UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Practical Approach to Fault-Tolerant Master/Worker in Condor

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science in
Computer Science

by

Ing-Wher Chen

Committee in charge:

Keith Marzullo, Chair
Henri Casanova
Jean Ferrante

2002

The thesis of Ing-Wher Chen is approved:

Chair

University of California, San Diego

2002

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank Miron Livny, the Condor team at University of Wisconsin, Madison, Xianan Zhang, Nadya Williams, the GriPhyN team at USC, and the Grid Research and Innovation Laboratory at UCSD for their helpful input, support, and the use of machines. I would also like to thank the members of my committee, Henri Casanova and Jeanne Ferrante, for their generous feedback. I especially would like to thank Keith Marzullo for his guidance and kindness throughout this process. Although seldom (and perhaps poorly) expressed, it was a pleasure working with Keith and an experience I cherish. Lastly, I would like to thank my family and friends for their unconditional love and support.

ABSTRACT OF THE THESIS

A Practical Approach to Fault-Tolerant Master/Worker in Condor

by

Ing-Wher Chen

Master of Science in Computer Science

University of California, San Diego, 2002

Professor Keith Marzullo, Chair

This thesis addresses the problem of providing fault-tolerance for master/worker programs from a practical point of view.

Although there have been many studies on fault-tolerance for master/worker programs, the tools that have been deployed and are commonly used only address the crash failures of workers. When computations scale to run over wide-area networks, then network partitions can occur. If the computations are large enough, the master can become a bottleneck and needs to be replicated as well. To the best of our knowledge, large scale master/worker on wide area networks has only been studied as a theoretical problem.

For this thesis, we develop a master/worker toolkit that can scale to wide-area systems. It is built using Condor, which is a popular vehicle for Grid-based computing. Our toolkit supports different configurations of replicated masters and different policies for partitioning out work among workers, thereby allowing the developer to tune the exact structure of the computation to best serve the environment and problem at hand.

In developing this toolkit, we are able to extract some common rules when providing fault tolerance to master/worker programs in Condor and existing distributed systems.

1

Introduction

Through the last decade, prices for personal computers have steadily dropped. At the same time, these computers have become more powerful, equal to that of a workstation. These advantages and the prevalence of personal computers have prompted studies in Grid computing system to satisfy the increasing demand for computing power. Studies in Grid computing typically focus on resource management of these machines that make up the Grids. As computational Grids become more popular, there has been more research done on providing tools to help researchers in diverse disciplines learn to use such systems.

Since a significant amount of the demand lies in processing large amounts of data in a similar manner, many parallel applications are structured in a master/worker framework. Such a tool typically has a central controlling point, often called the "master", that monitors a job, usually divided into hundreds and thousands of tasks that can be performed in parallel. The underlying Grid system then allocates machines to run as "workers" for this master. The master subsequently distributes its tasks to these workers and compiles a final output for the job as workers return with results.

As computational Grid systems mature and fast networks become prevalent, recent research has focused on building wide-area Grid systems running large scale jobs. As the size of the computation increases, the master in a master/worker application becomes the bottleneck. With the increase in computation time, there is also a greater chance of the master failing. Furthermore, when computations run over wide-area networks, network partitions can occur. Although the SETI@home project [28] has successfully been deployed across a wide area network and has high throughput, it handles network partitions by merely waiting for the partitions to cease. This approach is not appropriate for applications that are concerned not only with throughput but also with the length of execution time.

In this thesis, we attempt to develop a scalable and fault-tolerant master/worker toolkit. We have successfully built this tool on a popular platform, Condor, which has the interesting characteristic of allocating only idle machines to process jobs. Through working in such a unique, and sometimes limiting, computing environment, we have learned some common guidelines, such as the observation that providing fault tolerance requires cooperation of both software and machines, in building fault-tolerant applications.

The rest of the thesis is structured as follows. In Chapter 2, we discuss some related work that played important roles in the design of our tool. In Chapter 3, we provide an overview of an existing master/worker tool in Condor and our implementation of a fault-tolerant master/worker template. In Chapter 4, we further discuss details of our implementation and our attempt at providing a practical template. Chapter 5 provides some test results of our new tool in various conditions in addition to some observations we made on Condor itself. Lastly, in Chapter 6, we conclude with some thoughts and our experience during this process.

2

Related Work

There have been extensive studies on Grid computing systems, distributed applications, and parallel master/worker programming. Traditionally, many of these studies only consider fundamental problems regarding cooperation between individual machines. For the well-studied problem of master/worker applications, most studies focus on the allocation of machines and the scheduling of tasks among machines. Because such a system involves multiple machines and multiple independent tasks, some degree of fault tolerance for such applications is implicitly provided. A failed machine does not interfere with other machines or tasks in the system, or the success of the job. As these distributed systems span across wide area networks, requiring thousands of worker machines and taking days to complete, new scalability problems arise and existing fault tolerance concerns are magnified. Systems deployed across a wide area network over a longer period of time are more prone to experience network partitions. With the larger scale computations, having one single master may become the bottleneck of an application, as well as being impractical during network partitions. In this thesis, we have set forth to address the fault tolerance needs in wide area master/worker applications.

The following sections describe the various studies that led us to create our master/worker tool and to one design decision in building such a tool. In Section 2.1, we survey one approach to programming in computational Grids based on templates. In Section 2.2.1, we describe one of the computational Grids deployed across wide area networks. In Section 2.2.2, we look at theoretical studies on issues concerning partitionable systems. Finally, in Section 2.2.3, we provide examples of actual deployment of such systems across wide area networks.

