Scheduling Outages in Distributed Environments

Anthony Butler[1], Hema Sharda[1], and David Taniar[2]

[1] Department of Electrical Engineering and Computer System, RMIT, Melbourne, Australia,
{s9814033,hemas}@rmit.edu.au
[2] School of Business Systems, Monash University, Melbourne, Australia,
David.Taniar@infotech.monash.edu.au

Abstract. This paper focuses on the problem of scheduling outages to computer systems in complex distributed environments. The interconnected nature of these systems makes scheduling global change very difficult and time consuming, with a high degree of error. In addressing this problem, we describe the constraints on successful outage scheduling. Using these constraints, it describes a solution using Constraint Logic Programming, which is able to deliver rapid schedules for complex architectures. The developed approach is then applied and tested to several real world problems to ascertain its effectiveness and performance.

1 Introduction

A significant challenge faced by large enterprises relates to the scheduling of upgrade activities and the outages associated with them on hosts within a large, *distributed environment*. Complex distributed environments consist of many nodes that are interconnected and dependent upon one another. Coulouris et al [1] defined such environments as "one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages." These interconnections can be transactional dependencies, physical connections, or logical dependencies. With the world having moved away from the centralised model of mainframe computing to a distributed model, there are now new challenges to be addressed in the maintenance of these environments. These challenges were not present in the old mainframe architectures of a single centralised host and many dumb terminals running from it.

Amongst the challenges of managing a complex distributed environment is the challenge of managing the upgrade of components in such a way as to minimise downtime and ensure that all the many dependencies between systems are maintained. The challenge of scheduling these activities is compounded by the need to take into account the various dependencies as well as resourcing constraints. Any schedule is constrained by the availability of the human resources with the skills and availability to perform the necessary upgrades.

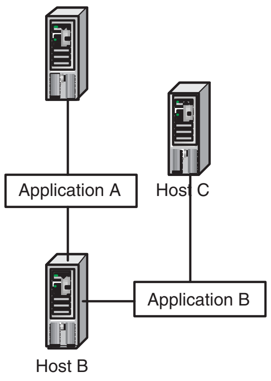

Fig. 1. Example Cluster

To attempt to solve these problems manually has proven to be a very time consuming and error-prone process. Furthermore, it is not scalable. As the number of hosts increase, so does the complexity, level of difficulty, and risk of error. This paper presents a solution to this complex problem by applying *Constraint Logic Programming* (CLP).

2 Migration Scheduling Problem: Background

Software migration is the managed upgrade or deployment of software to an organisations infrastructure – either in its entirety or a large subset. Some of these applications may have hardware dependencies between them, which makes scheduling difficult. A shared disk between two hosts, or multiple applications installed on a single host means that host-wide changes affect different applications, which in turn affect their own dependencies.

We defined the problem as "scheduling" and not "timetabling" or "planning". There has been considerable confusion amongst people outside of the scheduling community, as to the distinction between scheduling and planning. In short, planning is the process that decides what to do, whereas scheduling is the process which decides how and when to do the tasks specified within the plan [2]. Scheduling also involves resource allocation, wherein each activity is assigned to a resource(s). This is an important distinction. That said, there is still a planning component to the software migration effort. This planning consists of the determination is made as to what needs to be done, i.e. which software elements need to be upgraded and which hosts are affected.

3 Problem Definition

A detailed analysis of the problem domain was conducted in order to ascertain what constraints and dependencies existed that made the scheduling of outages in distributed environments difficult.

3.1 Using Clusters to Manage Complexity

In order to simplify the handling of a large number of hosts, it is useful to logically group hosts into clusters. The dependencies that bind hosts together into a cluster are: multiple applications on a common host, hosts sharing a common storage device, and up-stream and downstream transactional dependencies.

In Figure 1, Host A, Host B, and Host C are clustered because Host A and Host B share the same application and Host B and Host C share the same application.

The usefulness of clusters to scheduling is that it enables the scheduler to divide a large number of infrastructure components into human manageable chunks that can be dealt with in isolation. It is an example of the well-known principle of “divide and conquer”.

