) makeMove(m, n);` returns the value that objective *obj* will receive if the move is performed, by subtracting the current value of objective *obj* the difference *delta* is obtained. Line 12 accepts the move with probability $e^{-delta/temp}$. If the move is accepted, it is performed on line 13, and variables *bf* and *bi* are updated to reflect the best feasible and best infeasible solution found so far on lines 15–18.

After each iteration the temperature is lowered on line 24. In addition to lowering the temperature the weight is adjusted on line 25. This weight is used to vary the emphasis that the objective function places on the violations of the load-balancing constraint. Initially the weight is low so that the main focus is on minimizing the communication costs. By increasing the weight each iteration the emphasis slowly shifts towards finding feasible solutions. The temperature and weight work in tandem. They allow the search algorithm to first explore a large portion of the search space, without paying much attention to the feasibility of the solutions, in order to find areas with low communication costs. As the temperature decreases and the weight simultaneously increases, the search will converge onto feasible solutions with minimal communication costs.

Tracking the Best Solution The best solutions found during the search are recorded on lines 27–33. Although only *feasible* solutions are eventually stored on line 32, the algorithm also records the best *infeasible* solution it encounters. These infeasible solutions are recorded so that improvements in the communication cost function can be registered, even if the resulting solutions do not satisfy the load-balancing constraint. This is done to prevent unnecessary and premature triggering of the reheating and diversification functions, when improvements can still be found.

Reheating and Diversification When no improvements have been seen for 1000 iterations a reheating step is carried out, on lines 35–36. This step consists of resetting the temperature and weight to their initial value. This allows the search algorithm to move to other parts of the search space with possibly higher quality solutions. After 10 reheating steps a diversification step is performed on lines 38–39. This step replaces the current assignment of x with a random permutation of hosts, effectively restarting the algorithm from a new non-deterministic starting point.

CP Slave Figure 5.5 shows the function that performs the actual moves of the CBLS master. It consists of the swap (line 2) and the CP slave (lines 4–38). The function of the slave algorithm is to determine whether the new solution satisfies the load-balancing constraint. Note that the slave does not influence the communication cost objective. Instead, it attempts to find an assignment of read and write quorums to hosts, given the current assignment of hosts to members, which satisfies the load-balancing constraint.

Since a CBLS algorithm is used for the master, it seems logical to also use a CBLS algorithm for the slave. However, initial testing showed a CP algorithm to be more effective. One of the main reasons for this is that many of the problems considered by the slave will be infeasible. A CP algorithm is better suited and more efficient for proving infeasibility in these cases. Indeed, a CBLS algorithm is unable to prove infeasibility and can merely conclude that after spending a certain amount of time trying to find a solution, no solutions have been found. Furthermore, if a solution does exist it is typically found in only a couple of milliseconds using the CP algorithm, a CBLS algorithm is unable

to significantly improve on this already very fast method. The CP model that is used for the slave is essentially the same as the second phase of the CP model of section 5.3.

Although solving the slave is typically quite fast, it is still costly to perform it each iteration. The conditions on line 4 therefore distinguishes two cases where it is not necessary to perform the slave. In these cases the earlier discussed estimation constraint can be used instead. The first condition determines whether the new solution has a lower communication cost than the best known feasible solution. Solutions with a *higher* communication cost can be useful to visit, as they can be part of a path towards a solution of a better quality. However, it is not necessary to determine their feasibility with absolute certainty. They do not make up the final solution in any case and are merely used as intermediate solutions. The second condition determines whether there is any possibility the load-balancing constraint can be satisfied, by comparing the upper and lower bounds on the loads. If the scaled minimum of the upper bounds on the load is smaller than the maximum of the lower bounds on the load, the load-balancing constraint is necessarily infeasible, leaving no need to perform the slave model. If either of the conditions are not met, the estimation constraint is used. In these cases line 37 will assign the violations variable the value of the estimation constraint incremented by one. This increment is necessary to ensure that no solution which satisfies the estimation constraint is automatically considered feasible. Remember that satisfaction of the estimation constraint does not guarantee satisfaction of the real constraint.

If both conditions are met, the CP slave model shown on lines 5–35 will be executed. The decision variables $readQ[h]$ and $writeQ[h]$, declared on lines 10–11, denote the chosen read and write quorum for host h. Line 12 declares the $load[m]$ variable, which denotes the load on configuration member m, given the current assignment of read and write quorums to hosts. The constraints on lines 15–18 require the chosen read and write quorum for each host to be among the quorums with minimal delay for that host. Lines 19–20 specify the communication load on m as the sum of the operation frequencies of each host for which m is a member of its assigned read and/or write quorum. Finally, line 22 states the actual load-balancing constraint.

The search procedure, shown on lines 24–27, considers the most "talkative" hosts first (by decreasing frequencies) and attempts to assign one of the remaining legal (minimal) quorums from its domain. When an assignment of read and write quorums that satisfies all constraints has been found the boolean *feasible* is set to true and the search is terminated. If a feasible solution was found, line 33 sets the violations variable to 0. When no feasible solution was found the violations variable is set to the estimation incremented by one, again, to ensure that no solution is wrongfully considered feasible.