2.1 Programming Templates

Programming on a computational Grid, unlike on a single machine, is much more complex due to the underlying communication among cooperating machines. To reduce the burden of implementing this underlying communication each time a new application is created, several toolkits have been developed for computational Grids. Most notable are the master/worker parallel programming toolkits. The master/worker parallel programming paradigm is one in which a group of processors cooperate to finish a job. There is typically one processor acting as the master to administer the job. A job is usually divided into several small parallel and independent tasks. The master is responsible for distributing these small tasks to a group of worker processors and collecting results from them. Some examples of such toolkits are the AppLeS Parameter Sweep Template, the AppLeS Master/Worker Application Template, and the Master/Worker template on Condor. All of these tools have been widely accepted. These templates provide the frameworks for the master, the central controlling process in an application, and the worker, the process that is assigned to a remote machine that handles the bulk of the work. Communication mechanisms between these two kinds of processes are also built into the templates. They allow users to customize the tool to create new applications without re-implementing and debugging the underlying communication functions.

2.1.1 AppLeS Parameter Sweep Template

AppLeS Parameter Sweep Template (APST) [3, 9] is a template that aides in the development of applications that take advantage of the computing power in computational Grids. For example, one of the simulation programs that has been used in conjunction to APST is the Monte Carlo Simulator for Cellular Microphysiology (MCell) [23]. Communication within a living cell is achieved by chemicals moving in the cell and interacting with other chemicals. MCell simulates these interactions within a cell given different scenarios. Each of these scenarios are independent and can be simulated in parallel on remote machines. Without APST, users would need to manually submit the

MCell program along with various input files to different machines. Managing for these remote processes, such as restarting failed processes, would also need to be performed manually.

APST is thus designed to take a user-specified program and input files, and automatically manage the job over a group of machines. It is to be used for specific kinds of experiments or simulations where information is gathered for a set of parameters. With this template, a user who wishes to submit tasks provides a *task description file* written in XML. Each task description specifies the program to run, command-line arguments, input and output files, etc. For instance, to start five runs of MCell simulation with five different scenarios, a user constructs a task description file that contains five XML sections—each section specifying MCell as the program to run, command-line arguments, input files to describe a particular scenario, and the location of the output file. The user then starts the APST daemon (APSTD) and submits to the APSTD the task description file. The APSTD distributes tasks into the Grid system as it sees fit until all tasks have finished.

The developer claims that there is some degree of fault tolerance built into such an application. The failure of a machine executing a task does not affect the APSTD or the other machines executing tasks. The APSTD can simply distribute tasks on failed machines to other machines, although this may subsequently cause the job to finish behind schedule. Although some simple checkpointing is performed, the APSTD itself is still a single point of failure. Failure of the APSTD would require the user to restart the application. Hence, even when using APST, users should still expect to take up some accounting responsibilities manually. Furthermore, users may also be required to personally manage the experiment if network partitions occur between remote machines and the APSTD.

Thus, although the APST solves many communication and cooperation problems when using computational Grids, and makes them more accessible to scientists in different disciplines, it fails to consider some problems one can encounter in a distributed system.

2.1.2 AppLeS Master/Worker Application Template

The AppLeS Master/Worker Application Template (AMWAT) [1] is another tool built to aide programmers and scientists in using computational Grids. Instead of a template that helps the user manage a distributed application, this template can be used to create master/worker applications targeted to Grid resources.

To use AMWAT to create a master/worker application, an application programmer provides code for application-specific functions defined in the AMWAT application interface. A customized `main()` function can also be provided to invoke the AMWAT scheduler, which is the master process, as well as other actions to be taken before and after this invocation.

A configuration file provided by the user is used to enable an AMWAT application to run in parallel. It lists specific machines to be used in the parallel program. Additional information regarding the machines can also be provided for load-balancing among machines of different speeds and abilities. The configuration file can also indicate the particular process to be created on each machine listed. Each process is either a worker process or a sub-master process. The relation among the processes are indicated using the parent field in the configuration file. For example, the master does not have a parent because it is the root, and the parent of a sub-master or a worker is either the master or another sub-master.

To start a master/worker distributed application, the AMWAT scheduler (master process) is invoked using the `main()` function and the configuration file described above. The AMWAT scheduler is always started on the machine on which the application is invoked. Based on the application-specific functions, the AMWAT scheduler divides the job into a number of independent work units. It also spawns remote worker processes and sub-master processes on specific machines. Once all processes are started, each parent process is responsible for assigning work to its child processes, balancing the workload among its children such that faster processes are given more work than slower ones, and collecting results from them. Periodically, the master collects results from *its* child precesses and composes the results into a single application output.

Similar to APST, there is some degree of fault tolerance built into an AMWAT application because it allows multiple workers. The failure of one worker process or one machine running a worker process does not affect other machines and processes. Furthermore, with the support of multi-layered computation, which allows one single master and more than one sub-master, the user can design an application to tolerate the failure of the master, eliminating the single point of failure at the master process. Such a multi-layered configuration can potentially tolerate network partitions if sub-masters were placed such that when the network partitioned, each group of connected machines contained the master (or a sub-master) and its child worker processes. In this case, each group of machines can continue to make progress. However, the downside in this template is the allocation of computing resources. Unlike APST, the machines used in an AMWAT application are statically listed in the configuration file. Instead of using a service provided by the underlying Grid system to allocate suitable machines on the Grid, the user is required to specify the machines to be used before the start of the application. Thus, the application has no way of recovering from a machine failure. The unfinished work would have to be handled by the processes on the remaining machines. In addition, the use of the static configuration file means that the machine allocation and the relations among the machines and processes would have to be worked out for each Grid system and possibly even for each invocation of the application.

2.1.3 Condor Master/Worker Template

Condor's Master/Worker (M/W) template [15, 19] is also a template that helps a user build master/worker applications that utilize a Condor pool [20]. This template consists of two parts: the *master template* and the *worker template*. The master template provides code to create worker processes and to communicate with them. The worker template provides communication between a worker and a master.