3.2 Environmental Constraints

Based on discussions with business managers at the telecommunications company, it was found that one of the most important considerations in scheduling software migrations is need to maintain required levels of uptime for their systems.

As a general rule, no organization should perform an upgrade to its production infrastructure or to production instances of its applications, until that same upgrade has been applied and tested against test infrastructure. Likewise, the development environment should be upgraded before test, whilst keeping in mind the need to maintain at least some test and development environments on the old versions of the software in order to be able to continue to support the existing production environment.

3.3 Summary of Rules and Constraints

Examining the dependencies between hosts and applications, we determined the following set of scheduling rules that need to be applied:

1 For a given application, hosts must be migrated in the order of development, test, then production. Additional environments, such as QA or Training, are inserted between test and production.

2 Hosts that share storage migrate at the same time if multiple hosts share storage, and that storage is used for bootdisk, then all hosts must be migrated together. If multiple hosts share storage, and that storage is not used for bootdisk, then the hosts should be scheduled to migrate together, though it is not necessary.

3 Hosts that send data, but do not receive any from the target, are unidirectional. Therefore, the sending system can be migrated without consideration of the recipients.

4 Hosts that receive data, but do not send any to the target, are unidirectional. If an outage of this data link would bring the host(s) down, then they should be migrated together.

5 Hosts that send and receive data from a host, should be scheduled to be migrated at the same time.
6 Hosts that are failover or backup for other Hosts. If it is necessary that a given host is failover or backup for another host, and it is necessary to maintain uptime of the application, then the two hosts must be scheduled to be migrated at different times.
7 Hosts that have infrastructure-level dependencies on other hosts, should be migrated at the same time as that other host, if possible.
8 Each machine has a system administrator who performs the migration. A system administrator can only work on one machine at any given time.

4 Methodology

There are many approaches to solving complex scheduling problems. Whilst they vary greatly in their detail, efficiency and effectiveness, they basically fall into one of two categories: *Artificial Intelligence* (AI) or *Operations Research* (OR) methods.

In distinguishing between them, OR methods tend to be based on mathematical models that aim to achieve high levels of efficiency in their models. OR methods also tended to focus on problems that could be expressed in purely mathematical terms. Whereas, AI approaches attempted to solve scheduling problems with more general problem solving paradigms. In summarising the benefits of each type of approach, the distinguishing quality between the two was the efficiency of OR methods versus the general applicability of AI methods.

In the OR domain, scheduling problems have typically been approached through dynamic programming, branch and bound algorithms, simulated annealing, and heuristic searches. Research in the Artificial Intelligence domain has centered around the application of genetic algorithms and constraint logic programming (CLP); with the latter being the most significant.

4.1 Constraint Logic Programming

Constraint Logic Programming (CLP) represents one of the most exciting subdomains within AI research. As Freuder [3] opined, "Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."

A constraint was defined by Bartak[4] as a logical relation between several unknowns (or variables), each taking a value within a given domain. The constraint thus restricts the range of possible values that a variable can take. Humans use constraints constantly. For instance, we may say to someone that we will visit them between 10am and 11am. This is an example of a constraint. The simplest form of constraint in CLP is what is known as a finite-domain variable. A finite-domain is a set of values, and a finite-domain variable is a variable that is constrained to only take on values from the defined set.

```
Start::1..10
Name::[Fred,Albert,John]
alldifferent([X1,X2,X3])
```

The above example requires that Start must be a value between 1 and 10; that Name must be one of either Fred, Albert or John; and that X1, X2 and X3 are unique.

The solution to a constraint problem is therefore when there has been an assignment of values to all of the variables consistent with the problem constraints. A constraint problem can have many correct answers. For instance, the constraint $X < Y$, can be solved by any value of X and Y, provided X is less than Y. Therefore, X=1, X=2 and X=10,Y=100 are both correct answers.