The estimation is used to give an indication of the degree of violation of the load balancing constraint. This, in turn, can be used as a measure of distance from the current solution to a feasible solution. The estimation gives solutions, with the same communication costs, which are closer to a feasible solution a lower objective value than solutions which are farther. This guides the search towards high-quality *feasible* solutions. Another approach would be to, instead of using the estimation constraint, use a simple boolean to denote whether a solution satisfies the load-balancing constraint or not. This approach simplifies the model considerably, but lacks the guiding ability of the distance information. Both approaches were considered and evaluated. Their performance differences will be discussed in section 5.7.

```
1  void RAMBO::makeMove(int m, int n){
2      x[m] :=: x[n];
3
4      if ((O.value() < bestFeasible) && (max (m in M) loadMin[m] <= alpha * min (m in M) loadMax[m])) {
5          boolean feasible = false;
6
7          Solver<CP> cp ();
8          cp.limitFailures(1000);
9
10         var<CP>{int} readQ[H](cp, RQ);
11         var<CP>{int} writeQ[H](cp, WQ);
12         var<CP>{int} load[M](cp, 0..100000);
13
14         solve<cp> {
15             forall (h in H) {
16                 cp.post(readD[h, readQ[h]] == min(r in RQ) readD[h,r]);
17                 cp.post(writeD[h, writeQ[h]] == min(w in WQ) writeD[h,w]);
18             }
19             forall (m in M)
20                 cp.post (load[m] == sum(h in H)(f[h] * (readQC[readQ[h], m] + writeQC[writeQ[h], m])));
21
22             cp.post(max (m in M) load[m] <= alpha * min (m in M) load[m]);
23         } using {
24             forall (h in H : !readQ[h].bound() || !writeQ[h].bound()) by (-f[h]) {
25                 label(readQ[h]);
26                 label(writeQ[h]);
27             }
28
29             feasible = true;
30         }
31
32         if (feasible)
33             violations := 0;
34         else
35             violations := S.violations() + 1;
36     } else {
37         violations := S.violations() + 1;
38     }
39 }
```

Figure 5.5: CP Slave Search in COMET

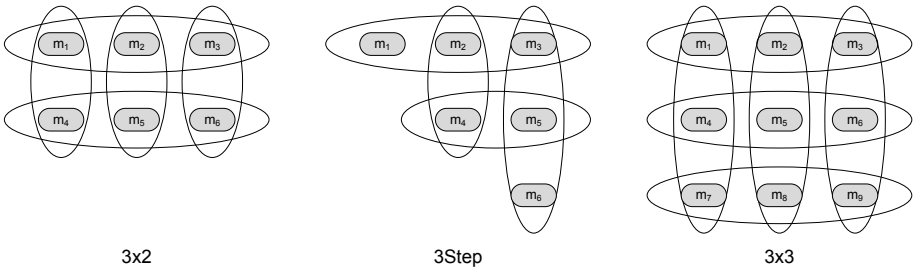


Figure 5.6: Quorum System Benchmarks `3x2`, `3Step`, and `3x3`

5.5 Parallel Composition

As was discussed in section 4.5, a hybrid that unifies the CP and CBLS models has potential benefits. The ability of the CBLS algorithm to deliver high-quality solutions quickly can be used to augment the CP algorithm, and increase its performance. The model that is presented here is a parallel composition of the CP model of section 5.3 and the CBLS/CP master-slave model of section 5.4. The key idea is to run both models concurrently and use the solutions found by the CBLS model to tighten the bound on the objective of the CP model, and thereby reduce the search space. Of course, there will only be a benefit if: 1) the CBLS model is able to located high-quality solution more quickly than the CP model, and 2) both models truly run in parallel.

The models are adapted to both run in separate threads and to communicate their progress to each other. In particular, the CBLS model notifies the CP model each time it finds a new solution. This communication is implemented in COMET through the use of events [16]. The instruction

```
1 c.tellNewSolution(new Solution(ls, MinimizeIntValue(O.value())));
```

is added to the CBLS model, to notify the CP model of a new solutions found by the CBLS model. And the snippet

```
1 whenever c@newSolution(Solution s)
2    if (s.getObjectiveValue().compare(cp.getObjective().getPrimalBound()) < 0)
3       cp.setPrimalBound(s.getObjectiveValue());
```

is added to the CP model to update the bound on the objective every time a new solution, which signifies an improvement over the current bound, is received. By tightening the bound the algorithm is likely able to discard parts of the search tree earlier than it otherwise would have been able to, thereby increasing its performance. Whenever the CP model terminates it will notify the CBLS model in a similar fashion, so that is can be terminated too.

5.6 Benchmarks

To test the performance of the models a variety of benchmarks is used. Each benchmark consists of the combination of an abstract quorum system and a network configuration. Six different network configurations and four quorum systems are used, representing common networks and quorums. This results in a total of 24 different benchmarks. Figure 5.6 shows the `3x2`, `3Step`, and `3x3` quorum system. The horizontal groups form the read quorums, while the vertical groups form the write quorums. The fourth quorum system, `Maj`, uses majority quorums [30]. This quorum system groups six members into four read quorums and four write quorums, each consisting of four members. The

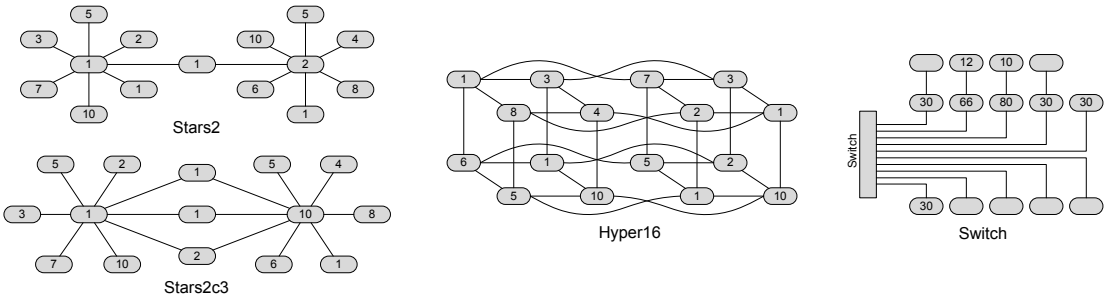


Figure 5.7: Network Configuration Benchmarks `Stars2`, `Stars2c3`, `Hyper16`, and `Switch`