A user building a parallel application on Condor can take this M/W template and fill in the necessary functions to create an application. One such function required in the master template is the code that divides a job into several smaller tasks. Other necessary functions for the master are ones to send and receive information to and from workers. Finally, the user also needs to provide code for the master to handle the information, usually results computed by tasks, from workers. Similarly, functions required by the worker template are those to send and receive information to and from the master. A function indicating how a worker should process a task should also be provided. With these application-specific functions and the built-in communication and administrative functions, a user can easily build a parallel M/W application.

To start this application, a user submits the master process to Condor. With the code provided by the user, the master divides the job into parallel tasks. Following the built-in code, the master submits several worker processes to Condor. Once a worker process is started, it contacts the master for a task and executes the task as specified by the user. Task results returned from workers are also merged as defined by the user.

As with AMWAT, users may freely extend the template into any M/W application in which the master and worker processes cooperate to produce one coherent outcome, rather than a set of independent results. In terms of providing fault tolerance, this template, being deployed on a Condor pool, is capable of tolerating random worker machine failures. Condor is a computational Grid built on *borrowing* idle CPU cycles. Machines in a Condor pool can become unavailable at unpredictable times. Thus, the Condor system monitors running jobs as well as machines in the pool. In the event that a machine becomes unavailable, Condor restarts the job running on that machine to another suitable idle machine, ensuring that a task eventually finishes.

However, this M/W template is still vulnerable to the failure of a master. The master is responsible for ensuring that the job finishes, despite uncertainties regarding the worker machines. To this end, the master runs on a machine dedicated to running this process without interference from Condor or any other monitoring devices, and hence, becomes a single point of failure. In the event that the master fails or partitions away from workers, workers would not be able to contact the master. If this occurs, the workers terminate.

Both templates described, the APST and Condor's M/W template, have successfully provided convenient tools to help users work with computational Grids. However, both templates have single points of failure and neither tolerates network partitions. We feel that these two properties are important when building applications on wide area distributed systems due to the instability inherent in such systems. Therefore, with these properties mind, we hope to take advantage of the flexibility in templates to provide a new template that is both powerful and robust in a volatile environment.

2.2 Computation in Wide Area Networks

One problem that arises with a computational Grid deployed across wide area networks is the relatively poor network connections among wide area networks. Poor wide area network connections lead to a failure mode commonly called network partition. As mentioned briefly in Section 2.1.1 and Section 2.1.3, many current applications built for Grid systems do not consider (and hence do not tolerate) network partitions. In the following, we first introduce in Section 2.2.1 a Grid system deployed across wide area networks, called Condor flocks. In Section 2.2.2, we survey two techniques useful for building partitionable systems. Lastly, in Section 2.2.3, we describe some real distributed applications that have been built to tolerate network partitions.

2.2.1 Computer Grids on Wide Area Networks

Until recently, each computational Grid has been a research project started and developed by a single institution. The resulting Grids were usually deployed across a local area network within an institution. As such studies mature and fast networks become widely available, it has become practical to deploy Grid systems across wide area networks. Several projects, such as Globus [14] and Legion [18], have focused on providing platforms for wide area computation. Another wide-are application, SETI@home [28], has been successfully deployed with high throughput but long execution time which may be unacceptable for certain applications. In this thesis, we focus on one particular system, the *Condor flock* [12, 13].

A Condor flock is a system consisting of several local Condor pools, each perhaps from different universities or research labs, “flocking” together to agree to process jobs for each other. Individual Condor pools in a flock are linked together through machines called *world machines*. A Condor pool consists of a *central manager* and other machines that have agreed to share workload. The central manager is a machine that monitors both jobs submitted to the pool and the other machines in the pool. To share resources between two Condor pools, each pool has a world machine. Each world machine represents a free machine in another Condor pool. For example, with a Condor flock consisting of two Condor pools, A and B, each pool contains a world machine. The world machine in Condor pool A represents a free machine in Condor pool B and vice versa. When the central manager in Condor pool A finds the world machine to be a suitable machine to process a job, it allocates the job to this world machine in the same manner it allocates jobs to other machines. This job is subsequently processed by the remote machine in Condor pool B that the world machine represents. We will see the limitations of Condor flocks and how we take advantage of this structure in Section 3.3 and Section 3.5.

2.2.2 Studies on Cooperation in Partitionable Systems

In a distributed system, such as the one described in the previous section, where machines are dispersed across wide area networks, network partitioning is a reality. Thus, a useful distributed application should provide mechanisms for disconnected machines to continue operation. The following sub-sections describe some techniques used to build efficient and reliable partition-aware systems.

2.2.2.1 Distributed Cooperation in the Absence of Communication

Consider a system with n processors cooperating to finish t tasks, which are all known to all processors. The processors are spread across a wide area network that can suffer frequent partitions. These processors are not arranged in the master/worker configuration but are equal instead. At the end of execution, each processor should know the results of all t tasks. A processor can know the result of a task by computing the task directly or by being given the result from another processor. A trivial solution to this problem is to schedule all tasks on all processors, where a schedule is a list of tasks given to a processor to execute in order. Although this approach tolerates partitions, since it requires no communication among the processors, it incurs a large amount of redundant work—each task is processed n times. There is clearly a tradeoff between the extent of communication, which is dependent on the networking environment, and the amount of redundant work. In fact, [21] shows that the complexity in the construction of efficient schedules also complicates the goal of minimizing wasted work.