A domain to which CLP has been successfully applied is that of scheduling. The vast majority of literature dealing with scheduling or timetabling utilise CLP techniques or their variants in order to solve the problem. Constraint Logic Programming has been utilised in solving varied scheduling problems, such as the scheduling of oil well activity [5], the scheduling of forest insecticide treatment [6], through to the scheduling of job shop style activities. It is generally accepted, within the literature, that CLP represents one of the most effective methods of solving complex scheduling problems. Whilst it seems there has been no research performed in scheduling software migrations, there has been a significant amount of effort expended in maintenance scheduling. The maintenance scheduling problem bares some similarity to the migration problem, because maintenance typically involves a resource being unavailable for a period of time (outage), and optimising for minimised downtime, cost and order of execution. One area where considerable research is being performed is in the area of Power Station Maintenance Scheduling.

A typical power plant maintenance operation would involve between 5,000 and 45,000 activities that need to be scheduled. Each activity has explicit constraints that dictate the order of work, for instance, the powering down of one generator would be a prerequisite activity for any maintenance work. In that sense, there is a great parallel with the problem of migration scheduling.

In addition, the total outage for a plant must be as small as possible, ensuring ordering constraints are adhered to, and also that the safety constraints are enforced.

Alguire and Gomes [7] have also carried out extensive research in the application of CLP to power outage scheduling. They found that the current method of scheduling power station maintenance was a manual task, based heavily on the Critical Path Methodology (CPM).

They found that whilst this method worked quite adequately in generating schedules, it did not take into account the safety constraints. As such, after each schedule was generated, a Safety System Functional Assessment (SSFA) Model was applied to the schedule, to determine whether it met all of the required safety constraints.

CLP was selected as the approach due to the requirement that the resulting solutions be readily proveable as optimal solutions. A further reason was the

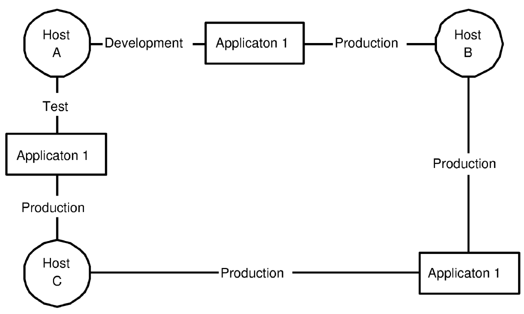

Fig. 2. Dependency Model

relative simplicity by which the problem could be defined using CLP constructs, relative to the use of genetic algorithms' fitness functions.

5 Applying CLP: Proposed Method

Figure 2 shows three hosts associated to one another by shared applications (Application 1). In order for a schedule to be built, it is necessary to determine the dependencies between each of the hosts.

The shared application resolves to a shared application dependency – where Host A has the test environment of Application 1 and Host C has the production environment.

5.1 Constraint Generation

For each host, identify the applications installed on it. Then, for each application, identify the other hosts that application is stored on and the environments. Referring to the rule that development must be scheduled prior to production, we can impose the following constraints:

Host B > Host A (Host B is scheduled after Host A); Host C > Host A (Host C is scheduled after Host A);

There is no constraint on the ordering of Host C versus Host B, as both are production hosts.

If there other dependencies that affect ordering, such as the existence of shared storage or transactional dependencies, then these would also be modelled during this phase.

5.2 Resourcing Constraints

Referring to the data that was sourced, each machine has a system administrator. Assuming that, for the example in Figure 3 each machine has the same system administrator.

Based on that, there is a constraint that only one machine can be migrated at any one time.

5.3 Solve

The resulting constraints are then fed into the solver. The solver uses the Eclipse 5.2 CLP environment, developed by Imperial College's Centre for Planning and Research Control (IC-PARC). The tool is freely available to academic users, and incorporates Application Programming Interfaces (APIs) to Visual Basic, Java, C++ and other languages.

CLP Problem Expression. First, the required libraries are loaded into Eclipse. In this case, it is a finite domain problem (FD), so it is only necessary to load the FD library. A finite domain problem is one where the range of possible values is limited to a defined (finite) domain.

```
:-lib(fd).
```

Each machine being scheduled is represented by a data structure consisting of the following attributes:

1. When a machine will be migrated (start);
2. Prerequisite Machines (pre);
3. Machines that must be scheduled at the same time (same);
4. Machines that cannot be scheduled at the same time (prohib);
5. The Human Resource that must perform the migration (use);

```
:-local struct(host(start,need,same,prohib,use)).
```