		3x2			3Step			3x3			Maj		
Benchmark		Opt	T_{end}	T_{opt}	Opt	T_{end}	T_{opt}	Opt	T_{end}	T_{opt}	Opt	T_{end}	T_{opt}
Stars3	μ	261	0,98	0,41	285	0,37	0,09	284	42,12	10,95	340	5,30	0,03
	σ		0,18	0,29		0,03	0,06		5,37	10,71		0,12	0,01
Stars2	μ	303	5,70	2,76	284	0,57	0,30	316	536,50	92,69	374	20,76	1,35
	σ		9,71	9,67		0,24	0,21		203,53	197,01		3,62	3,71
Stars2c3	μ	238	1,16	0,21	239	1,52	1,37	268	1414,09	914,20	270	8,07	0,03
	σ		0,15	0,14		1,79	1,79		402,47	609,09		0,17	0,01
Line	μ	485	4,02	2,76	479	2,92	2,59	517	445,28	319,96	607	26,04	6,91
	σ		0,57	0,80		1,88	2,02		109,01	155,78		0,87	0,48
Hyper16	μ	246	9,87	5,04	256	2,13	0,85	249	508,28	226,04	305	66,94	3,37
	σ		1,24	2,27		0,16	0,60		82,37	153,78		0,89	0,24
Switch	μ	610	1,59	0,71	620	0,46	0,30	620	59,03	45,06	620	0,23	0,08
	σ		0,80	0,70		0,58	0,57		69,38	69,67		0,05	0,05

Table 5.1: Experimental Results for the CP Model with $\alpha = 2$

quorum systems `3x2`, `3Step`, and `Maj` each consist of 6 members, while the `3x3` quorum system consists of 9 members. The amount of read and write quorums varies, from the maximum of four each for the `Maj` quorum system, to the minimum of two each for `3Step`.

RAMBO is of course intended to be used in dynamic network setting. The network configurations used for the benchmarks are, however, "static", as they represent a certain network structure at the time of reconfiguration. The six different networks that are used for the benchmarks are: the `Stars3` network, shown in figure 5.1; the `Stars2`, `Stars2c3`, `Hyper16`, and `Switch` networks shown in figure 5.7; and the `Line` network, which consists of 15 hosts arranged in a single line (a bus). Figure 5.1 and 5.7 give the average frequency of read and write operations for each host in networks. The number of "hops" between a pair of hosts is used as the delay between those hosts.

5.7 Experimental Results

5.7.1 Constraint Programming Model

Table 5.1 reports the results for the CP model with COMET 1.1 (on a Core 2 @ 2.16 GHz) using a load balancing factor of 2 ($\alpha = 2$). The table provides two rows for each benchmark: the first reports averages and the second reports standard deviations. Columns are grouped by quorum system type. Within each group, column Opt gives the objective value of the optimal solution, T_{end} gives the time

		3x2				3Step				3x3				Maj			
Benchmark		T_{end}	T_{best}	$\#O$	$Best$	T_{end}	T_{best}	$\#O$	$Best$	T_{end}	T_{best}	$\#O$	$Best$	T_{end}	T_{best}	$\#O$	$Best$
Stars3	μ	10,34	0,54	50	261	8,21	1,63	50	285	10,84	2,80	46	284,3	20,81	0,34	50	340
	σ	0,05	0,39		0	0,10	1,40		0	0,11	2,49		1,1	0,20	0,23		0
Stars2	μ	9,43	1,02	50	303	7,58	0,43	50	284	11,55	2,05	50	316	18,72	0,44	50	374
	σ	0,22	1,16		0	0,11	0,41		0	1,60	1,64		0	0,61	0,61		0
Stars2c3	μ	8,80	0,10	50	238	7,14	0,34	50	239	96,26	32,88	50	268	18,71	0,31	50	270
	σ	0,03	0,05		0	0,24	0,29		0	6,49	28,35		0	0,42	0,43		0
Line	μ	11,14	0,56	50	485	9,14	1,05	50	479	11,55	0,89	50	517	22,35	0,65	50	607
	σ	0,03	0,38		0	0,05	1,42		0	0,06	0,88		0	0,12	0,65		0
Hyper16	μ	12,83	2,90	38	246,2	9,46	2,45	36	257,0	14,03	3,85	0	252,2	24,51	7,53	42	305,2
	σ	0,04	2,59		0,4	0,04	2,40		1,8	0,05	3,96		1,3	0,06	6,20		0,4
Switch	μ	8,91	0,08	50	610	6,13	0,04	50	620	8,26	0,12	50	620	17,34	0,14	50	620
	σ	0,16	0,07		0	0,02	0,02		0	0,13	0,11		0	0,09	0,09		0

Table 5.2: Experimental Results for the CBLS Model with the Estimation Constraint and $\alpha = 2$

in seconds to find the optimum and prove optimality, and column T_{opt} reports the time in seconds to find the optimum. The results are the average and standard deviation over 50 runs.

The results show that in general the `3Step` quorum system is the easiest to solve, followed by `3x2`, `Maj`, and then `3x3`. The `3x3` quorum system in particular seem to be significantly harder to solve than the others. This is not entirely surprising as this quorum system has more members. In addition to the results shown, results were obtained for load factors $\alpha = 3$ and $\alpha = 4$. In general these results are similar, although in a few cases they differ significantly. In particular the runtime of benchmark `Stars2c3` with the `3x3` quorum system is dramatically reduced when a less restrictive load balancing factor is used.