[21] shows that it is relatively easy to deterministically construct, within a reasonable amount of time, short schedules that minimize redundant work. Schedules of

length $\Theta(\sqrt{n})$, such that exactly one redundant task is performed for any pair of processors, can be constructed independently in $O(\sqrt{n})$ time by each processor. However, such short schedules imply that processors can resume communication after they each finish their tasks. For longer disconnections, longer deterministic schedules of length $\frac{4}{9}n$ can be constructed in time linear to the length of the schedules. With this construction, for any two processors, p_1 and p_2, the overlap, tasks that are performed by both processors, of the first t_1 tasks of p_1 and the first t_2 tasks of p_2 is bounded by $O(\frac{t_1 t_2}{n} + \sqrt{n})$. The study also shows that randomized construction of schedules yields surprisingly good results. By allowing each processor to randomly pick the next unprocessed task to compute, one produces schedules such that for any pair of processors, p_1 and p_2, the expected overlap of the first t_1 and t_2 tasks of p_1 and p_2 respectively is no more than $\frac{t_1 t_2}{t} + O(\log n + \sqrt{\frac{t_1 t_2}{t} \log n})$.

Hence, we see many approaches in scheduling tasks, each of which is suitable for a particular environment and need. In our new template, we implemented mechanisms (to be discussed further in Chapter 4) to allow users to customize the scheduling algorithm in the template to fit particular applications and network environments.

2.2.2.2 Group Membership Protocols

The ideas in the previous section can be applied to applications that have a well-defined number of processors and independent tasks. Such applications can often function at an acceptable level without knowledge of communication failures. There are, however, other distributed applications that require closer cooperation among processors. For these applications, an awareness of the networking environment is necessary to offer acceptable quality of services. Thus, many studies have proposed a set of services to be provided by the system to support partition-aware applications. Such services are often referred to as group membership services.

A group membership service typically identifies processor groups [8] in each network partition. In [4, 5, 6], the protocol provides a *view* to each processor. A view is an abstraction of the environment in terms of partitions and merges. It is how a processor "sees" the network environment—as sets of strongly connected processors. Processors that agree on the same set of connected processors *install* one view. The protocol tracks the changes in the environment and advises processors of the changing of views. In addition to providing processors knowledge of other connected processors, this protocol also provides a *view synchrony service*. With this service, all processors that belong to a given view and subsequently install the same successor view, deliver the same set of sequence of messages.

Such a service is clearly beneficial in the implementation of many sophisticated distributed applications such as the Bancomat problem (discussed in Section 2.2.3.3). It restricts the possible failure scenarios. For instance, with the view synchrony service, two processors that install the same views can conclude deterministically that they have the same information. Hence, further computation can be based on one's knowledge which requires no further communication. However, in terms of simpler applications, such as the M/W application in Section 2.1.3, where possible conflicts are simple duplication and incompleteness of task results, we feel that the cost of implementing a group membership service is too high for such low return. Hence, in our implementation of M/W template, we have decided not to build upon a group membership protocol.

2.2.3 Disconnected Operations in Applications

Although not many Grid applications have considered operation during the absence of communication, many distributed applications have long provided disconnected operations. The following sub-sections discuss such applications.

2.2.3.1 Coda Distributed File System

Many studies have focused on distributed file systems. One of the topics often studied is providing highly available distributed file systems. In the Coda file system [17, 27], researchers have provided a solution to increase the availability of the distributed file system in the face of both voluntary and involuntary disconnections between the file servers and client machines.

The first step to providing high availability in Coda is replication of servers. Files are replicated across several servers. Hence, failure of some file servers may not adversely affect the availability of the system. The second step is the caching of files on local client machines to be used in the event of an involuntary network partition or voluntary disconnection. Coda provides mechanisms to allow users to specify a list of files that should be cached for future disconnected operations. In a fault-free system, a connected client is able to gain access to up-to-date files by retrieving from a replicated file server while a disconnected client may use its local copy to tie it over until connection is re-established. In a system with network partitions or server failures, a connected client may still be connected to some fault-free replicated server and gain access to files.

Because of this design, files may be accessed and modified simultaneously. Albeit rare [25, 26], it is still imperative to provide mechanisms to resolve these conflicts. In Coda, resolution of conflicts of file updates are done when disconnected clients connect to a replicated server again. It is also done when network partitions heal and replicated servers are reconnected.

Overall, the Coda file system is a file system that can offer some services during involuntary disconnection. Users can also plan their disconnections by caching certain files in advance. Conflict resolution is achieved in terms of files. We will see in the next section that this approach may not be the optimal solution to resolve conflicts in file systems.

2.2.3.2 Bayou Storage System

Bayou [30] is another storage system that tolerates network partitions. In fact, it is specifically built for an environment, such as a wireless network, with weakly connected computers.

To provide a highly available storage system despite frequent disconnections, Bayou takes the same approach as Coda—it replicates the storage system across various servers. Unlike Coda, which distinguishes between connected and disconnected servers, Bayou relies only on pair-wise communication between computers. When two servers (periodically) connect with each other, they exchange information between themselves via *anti-entropy* sessions [11]. At this point, Bayou provides application programmers and users with two functions to allow the system to resolve conflicts. For each entry or record in a file, application programmers can provide a function to detect conflicts. When a conflict is detected, the programmer can provide another function to specify alternate values or actions to resolve inconsistencies.

In addition to the ability to merge files between two servers in a single anti-entropy session, extra properties are needed to provide *eventual consistency*. With anti-entropy sessions, Bayou guarantees that all servers eventually receive all writes. However, merging different copies of files in different orders may cause inconsistency among replicated servers. Therefore, in addition to deterministically detecting and resolving conflicts, Bayou also ensures that all writes are performed in the same order at all servers. Hence, all replicas that receive the same set of writes *eventually* contain the same data.

Overall, Bayou takes an approach similar to that taken in Coda to tolerate network partitions—replication. Each replica is free to offer the complete array of services under any circumstances. None of the replicated servers in the system requires any particular network configuration knowledge of other replicas. Unlike Coda in which all conflicts are resolved in terms of files, conflicts in Bayou are resolved deterministically with application-specific recovery schemes. Thus, Bayou presents a more general, improved approach to conflict resolution.