The solution space is then modelled, with all the hosts enumerated. As CLP (like prolog) works on the principle of defining goals, it is necessary to define a goal called solve:

```
solve(Solution):-
```

The "atoms" that make up the solution are then defined. In this case, it is the hosts that are being scheduled. Once the solution domain is defined, the structures that contain the dependencies for each host are built:

```
solve(Solution):- Solution=[HostA,HostB,HostC], HostA = host with
[need : [] ,same : [], prohib: [], use : Dev], HostB = host with
[need : [HostA] ,same : [], prohib: [],use : Dev], HostC = host
with [need : [HostA] ,same : [], prohib: [],use : Dev],
starts(Solution,L), L :: 1..3,
```

The following constraints specify that machines that have prerequisites must not be scheduled before those machines, that machines that must be scheduled at the same time as others are, and that machines that are forbidden to be scheduled together are not allowed.

```
% Precedence constraints

( foreach(host with [start:Si,need:Neededhosts], Solution) do (
foreach(host with [start:Sj], Neededhosts), param(Si) do Si #> Sj)
),

% Select machines that have to be migrated at the same time
( foreach(host with [start:Si,same:Samehosts], Solution) do (
foreach(host with [start:Sj], Samehosts), param(Si) do Si #= Sj)
),

% Select machines that cannot be migrated at the same time
( foreach(host with [start:Si,prohib:Prohibhosts], Solution) do (
foreach(host with [start:Sj], Prohibhosts), param(Si) do Si #\=
Sj)

),
```

The remaining code imposes several generic constraints that apply to all problems in this domain:

1. The constraint that no two machines can be upgraded simultaneously if both require the same resource.
2. The constraint that the schedule must be the optimum solution (lowest cost);

```
% minimize cost
min_max(( no_overlaps(Solution), labeling(L)),X), nl, write(L),nl.
max([X],X). max([X|L],X):-max(L,Y), Y =< X.
max([X|L],Y):-max(L,Y), Y > X.

starts([],[]). starts([host with [start : X]
|L],[X|L2]):-starts(L,L2).

no_overlaps(hosts) :- ( fromto(hosts, [host0|hosts0], hosts0, [])
do ( foreach(host1,hosts0), param(host0) do host0 = host with
[start:S0, use:R0], host1 = host with [start:S1, use:R1], ( R0 ==
R1 -> no_overlap(S0, S1) ; true ) ) ).

no_overlap(Si,Sj) :- Sj #> Si. no_overlap(Si,Sj) :- Si #>
Sj.
```

5.4 Analysis of Output

The solver outputs the following cryptic message:

```
[1, 2, 3]
```

Table 1. Performance Results

$NumberofHosts$	$TimetoSolve(secs)$
18	0.01
36	0.03
54	0.13
108	2.83
162	19.71
216	91
270	138.75

This represents, [Host A = 1, Host B =2, Host C = 3], where the number denotes the "slot" where that activity should be performed.

This data was then entered into Microsoft Project in order to produce a Gantt chart. Once in that format, real dates are assigned to the notional slots, and the project plan is used as the basis for the execution phase.

6 Conclusion

In assessing the effectiveness of the approach, it was necessary that the system perform better than the manual method, with a lower margin of error, and with an output that can be readily proven and explaining to people not literate with scheduling algorithms.

6.1 Performance Tests

In order to ascertain the scalability of this approach, a series of tests were conducted with varying numbers of hosts and constraints. One of the key success factors to the usability of the developed approach is that it performs faster than manual methods and that it is scaleable.

Tests were conducted on infrastructure models of 18, 36, 54, 108, 162, 216 and 270 hosts, using a 1.4gb Athlon processor, with 512mb Ram. 1 documents the results.

Based on these results, some projections can be made as to performance against larger numbers of hosts. As Figure 3 shows, overall performance is very good, with even 2,000 hosts taking approximately only 53 minutes (3197 seconds) of CPU time.

It is expected that most clusters would still be under 100 hosts, which would place performance well within an acceptable range.