It is worth to note the variation in the time it takes to find the optimal solutions. With quorum system `Maj` the optimum is found quickly, sometimes as soon as one percent into the run. In contrast, benchmarks `Stars2c3` and `Line` with `3Step` do not find the optimum until the very end. The standard deviations of the various benchmarks show further variations. While the `Maj` quorum system induces small deviations, the `3x3` quorum system exhibits deviations that are often larger than the averages. Closer examination of the runs reveals that these high standard deviations are due to a few outliers with a significantly longer runtime.

5.7.2 Hybrid CBLS/CP Master-Slave Algorithm

Table 5.2 reports the results for the CBLS/CP master-slave model with the estimation constraint and a load factor of 2 ($\alpha = 2$), using COMET 1.1 (on a Core 2 @ 2.16 GHz). Column T_{end} reports the total runtime in seconds, while T_{best} gives the time to find the best solution. The $\#O$ column indicates how often the best solution was found (out of 50 runs), and the $Best$ column gives the quality of the best solution that was found. All values are the average and standard deviation over 50 runs.

The results show that optimal solutions are found very reliable. Indeed, when the `Hyper16` network configuration and the `3x3` quorum system are left out, the optimal solution is found on every run of every benchmark. The `Hyper16` network and `3x3` quorum system clearly are the hardest to solve. For the benchmark which is a combination of the two, the algorithm is unable to find the optimum in any of the runs. However, even on this benchmark the average best solution is only one percent away from the optimum, with a standard deviation of only 1,3.

The standard deviations of the total runtime and time to best solution are consistently very low. Furthermore, on average the time to the best solution is relatively short compared to the total run-

		3x2				3Step				3x3				Maj			
Benchmark		T_{end}	T_{best}	$\#O$	$Best$	T_{end}	T_{best}	$\#O$	$Best$	T_{end}	T_{best}	$\#O$	$Best$	T_{end}	T_{best}	$\#O$	$Best$
Stars3	μ	10,02	0,74	50	261	8,18	1,43	50	285	10,48	1,61	50	284	19,99	0,36	50	340
	σ	0,04	0,89		0	0,10	1,00		0	0,19	1,44		0	0,27	0,30		0
Stars2	μ	9,37	2,24	50	303	7,37	0,38	50	284	11,03	1,79	50	316	17,85	0,29	50	374
	σ	0,23	1,72		0	0,11	0,29		0	1,95	1,98		0	0,56	0,58		0
Stars2c3	μ	8,74	0,11	50	238	7,07	0,32	50	239	33,12	17,10	13	269,8	18,71	0,25	50	270
	σ	0,02	0,06		0	0,22	0,26		0	5,93	11,58		4,2	0,43	0,43		0
Line	μ	10,83	1,24	50	485	8,52	2,23	48	479,2	11,25	5,57	16	523,8	22,53	0,70	50	607
	σ	0,03	1,21		0	0,06	2,13		0,8	0,07	3,69		7,8	0,13	0,48		0
Hyper16	μ	12,46	2,97	48	246,0	9,69	2,80	46	256,2	13,35	2,98	3	251,6	23,61	8,37	48	305,0
	σ	0,05	3,28		0,2	0,05	2,56		0,9	0,06	3,13		1,2	0,06	6,17		0,2
Switch	μ	16,26	0,10	50	610	7,26	0,03	50	620	9,95	0,09	50	620	18,66	0,12	50	620
	σ	0,63	0,06		0	0,03	0,02		0	0,08	0,08		0	0,09	0,06		0

Table 5.3: Experimental Results for the CBLS Model without the Estimation Constraint and $\alpha = 2$

time. On many runs the best solution is found as little as 10% into the run. All in all this indicates a very robust model. Results were also obtained for load factors $\alpha = 3$ and $\alpha = 4$. These results were very similar to the results shown in table 5.2, with only minor variations in the frequency with which the optimal solution was found. Some benchmarks peform slightly better when a less restrictive load balancing constraint is used, while others perform slightly worse.

One particularly interesting benchmark is `Stars2c3` with the `3x3` quorum system. This benchmark has a total runtime that is almost five times as long as any other benchmark. Closer inspection reveals that the longer runtime is due to a relative large number of invocations of the slave search. In the other benchmarks most of the slave searches are eliminated through the conditions described in section 5.4. For this benchmark, however, this strategy is not as effective. In particular, the benchmark is characterized by a large number of *infeasible* solutions with a lower communication cost than the optimum solution. The slave search can not be eliminated for these solutions.

The `Stars2c3` benchmark with `3x3` quorum system also serves as a good example to illustrate the difference between the model that uses an estimation of the violations of the load balancing constraint and the model that uses a boolean. Table 5.3 shows the results of the model that uses a boolean to denote whether the load balancing constraint is violated or not, instead of the estimation of the degree of violation of the load balancing constraint. Most results are similar for the two models, with some benchmarks performing slightly better in one, while others perform slightly better in the other. The only benchmark where there is a significant difference is `Stars2c3` with the `3x3` quorum system. This benchmark is almost three times faster using the model *without* the estimation. This might suggest that this model is more effective, however, closer inspection reveals that the benchmark performs better due to a smaller number of invocations of the slave search. This, in turn, is due to the fact that the model without the estimation visits fewer solutions with a communication cost lower than the optimum. Although the model *with* the estimation has a longer runtime, it spends more time exploring areas of the search space with higher quality solutions. Unfortunately these solutions are infeasible, however, it does indicate a more effective search algorithm.