2.2.3.3 Bancomat Problem

The Bancomat problem in [29] studies a resource allocation problem in partitionable systems. This problem is similar to that of automatic teller machines. Unlike Coda or Bayou, resource in this type of application should not and perhaps cannot be replicated. For example, consider a system of n automatic teller machines (*bancomats*) that share an initial balance of B amount of money. Clearly, a bank cannot replicate the amount of money a client deposited in order to provide services at various bancomats or branches just in case of network partitions. The strategy taken in such a problem is to divide the resources among the group of bancomats and require them to *borrow* or *reallocate* resources when necessary.

In the case of the banking problem, there are two actions a client can take at a bancomat—deposit d amount of money and withdraw d amount. To tolerate network partitions, the initial B amount is divided among the n processors. Deposits are done at the machine that received the request. When a bancomat, b, receives a withdrawal request, it tries to fulfill that request with its portion of funds. In the case of insufficient funds, b tries to borrow funds from other bancomats. Thus, instead of having a merging problem as in Coda and Bayou, this system has a communication problem and the possibility of waste of resources.

Depending on the coordination scheme among the bancomats, different amounts of money could be rendered unavailable to clients. For example, when a client makes a request at bancomat b_1 to withdraw w amount while b_1 only has a local amount of $w_{b_1} < w$, b_1 may try to contact another bancomat, b_2, for additional money, an amount of $u = w - w_{b_1}$. When b_2 receives this request, it subtracts u from its local balance, w_{b_2}, and sends a message to bancomat b_1 regarding its decision. However, if b_1 and b_2 partitioned away at this instant, having not received the money from b_2, b_1 could not use the u amount of money b_2 had agreed to lend. On the other hand, b_2 could not access that u amount of money even though it would never be used by b_1. Hence, the withdrawal cannot be serviced at b_1 despite having sufficient funds to service the request in reality.

In this type of application where resources cannot be replicated, reallocation of resources is often necessary to provide services in the face of network partitions. Hence, efficiency in communication and coordination among cooperating servers is vital to the quality of such applications. As mentioned in Section 2.2.2.2, group membership services can be useful in such applications because they can reduce uncertainty in global states when disconnections occur, and hence, reduce waste and increase communication efficiency. Furthermore, the cost of incorporating a group membership service in such applications may be reasonable, considering the potential waste of resources and degradation of service that may occur without it.

2.3 Summary

2.3.1 Design Decision

From the previous sections, we see practical approaches, using templates, to implementing tools for Grid systems. However, despite the reality of partitions in computational Grids (due to their deployment across wide area networks), we do not know of any templates that can tolerate partitions. Thus, out of necessity and based on our survey of theoretical techniques in partitionable systems and existing distributed applications, we have come to design a new M/W template, one that can tolerate network partitions, to be used on Condor. In our design, we use the replication of master processes to tolerate master failures and network partitions. Since each master has knowledge of all the tasks, and conflicts in our applications only involve duplicated or missing results, we believe that this template is simple enough to do without a group membership service. Instead, to facilitate cooperation among masters, we utilize the power of separation of mechanism and policy (described in Chapter 4) to provide a cheaper solution.

2.3.2 Potential Use

Parallel programming templates have made computational Grids more accessible to researchers in all disciplines. As computational Grids continue to expand across wide area networks, partition-aware parallel application templates may become increasingly important in helping researchers navigate through such large systems. We hope our attempt to provide a template for Condor that tolerates network partitions may enhance other systems, such as Nile [10, 22, 24] and GriPhyN [16], that are also deployed across wide area networks.

3

A Partition-Aware M/W Template for Condor

3.1 Brief Introduction to Condor

Condor is a distributed system built to take advantage of idle CPU cycles. A Condor pool consists of a group of computers that can execute jobs for each other to increase their utilization. When joining a Condor pool, the owner of each machine specifies the terms under which his or her machine will process job requests from other machines. For example, a resource owner may specify that a machine can only accept a job request when it has been idle for two hours, or may only process job requests between midnight and six in the morning.

One machine in each Condor pool is designated as the *central manager*. The central manager maintains a database of user specifications as well as information about each machine in the pool (e.g., the CPU speed, memory size, disk size.) This information is used to match job requests with machines. A user sends a job request to the central manager, indicating the job's requirements, such as the required amount of memory and disk space. The central manager in turn searches its database to find a suitable machine that is idle and willing, based on the conditions put forth by its owner, to process the job. Once a Condor job is started on a remote machine, the involvement of the central manager varies depending on the *universe* in which the job is started. The central manager continues to monitor certain types of jobs and migrates and restarts these processes to other machines when the current host machines become unavailable (perhaps crashing, or being reclaimed by its owner.)

3.1.1 Universes in Condor

Technical advances have lowered the prices of commodity personal computers while increasing their computating power. Based on the observation that personal computers are frequently idle, such as in the middle of the night, etc., the twist in the Condor Grid system is to scavenge only idle CPU cycles of a group of machines to share processing power. To this end, a Condor job processing on a remote machine is checkpointed and migrated when the central manager detects the return of that machine's owner. Thus, it is possible for a Condor job, especially a long job, to be processed by several different machines during its lifetime. If system calls were used in such a job, inconsistencies could arise due do the different behaviors of certain system calls on different machines. For example, the behavior of the prohibited system call `exec()` depends not only on the behavior of the executable file on different machines but also on whether the file exists or not. Process migration also prohibits the use of memory mapping functions such as `mmap()` and `munmap()` to prevent mapping to different address spaces on different machines. To eliminate the inconsistencies that process migration can cause, Condor limits the system calls that a job can use. The exact set of excluded system calls is determined by the universe in which the job is run. The *vanilla universe* and the *scheduler universe* are of the most interest to us because processes in both versions of the M/W tool are started in these two universes.