In order to develop some baseline for measuring the effectiveness of this approach versus manual approaches, we surveyed a large IT outsourcing company that had previously attempted a number of large scale migrations. They estimated that for each cluster (of approximately 20 hosts), the time to build a schedule was 4 hours, with an additional 3–4 hours of rework once all clusters

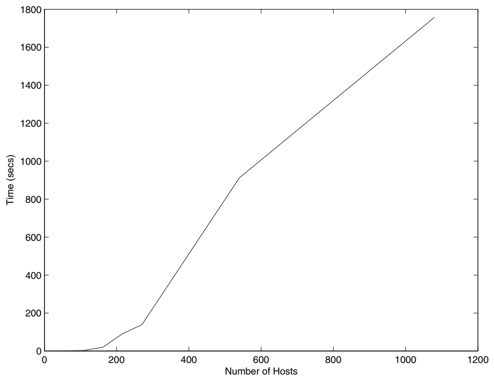

Fig. 3. Performance

were completed. This represents a time, per host, of 12 minutes as compared to less than 1/100th of a second per host using the method outlined here.

7 Further Work

Further research can be performed in the development of some concept of "soft constraints". A "soft constraint" is a constraint that can be broken in certain circumstances. For instance, we may specify that we want all development hosts to migrate before production, however if it is not possible, then as long as at least one development host has been migrated then production can be done.

There is a well-known issue in CLP where a problem can be over-constrained – meaning that it is essentially impossible to solve without breaking constraints. Through the utilisation of soft constraints, it is possible to put priorities on constraints – separating them into mandatory and preferred.

There are significant opportunities for automating and streamlining the process of schedule generation. Of particular interest, would be to automate the interface between the Eclipse solver and the configuration database. As eclipse has a Java-based interface, a GUI-based front end could be built that enables the data to be dynamically sourced from a configuration database and sent to the solver in a single pass.

The scheduling of activities that take place during an outage provides opportunities for further research. For instance, in some cases, there is a need for cross-discipline support, such as from both DBA's and applications support staff. Being able to handle multiple resources and the activities performed within each outage would improve the ability to plan more complicated migration efforts, some of which may require multiple outages.

Whilst an assessment was made regarding the suitability of several scheduling approaches, there is still some research that can be conducted into the usefulness of genetic algorithms (GAs) to solving these problems.

The challenge in a GA-based approach is constructing a fitness function to determine a "good schedule". An interesting aside might be to investigate how user feedback can be incorporated into the model to capture some of the constraints that are not necessarily easy to define. For instance, in the authors reading of the literature regarding power plant scheduling, it mentioned consistently the role that "gut feel" plays in building good schedules. Using some of the existing AI technologies, it may be possible to capture these abstract constraints via user feedback on the quality of generated schedules.

References

1. Coulouris, G.e.a.: Distributed Systems Concepts and Design. third edn. Addison-Wesley (2001)
2. Bartak, R.: Towards mixing planning and scheduling. In: Proceedigns of CPDC2000 Workshop. (2000)
3. Freuder, E.C.: Synthesizing constraint expressions. Communications of the ACM **21** (1978) 958–966
4. Bartak, R.: Constraint programming: In pursuit of the holy grail. In: Proceedings of Workshop of Doctoral Students'99. (1999)
5. Johansen, B.e.a.: Well activity scheduling – an application of constraint reasoning. In: Proceedings of Practical Applications of Constraint Technology (PACT). (1997)
6. J. Adhikary, G.H., Misund, G.: Constraint technology applied to forest treatment scheduling. In: Proceeding of Practical Application of Constraint Technology. (1997) 15–34
7. Alguire, K., Gomes, C.: Roman – an application of advanced technology to outage management. In: Proceedings of 1996 Dual-Use Technologies and Applications Conference. (1996)
8. Gomes, C.: Automatic scheduling of outages in nuclear power plants with time windows. Rome Lab Technical Report (1996)
9. Langdon, W.: Scheduling maintenance of electrical power transmission networks using genetic programming. Research Note RN/96/49 (1996)
10. Louis, R., e.a.: Software agents activities. In: Proceedings of 1st International Conference on Enterprise Information Systems. (1999)
11. Mason: Genetic Algorithms and Job Scheduling. PhD thesis, Department of Engineering (1992)