Figure 5.8 shows the evolution of the objective over time for the `Stars2c3` benchmark with the `3x3` quorum system. The left side of the figure shows the objective over time for the CP model, while the right side shows the same for the CBLS/CP master-slave model. Note the difference in time scale for both graphs. Each point in the graph represents a solution found by the algorithm, which is an improvement over the previously found best solution. Each graph combines 50 runs of the algorithm. The black points represent the best solution found during a run, while the gray points

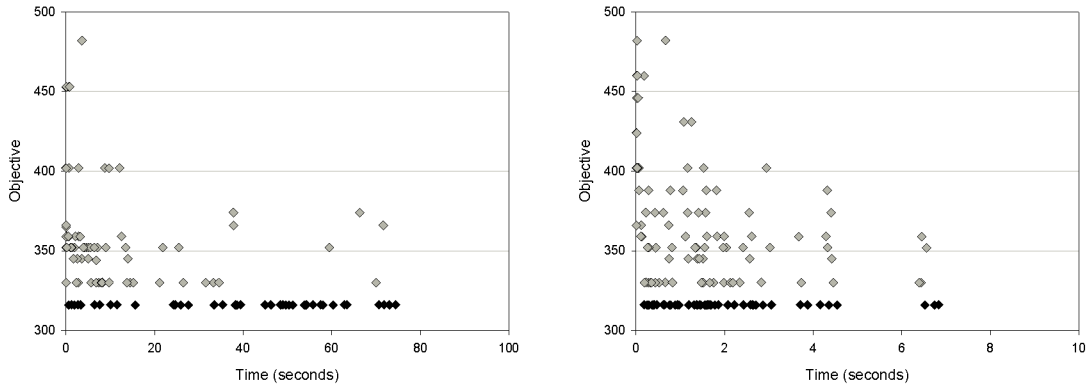


Figure 5.8: Evolution of the objective over time of the `Stars2` benchmark with the `3x3` quorum system using the CP model (left) and CBLS model (right)

represent intermediary solutions found during the run. Some ten outliers have been excluded from the CP graph to allow a clearer picture of the more relevant areas of the graph.

The graph shows that the CBLS model is able to find high-quality solutions more quickly than the CP model. Although distribution of points in the section < 10*secs* is not significantly different, the CBLS model does not suffer from the same outliers as the CP model does. This indicates a potential benefit for a hybrid model, which combines both the CP and CBLS models. Such a hybrid can potentially eliminate the outliers by delivering high-quality solutions to the CP model more quickly and allowing it to discard many solutions.

5.7.3 Parallel Composition

Tables 5.4 and 5.5 report the results for the parallel composition and compare them to the CP model. The results were obtained using COMET 1.1 (on a Core 2 @ 2.16 GHz). The CBLS component of the composite model uses the estimation constraint. The tables report the total runtime and time to optimum for each benchmark, for both the CP model and the composite model. All values are the average and standard deviation over 50 runs.

Table 5.4 shows the results for the benchmarks with the `3x2` and `3Step` quorum systems. These benchmarks show an average total runtime which is very similar for both the CP and the composite model. In some instances the composite is slightly faster, while in others the CP model performs slightly better. The composite shows more of an improvement in the time to find the optimum. The biggest improvement is for the `Line` benchmark with the `3x2` quorum system, here the average time to optimum is reduced from 2,76 to 0,57. Another notable improvement is the standard deviation of benchmark `Stars2` with the `3x2` quorum system, the standard deviation for both the total time and time to optimum is reduced from over 9 seconds to under 1. Most of the benchmarks already show reasonably low times, leaving not much room for improvement.

The results for the benchmarks with quorum systems `3x3` and `Maj` (table 5.5) show more significant differences between the CP and composite model. The composite clearly benefits from its local search component as it consistently delivers the optimum very early on and quite reliably. The improvement in the time to optimum is most significant for the `Stars2c3` benchmark with the `3x3` quorum system, here it is reduced from 914 seconds to 37 seconds. The time to complete the optimality proof offers a mixed set of results. For some instances (e.g., instances based on `Maj` quorums),

		3x2				3Step			
		CP		Composite		CP		Composite	
Benchmark		T_{end}	T_{best}	T_{end}	T_{best}	T_{end}	T_{best}	T_{end}	T_{best}
Stars3	μ	0,98	0,41	1,06	0,46	0,37	0,09	0,46	0,17
	σ	0,18	0,29	0,16	0,21	0,03	0,06	0,05	0,10
Stars2	μ	5,70	2,76	3,83	1,10	0,57	0,30	0,50	0,23
	σ	9,71	9,67	0,33	0,69	0,24	0,21	0,18	0,17
Stars2c3	μ	1,16	0,21	1,20	0,11	1,52	1,37	0,48	0,30
	σ	0,15	0,14	0,19	0,05	1,79	1,79	0,29	0,29
Line	μ	4,02	2,76	3,39	0,57	2,92	2,59	1,17	0,74
	σ	0,57	0,80	0,25	0,42	1,88	2,02	0,39	0,43
Hyper16	μ	9,87	5,04	10,17	3,80	2,13	0,85	2,22	0,83
	σ	1,24	2,27	1,00	3,05	0,16	0,60	0,14	0,63
Switch	μ	1,59	0,71	1,20	0,09	0,46	0,30	0,22	0,04
	σ	0,80	0,70	0,29	0,07	0,58	0,57	0,07	0,02

Table 5.4: Experimental Results for the CP and Composite model with $\alpha = 2$

		3x3				Maj			
		CP		Composite		CP		Composite	
Benchmark		T_{end}	T_{best}	T_{end}	T_{best}	T_{end}	T_{best}	T_{end}	T_{best}
Stars3	μ	42,12	10,95	39,78	3,74	5,30	0,03	6,04	0,40
	σ	5,37	10,71	4,09	5,50	0,12	0,01	0,23	0,24
Stars2	μ	536,50	92,69	494,37	2,14	20,76	1,35	21,37	0,38
	σ	203,53	197,01	56,45	1,67	3,62	3,71	0,98	0,48
Stars2c3	μ	1414,09	914,20	778,96	37,00	8,07	0,03	9,13	0,26
	σ	402,47	609,09	55,87	63,11	0,17	0,01	0,40	0,28
Line	μ	445,28	319,96	247,73	1,07	26,04	6,91	27,26	0,76
	σ	109,01	155,78	18,80	0,87	0,87	0,48	1,11	0,42
Hyper16	μ	508,28	226,04	513,29	214,23	66,94	3,37	71,85	7,11
	σ	82,37	153,78	58,08	151,45	0,89	0,24	1,75	6,72
Switch	μ	59,03	45,06	14,96	0,10	0,23	0,08	0,29	0,12
	σ	69,38	69,67	2,99	0,10	0,05	0,05	0,07	0,07