A job started in the *vanilla universe* can be migrated by the central manager whenever necessary. Because these jobs can be migrated, they do not have permission to execute system calls that may produce inconsistent behaviors on different machines. One important characteristic of jobs submitted in this universe is that the central manager constantly monitors such jobs. Because Condor does not checkpoint vanilla universe jobs, they are restarted from scratch on idle remote machines if the central manager detects failures of the remote machines hosting these jobs or the return of the owners of these remote machines.

The vanilla universe is also the universe in which worker processes are started in the existing version of the M/W tool. Assuming that the central manager does not fail, fault tolerance for worker processes is automatically supplied by the central manager. If the central manager does fail, worker processes are not migrated if their host machines become unavailable. In this case, a M/W job may fail when the master remains fault-free but with all its workers and the central manager crashed.

Jobs started in the *scheduler universe* are never migrated and thus are allowed to invoke any system call. Such jobs stay on their host machines for the duration of their execution. They are not preempted when the owner of the host machine returns.

The master process in the original M/W tool needs to start workers and so is run in the scheduler universe. To create workers, it simply calls `popen()`, which opens a process by creating a pipe, forking, invoking the shell, and has a behavior that is highly machine dependent, to execute the Condor job submission command, `condor_submit`. By creating only one master in the scheduler universe though, it is easy to see that a M/W application would fail if the master fails.

3.1.2 Submitting a Condor Job

To submit a program to Condor, one uses the command `condor_submit`. This command takes a single argument, which is the name of the *submit description file*. A submit description file specifies the name and location of the executable file, the universe in which this Condor job is to be submitted, and the names and locations of the input files, output files, error log, and work log. It can also contain command-line arguments for the job, hardware requirements of the remote execution machines, etc. For example, Figure 3.1 shows the submit description file to submit a master job to Condor. It indicates that the job is to be submitted in the scheduler universe and that the executable file is a file called `master_matmul`. There are no arguments for the job, but there is an input file, `input_matrix`. An e-mail message is to be sent to `ichen@cs.ucsd.edu` upon termination of this job. The keyword "queue" indicates that exactly one instance of this job is to be submitted.

For a vanilla universe job, the central manager locates an idle machine, copies the executable to that machine, and monitors the job. This job is migrated and restarted when necessary.

```
getenv = True
universe = Scheduler
executable = master_matmul
arguments =
input = input_matrix
output = product_matrix
error = error_log
log = work_log
notify_user = ichen@cs.ucsd.edu
queue
```

Figure 3.1: Example of Condor submit description file.

3.2 The Original M/W Tool

The original version of M/W tool consists of two templates— one for the master and one for the worker. The templates include code that describes the underlying M/W system and communication mechanisms between the master and its workers. They also contain code that dictates the general behaviors of the master and its workers. To turn these templates into working programs, one needs to provide certain customized functions. The following sections describe the two templates and the customized functions in more detail.

3.2.1 Original Master

The master template dictates general behaviors and responsibilities of the master. In a complete M/W application, the master's responsibilities are as follows. It:

1. divides the job into a number of small tasks,
2. creates an appropriate submit description file for the workers and submits a predetermined number of worker processes to Condor,
3. waits for workers to connect and accepts these connections,

4. sends out job information that is pertinent for the entire duration of the job (e.g. initial input file),

5. sends out a task and additional information needed to complete the task to a contacting worker,

6. accepts results from workers,

7. composes the individual results from workers into a comprehensive output, and finally,

8. terminates workers when the job is completed.

Figure 3.2 gives the original master template in pseudocode. It is really written in C++ and is 3504 lines long. To turn this master template into a working program, the user is required to provide certain functions to complete the master program. Of the items listed above, for each application, the user should provide specific functions for items 1, 4, 5, 6, and 7.

Because the division of a job is specific to each application, the user needs to provide a function to divide the job into a number of small tasks. For example, if one wishes to build a M/W application based on this template to compute the product of two 10x10 matrices, such a job can be broken up into 10 small tasks, each of which involves determining one row of the product matrix. The user also needs to provide a customized function to send specific initial job information to workers. Using the example above, the initial job information could be the second matrix. Although communication mechanisms are provided by the underlying template, to send this initial information, one still needs to provide details on *what* to send, which differs in each application. Hence, sending initial information of a job consists of copying only relevant data into a buffer and calling the underlying communication mechanisms to actually send the message. Likewise, although the communication functions used to send and receive messages to and from workers are provided by the underlying M/W template, the user should provide functions to specify what to send in outgoing messages and what to expect from incoming ones. Thus, to send out a task, a master first calls a user-provided function to copy relevant information to the outgoing buffer, and then calls a function provided by the template to send out

```
begin Master

   initial setups;

   create_initial_tasks();

   system ("condor_submit worker_processes");

   begin loop until there are no more tasks

      if there is a worker trying to connect
         accept();
         handle initial setup;
         send_task_to_worker ();
      else if there is a RESULTS message received from a worker
         handle this RESULTS message;
         send_task_to_worker ();
      end if

   end loop

   terminate workers;

end Master
```

Figure 3.2: Sample original master template.

the task. Similarly, to receive results of a task from a worker, a pre-defined function is called to copy the incoming message to buffer and a user-defined function is called subsequently to interpret the message. In the case of the 10x10 matrix multiplication job that is broken up into 10 tasks, where each task determines one row in the product matrix, data sent to a worker regarding a particular task may include a specific row of the first matrix. This row, along with the complete second matrix that was sent earlier, allows a worker to compute a row in the product matrix. The function to receive results from a worker can in turn be customized to read in a row in the resulting matrix. Finally, the user is required to provide a function to interpret the individual results returned by workers to produce a coherent final result. Continuing the example above, this function probably consists of procedures to order the data returned by each worker, which should be rows in the final matrix, to output a correct 10x10 matrix to the user.

3.2.2 Original Worker

Similar to the master template, the worker template also determines how a worker behaves in general. By incorporating some customized functions provided by the user into this template, one can create a working worker program that can cooperate with its corresponding master.

As the worker template dictates, a worker has the following responsibilities. It:

1. executes `connect()` to contact the master,
2. accepts initial data pertinent to the entire job from the master,
3. repeatedly accepts a task from the master, processes the task, and returns the result of that task to the master, and finally,
4. terminates itself when told to by the master.

Figure 3.3 gives the original worker template in pseudocode. The template is in fact written in C++ and is 458 lines long. To complete this template, the user is required to provide functions to process items 2 and 3 listed above. As before, although the underlying template provides mechanisms to send and receive messages to and from a master, these mechanisms do not know how to interpret the data. Thus, the user should provide a function to interpret the message that contains the initial data sent by the master. In the matrix multiplication application described in the previous section, since the initial data the master sends is the second matrix involved in the multiplication, a worker should have the ability to read in, unpack the 10x10 matrix, and take appropriate steps to store it. Similarly, the user should provide a specific function to interpret the message from the master regarding a task. For instance, in the matrix multiplication application, each task involves finding a row in the resulting matrix. Thus, the master sends a different row in the first matrix for each task. Consequently, a worker's responsibility, in addition to accepting the message from the master, is to interpret this data and read in a row of the first matrix. Once all information with respect to a task is ready, a worker begins to process the task. Since the nature of tasks differs from application

```
begin Worker

   connect() to the master and send initial information;

   begin loop

      recv() a message;

      if message tag is DO_THIS_WORK
         unpack data in this message;
         execute_task ();
         pack results into buffer;
         send a RESULTS message to master;
      else if message tag is KILL_YOURSELF
         terminates;
      end if

   end loop

end Worker
```

Figure 3.3: Sample original worker template.

to application, the user is required to provide a function to specify how a worker should execute a task. For instance, since each task in the matrix multiplication application involves finding a row in the product matrix, and a worker is given a row in the first matrix and the entire second matrix, the function dictating how a worker is to process a task is simply doing a simple matrix multiplication of the two matrices it is given, the 1x10 row matrix and the 10x10 second matrix. Once a task is completed, a worker sends a result message to the master. Again, sending such a message first involves a worker copying relevant data into the outgoing buffer. This is done with a user-supplied function. Subsequently, with the help of the existing communication mechanisms, the user sends a proper result message to the master.

3.2.3 Summary

Overall, a parallel application based on the existing M/W templates runs as follows. The master is manually submitted to a Condor pool by the user. Upon starting, the master divides the job into several tasks based on the user-supplied code. The master then uses the `condor_submit` command to start several workers in the pool. After worker submission, the master waits for workers to contact. For each worker, once contacted, the master acknowledges the worker and sends initial data information that is pertinent to the entire job. The master also sends out a task to that worker to start the actual processing of the job. Upon receiving a task, the worker processes the task using the customized function provided by the user, and returns some results to the master. The master then interprets these results and incorporates them into the final output of this job. The master and its workers remain in this loop until the master detects the completion of all tasks.

3.3 Limitations and Challenges

Since there is exactly one master process in each M/W application, a M/W application can fail if the master process fails. In addition, although the worker processes are monitored and restarted by the central manager, it should be noted that the central manager is on one single machine, and thus, is also a single point of failure. Furthermore, the M/W template does not tolerate network partitions in the case where a worker, w, partitions away from the master, m. In this case, w would eventually terminate. Depending on whether the central manager remains in the same network partition as m, m either loses a worker, w, or is given a new worker, w'. If the central manager remained in the same partition as m, it would detect the loss of w, and subsequently locates another idle machine to replace w. If the central manager also partitioned away from m, the loss of w may not be detected and w would be lost to m.

Our challenge then, is to build a master/worker template that gracefully tolerates the failure of the central manager, the failure of the master process, and network partitions among cooperating processes. Instead of having the central manager logging the application progress and requiring the user to manually restart the application when the master fails, we would like the tool to continue execution in the face the master failure. In addition, we also want to avoid depending on one single central manager to provide fault tolerance for an entire application. Instead of potentially wasting resources during network partitions, we would like the tool to continue to make progress independently in different network partitions and to propagate information to necessary processes once the partitions heal. The steps taken to acheive our goals are described in more detail in Section 3.5.

3.4 Implementation Constraints

Through studying Condor and the M/W template, we have reached various decisions on the design of our new tool (denoted as M/W' for clarity). For instance, we have decide to continue to submit worker processes in the vanilla universe despite the lack of control over these workers by the master or the user. We have also decided to replicate the master in the M/W tool to provide additional fault tolerance, despite the potential cost of replication. The following two sub-sections illustrate some of the reasons that prompted these design decisions.

3.4.1 Constraints in Condor

As mentioned in previous sections, jobs in different universes have different capabilities that can be advantageous but can also be inconvenient. The implementations of both M/W and M/W' take advantage of the fact that Condor's central manager monitors jobs created in the vanilla universe and restarts them when necessary. By creating several workers in the vanilla universe, the tool automatically tolerates worker failures. However, with worker processes in the vanilla universe, there is no guarantee that these processes would stay on one machine for the duration of the job. Thus, it would be nearly impossible for workers to communicate with each other short of flooding messages across the system. This indicates that message logging techniques such as Lazy Active Replication [7] would be quite difficult to implement without modifying Condor itself. While it is possible for each worker to log messages and checkpoint files in memory and can conceivably become the master should the original master fail, the M/W system would still break down when this worker becomes a master. This is because this worker-turned-master is still in the vanilla universe and may be migrated when needed, causing problems when other processes attempt to communicate with it. Thus, to provide fault tolerance to master processes, our decision is to simply replicate masters, and to run them all in the scheduler universe. Having decided to do this, we implemented new communication mechanisms for replicated masters to communicate with each other. We also created a script, which is essentially a simple program, to automatically submit multiple master processes and facilitate the communication links set up among these masters.