Table 5.5: Experimental Results for the CP and Composite Model with $\alpha = 2$

there are no benefits to speak of. For others, the availability of the optimum early on translates into a shorter optimality proof and decreased standard deviation. The most dramatic instances in this respect are, perhaps, `Switch`, `Line`, and `Stars2c3` with `3x3` quorum systems. The composite model further delivers consistently lower standard deviations.

6 Conclusion

The previous chapters have described the work that has been done on the ESDS and RAMBO deployment problems, and the results that were obtained. It is useful to review these results and place them in the broader context of the more general deployment problem, and reflect on what has been accomplished as well as the specific contributions of this work.

6.1 Results

6.1.1 Eventually-Serializable Data Services

Previous work on the eventually-serializable data services deployment problem already resulted in solutions for this problem in the form of mixed integer programming (MIP) and constraint programming (CP) methods. The constraint programming method was shown to be vastly superior to the mixed integer programming method for this particular problem. Many instances could be solved in seconds, rather than the hours it took the MIP. However, larger instances of the problem, and in particular those involving bandwidth constraints, still took a considerable amount of time to solve. One particular instance took well over 30 minutes using constraint programming, leaving a lot of room for improvement.

The constraint-based local search method, that was developed as part of this research project, delivers excellent solutions very quickly. For the easier instances it is able to find the optimum almost every time. On the harder instances it delivers the optimum less reliably, and on the hardest instance it only finds to optimum very sporadically. However, even in these cases the algorithm still delivers very high quality solutions, and it does so much faster that the CP method. Of course it is typical for local search to not be able to always find the optimal solution. It is an incomplete method, and thus there are no guarantees of finding optimal solutions, or any solutions at all for that matter. The trade-off, however, is that efficient local search algorithms are very often able to locate high-quality solutions very fast. For the ESDS deployment problem this is certainly the case. The graphs of the evolution of the objective during the course of the search are especially telling in this regard. They clearly show that the constraint-based local search method is able to find high-quality solutions much faster than the constraint programming method.

Because local search can not prove optimality of a solution it does not have any “natural” exit criteria. Instead, it is allowed to run for a predefined amount of time, or iterations. In the case of the ESDS deployment problem the CBLS algorithm was allowed to run for a certain amount of iterations, which corresponded to somewhere between 5 and 10 seconds for the various benchmarks. This means the CBLS algorithm spends more time on the *easy* instances than it should, but it is a considerable improvement over the up-to 30 minutes of run-time of the constraint programming method for the hardest instances. Of course, this comes at the price of a potentially sub-optimal solution. However, a user might not always need an optimal solution. In many situations it will be

much more desirable to have a *good* solution very fast. The constraint-based local search method is clearly valuable in this regard.

The co-location preprocessing technique that was developed for the constraint-based local search method is advantageous to this method as it increases its performance and makes it more robust. However, it is perhaps even more advantageous to the constraint programming method. This slight change in the problem representation results in a reduction of the run-time of the constraint programming algorithm on the hardest benchmark from well over 30 minutes to a little over 6 minutes. This closes the gap between the CBLS and CP methods considerably, although it remains significant, especially on the harder instances.

The hybrid models offer a mixed set of results. The sequential hybrid performs worse than the constraint programming method on the easier benchmarks, and slightly better on the harder benchmarks. The parallel performs better in almost every regard, although the difference is not that great. The hybrid models in a sense combine the advantages of both the local search and constraint programming methods. Their main advantage over the CP method is a faster time to find "good" and optimal solutions, while the main advantage over CBLS is their ability to prove optimality of solutions. On the hardest benchmarks the parallel hybrid shows, on average, an improvement of about 10% in total run-time over the CP method. The hybrid models can be particularly useful when a user wants a quick solution to work with, while the algorithm continues to run in order to prove optimality of the solution, or possibly improve on it.

All-in-all the developed constraint-based local search and hybrid solutions add new and useful methods to solve the eventually-serializable data services deployment problem. They are perhaps not superior to the constrain programming method in every way, but they provide different advantages and give the user the ability to chose from a wider toolset and strike trade-offs between different aspects.

6.1.2 Reconfigurable Atomic Memory for Basic Objects

The RAMBO deployment problem is particularly well suited for a local search based solution, due to its online nature and the tight time constraints in which it must be solved. The existing constraint programming solution for this problem performed well on the easier benchmarks, solving them in mere seconds. The harder benchmarks, however, proved more challenging, with the hardest instance taking well over 20 minutes to solve and prove optimality.

Reconfiguring to a new quorum configuration can be done while RAMBO is running, the service does not have to be interrupted for it. The process of proposing a new configuration and reaching consensus on it among the participants of the services takes time in and of itself. Because of these factors there is a time window of some seconds available to compute a new configuration. Although there is no *hard* time limit for computing the configuration it should ideally not take much more than 10 seconds. This means the constraint programming method is not practically usable for some of the harder instances.

The constraint-based local search solution (CBLS/CP master-slave) that was developed for this problem performs very well. As usual with local search the algorithm is allowed to run for a predefined number of iterations. In this case corresponding to approximately 10 seconds of run-time. The optimal solution is found very reliably, and very early on during the search. For the majority of the benchmarks the best solution is found (on average) in under a second. On only two benchmarks it takes more than five seconds (on average) before the best solution is located. This indicates that the number of iterations, and subsequently the run-time, could perhaps be reduced further. The results furthermore show the CBLS method to be very robust. On only on the hardest instances the optimal solution is not found on every run of the algorithm, but even on these instances a very high-quality solution is found very reliably and fast. Even on the hardest instance the average best solution that is found is less than one percent away from the optimum, and it is found (on average) in under four

seconds. Only one instance of the problem takes considerably longer to solve, as was discussed in section 5.7 this is due to some specifics of this instance. Studying this instance more in depth could perhaps reveal some characteristics that could be exploited to bring the performance more in line with the rest.

The developed composite model does perhaps not deliver all of the desired performance gains. The total run-time of the composite is generally similar to that of the constraint programming method. Two of the hardest benchmarks have their average run-time halved, but for most benchmarks the difference is negligible or even slightly in favor of the CP method. The time-to-optimum, however, is dramatically reduced for almost all benchmarks. This shows the composite greatly benefits from the CBLS component, by being able to find optimal solutions faster, but that it can not sufficiently use this to shorten the optimality proof. However, the composite is still a very usable solution to the RAMBO reconfiguration problem. For instance, it can be used to run within a certain time frame. Even with a time frame as short as ten seconds it will: very often find the optimal solution; on the easier instances prove optimality; and only in a very few cases return a sub-optimal solution, which however will still be of very high-quality.

In summary, the developed local-search based method is particularly appealing for the RAMBO reconfiguration/deployment problem, due to its ability to deliver high-quality solutions fast. The composite model which combines this new solution with an already existing solution also has certain advantages. The various methods that now exist strike different trade-offs in their ability to deliver high-quality/optimal solution fast, and the ability to prove optimality of these solutions. This gives the user more freedom to chose the tools which are best for any particular situation.

6.2 Discussion

The results discussed in the previous section show that significant progress has been made on these particular problems. New methods have been developed to solve them, which strike different trade-offs in time that is spend and the "quality" of the solution that is obtained. The developed methods are useful as solutions for these particular deployment problems in itself. However, the problems themselves were also chosen because they represent more general deployment problems.

The ESDS deployment problem is especially appealing in this regard. The terms in which the problem is stated are very general, and can be applied to a great variety of deployment problems. This characteristic has already been exploited. The methods that were developed for the ESDS deployment problem have been generalized and added to the Tempo Toolkit [19] as deployment optimization schemes [10]. Specific annotations were added to the Tempo language that allow the formal specification of a deployment, along with the formal specification of a distributed system. Based on these specifications the toolkit is able to automatically generate an optimization model (and search) in the COMET language. This model then only needs to be run in COMET, without any further input from the user, to obtain a solution. The generated optimization models are based on the methods developed for the ESDS deployment problem. The user is able to specify whether constraint programming, constraint-based local search, or either a sequential or parallel hybrid is to be used.

The RAMBO reconfiguration problem lends itself less to generalization. Although quorum systems are fairly common, the way in which they are used in RAMBO is rather specific. Furthermore, the objective function is unlikely to apply to other deployment problems in exactly the same sense. This, however, does not mean the results have no relevance outside of this single application domain. Up and foremost the results show that these kind of online problems *can* be solved using these techniques. Furthermore, even though the exact methods that have been developed for the RAMBO deployment problem might not be applicable to other deployment problems, many elements of them most likely *will* be. The specifics of the deployment problems might differ, but many of the underlying issues will be similar, allowing for certain parts of the methods to be reused. No actual

RAMBO implementation that uses the reconfiguration optimization methods has been developed so far. However, plans in this direction do exist.

In addition to the progress made on the deployment problem there is also some significance in this research for the field of constraint programming. Constraint programming is a developing field, and as such is constantly looking for new applications to test and extend its repertoire of techniques. Although the various techniques that were used in this thesis are not new, they are combined in interesting and novel ways. Hybridization of different methods, for instance, is not a new concept, but few real-world applications of it exist. Therefore it is valuable to test these concepts on real problems and examine their performance. The results of this research provides relevant feedback to the constraint programming community in this regard.

6.3 Future Work

This thesis has investigated the problem of optimally deploying distributed systems. Significant progress has been made, but there is clearly more work to be done. This section will address some of the directions future research can take.

Most research aims at eventually having a real world application, and this is certainly the case here. The implementation of the deployment optimization methods in the Tempo Toolkit is a very solid first step in this regard. Users are able to formally specify a system and deployment, and have the toolkit automatically find an optimal deployment based on these specifications. This makes the techniques much more accessible to non-expert users. However, more work is still needed. The RAMBO reconfiguration optimization methods have not yet been applied in practice. A natural extension of the work that has been done in on this problem is the development of an actual RAMBO implementation that uses the developed optimization methods. Achieving this is not too complex as all the needed elements are available, they merely need to be brought together. Plans in this direction exist, so this may be realized in the future.

Another important element is of course the applicability of the developed methods on other deployment problems. Although there is confidence that the methods which have been added to the Tempo Toolkit are applicable to many other deployment problems, this remains somewhat untested. The degree to which other deployment problems can be stated in the terms provided by the annotations, and the efficiency with which the developed optimization methods can solve these problems, needs further examination. More rigorous testing on a wider set of problems is needed. Furthermore, the results on the RAMBO deployment problem show that this type of *online* optimization problem can be solved using these techniques. However, to what degree these results can be generalized to a broader set of problems is a question that needs further investigation. Investigating the extend to which all of this is possible is an interesting direction for future research.

A major limit on the developed methods is the scale of the distributed systems on which they are applied. Many real world system will be much larger and consist of many more hosts and components than the systems used for the benchmarks in this thesis. Currently the methods do not scale well enough to allow for such networks. As the number of hosts and components increases to a certain size the computation time will explode. A challenging and interesting direction for future research is to investigate the possibilities for improving scalability, including decomposition schemes. Indeed, an option might be to divide larger system into smaller subsystems which can be deployed optimally. To what extend this approach can lead to optimality of the whole system remains to be investigated.

Bibliography

[1] The comet programming language and system. `http://www.comet-online.org/`.

[2] Emile Aarts and Jan K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[3] Roman Bartak. Constraint programming: In pursuit of the holy grail. In *Proceedings of the 8th Annual Conference of Doctoral Students (WDS'99)*, pages 555–564, 1999.

[4] M. Cecilia Bastarrica. *Architectural Specification and Optimal Deployment of Distributed Systems*. PhD thesis, The University of Connecticut, 2000.

[5] M. Cecilia Bastarrica, Rodrigo E. Caballero, Steven A. Demurjian, and Alexander A. Shvartsman. Two optimization techniques for component-based systems deployment. In *Proceedings of the 13th International Conference on Software Engineering & Knowledge Engineering (SEKE'01)*, pages 153–162, 2001.

[6] M. Cecilia Bastarrica, Steven A. Demurjian, and Alexander A. Shvartsman. Software architectural specification for optimal object distribution. In *Proceedings of the 18th International Conference of the Chilean Computer Science Society (SCCC'98)*, pages 25–31, 1998.

[7] M. Cecilia Bastarrica, Alexander A. Shvartsman, and Steven A. Demurjian. A binary integer programming model for optimal object distribution. In *Proceedings of the 2nd International Conference on Principles Of Distributed Systems (OPODIS'98)*, pages 211–226, 1998.

[8] Roberto Battiti and Giampietro Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6(2):126–140, 1994.

[9] J. F. Benders. Partitioning procedures for solving mixed variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.

[10] Carleton Coffrin, Laurent D. Michel, Alexander A. Shvartsman, Elaine L. Sonderegger, and Pascal Van Hentenryck. Optimizing network deployment of formally-specified distributed systems. In *Proceedings of the 18th International Conference on Software Engineering and Data Engineering (SEDE'09)*, pages 230–237, 2009.

[11] Thomas Emden-Weinert and Mark Proksch. Best practice simulated annealing for the airline crew scheduling problem. *Journal of Heuristics*, 5(4):419–436, 1999.

[12] Alan Fekete, David Gupta, Victor Luchangco, Nancy A. Lynch, and Alexander A. Shvartsman. Eventually-serializable data services. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 300–309, 1996.

[13] Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. Rambo ii: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN'03)*, pages 259–268, 2003.

[14] Fred Glover and Manuel Laguna. Tabu search. In *Modern heuristic techniques for combinatorial problems*, pages 70–150. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[15] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[16] Pascal Van Hentenryck and Laurent D. Michel. Control abstractions for local search. *Constraints*, 10(2):137–157, 2005.

[17] Dilsun K. Kaynar, Nancy A. Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata*. Synthesis Lectures in Computer Science. Morgan & Claypool Publishers, 2006.

[18] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[19] Nancy A. Lynch, Stephen J. Garland, Dilsun K. Kaynar, Laurent D. Michel, and Alexander A. Shvartsman. The tempo language user guide and reference manual, 2008.

[20] Nancy A. Lynch and Alexander A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Conference on Distributed Computing (DISC'02)*, pages 173–190, 2002.

[21] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[22] Laurent Michel, Alexander A. Shvartsman, Elaine L. Sonderegger, and Pascal Van Hentenryck. Optimal deployment of eventually-serializable data services. In *Proceedings of the 5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'08)*, pages 188–202, 2008.

[23] Laurent D. Michel and Pascal Van Hentenryck. A constraint-based architecture for local search. *ACM SIGPLAN Notices*, 37(11):83–100, 2002.

[24] Laurent D. Michel and Pascal Van Hentenryck. Constraint languages for combinatorial optimization. In *Tutorials on Emerging Methodologies and Applications in Operations Research*. Springer, New York, NY, USA, 2005.

[25] Laurent D. Michel, Pascal Van Hentenryck, Elaine L. Sonderegger, Alexander A. Shvartsman, and Martijn Moraal. Bandwidth-limited optimal deployment of eventually-serializable data services. In *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09)*, pages 193–207, 2009.

[26] Laurent D. Michel, Martijn Moraal, Alexander A. Shvartsman, Elaine L. Sonderegger, and Pascal Van Hentenryck. Online selection of quorum systems for rambo reconfiguration. In *Proceedings of the 15th International Conference On Principles and Practice of Constraint Programming (CP'09)*, pages 88–103, 2009.

[27] Laurent D. Michel, Alexander A. Shvartsman, Elaine L. Sonderegger, and Pascal Van Hentenryck. Optimal deployment of eventually-serializable data services. *Annals of Operations Resarch*, 2008. Extended version of the CPAIOR'08 paper (accepted).

[28] Francesca Rossi, Peter Van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier Science Inc., New York, NY, USA, 2006.

[29] Barbara M. Smith. Sets of symmetry breaking constraints. In *Proceedings of the 5th International Workshop on Symmetry and Constraint Satisfaction Problems (SymCon'05)*, 2005.

[30] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[31] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press., London, 1993.

[32] Chris Voudouris and Edward Tsang. Guided local search. Technical Report CSM-247, Department of Computer Science, University of Essex, 1995.

[33] Ling-Yun Wu, Xiang-Sun Zhang, and Ju-Liang Zhang. Capacitated facility location problem with general setup cost. *Computers and Operations Research*, 33(5):1226–1241, 2006.