3.4.2 Practical Constraints

One other reason we decided to replicate masters, in spite of the potential overhead of replication, is the status of the existing tool. Given the popularity of Condor and the M/W tool, it seemed likely that a new tool would not be accepted by current users if they needed to modify existing code to use the new tool. Therefore, we opted to make M/W' compatible with M/W and thereby allowing casual users to reuse their original code. Thus, in M/W', we retained the original templates, because they contain interfaces to application-level code. Hence, behaviors of the masters do not change dramatically. For instance, masters divide a job into small tasks as before; they also send and receive tasks and results as before.

3.5 Implementation

M/W' is divided into three sections—script, master, and worker. The script is a simple program used to set up the masters by submitting master processes to specific machines. It also coordinates communication among these master processes. It acts as the central controlling point to synchronize these masters so that the masters can take turns connecting to one another. The master section is extended from the original master template to implement replication of masters. This new master template handles merging and disseminating of information among masters. The worker section is also extended from the original worker. It has an additional ability to communicate with more than one master. The following sections describe each of these components in turn and also highlight a pivotal piece of information used in our tool: the *IP address list*.

3.5.1 Script

As described previously, we decided to tolerate master failures through replication. To comply with M/W, M/W' simply creates several masters on different machines. M/W' is built to take advantage of a system of multiple Condor pools flocked together. We assume that each Condor pool consists of machines in one local area network, and these machines do not partition away from each other easily. However, the different Condor pools can partition away from each other frequently because they are dispersed across a wide area network. To tolerate different Condor pools partitioning away from each other, we require that there is at least one master in each relevant Condor pool such that when they do partition away from each other, each group of master and its workers, workers created by the local master, can continue to process the job independently. Each of these replicated masters subsequently creates its group of workers upon starting. Instead of requiring a user to submit these replicated master jobs manually, we have created a simple program called the "script" to automate this process [1]. Figure 3.4 gives the script in pseudocode. It is actually written in C++ and is 409 lines long.

This script program, shown in pseudocode in Figure 3.4, is started with the name of an input file as the simple argument. This input file contains the total number of masters to be started, a port number at which the script accepts connections from masters, the remote machines to submit the master jobs, and their corresponding submit description files. Based on this information, the script program submits the replicated masters on the remote machines specified. An example of the input file to the script program is shown in Figure 3.5.

The submit description files in this case are similar to that in Figure 3.1 with additional arguments. For M/W', we require the user to provide some mandatory arguments in each submit description file (Figure 3.6). Let m be the master that is being created with the submit description file shown in Figure 3.6. The required arguments

[1] In our current implementation, although the script submits master processes automatically, it does need help from the user to choose appropriate machines upon which the processes should be submitted. It would be simple to provide a service to acquire such system information automatically in the future.

```
begin script
   numMasters = read in first integer;
   port = read in second integer;
   create socket on port "port";
   set socket to listen for connections;

   for (int i = 0; i < numMasters; i++)
      remoteMachine = read in the remote machine;
      submitDescFile = read in the name of the submit description file;

      //Submit the job, master i, to the specified remote machine.
      //This is done using the -r option in the condor_submit command.
      system ("condor_submit -r remoteMachine submitDescFile");
   end for

   //Help the masters connect to other masters.
   //(This is needed because the masters are not synchronized.)
   for (int i = 0; i < numMasters; i++)
      tell master i to listen for connection requests from other masters;
      for (int j = 0; j < numMasters; j++)
         if (j != i)
            tell master j to connect to master i;
         end if
      end for
   end for
end script
```

Figure 3.4: Sample script program.

are as follows:

1. the unique ID of the master, a number assigned by the user.
2. the total number of masters in this system,
3. the IP address of the machine running the script program,
4. the port number the script opens to accept connections,
5. the total number of IP addresses listed in the following IP address list,

6. a list of IP addresses of machines running the master processes,

7. the list of unique IDs corresponding to each of the masters running on machines listed in the previous IP addresses,

8. the length of time a worker should wait for a response from a master before deciding its master has failed or partitioned away,

9. the length of time a master waits for a response from workers before going on to check for responses from other masters,

10. the total number of masters that m should listen to for information,

11. the target number of workers, and

12. the first worker unique ID of m's workers.

```
3
8992
beak.cs.wisc.edu submit_description_file0
ophelia.cs.wisc.edu submit_description_file1
blackbird.cs.wisc.edu submit_description_file2
```

Figure 3.5: Example of script input file.

Most of these arguments, such as IP addresses, master unique IDs, total number of master, number of replicated masters a particular master should listen to for messages, are used to facilitate communication among masters and workers, and will be discussed in more detail in Chapter 4. Arguments such as the length of time a master or worker waits for a response is a necessary addition in M/W'. In M/W, with only one master in the system, a worker waits on that master forever. In our version, however, a worker should decide to abandon an unresponsive master after a certain period of time because there are other sources of tasks to do. Similarly, in M/W, the master waits until a worker responds because it can only obtain information from workers. In M/W', there is a possibility that other masters, through *their* workers, have successfully received some results. Therefore, in M/W', a master is programmed to not wait on workers forever. In

```
getenv = True
universe = Scheduler
executable = master_process
arguments = 0 3 128.105.146.14 8992 3 128.105.167.49
128.105.146.14 128.105.143.15 0 1 2 30 5 2 10 0
input = input_matrix
output = product_matrix
error = error_log
log = work_log
notify_user = ichen@cs.ucsd.edu
queue
```

Figure 3.6: Example of Condor submit description file for the new M/W template.

fact, in the primary/backup protocol discussed in Chapter 4, it is essential that backup masters do not wait for workers forever, because under correct conditions, workers are programmed to only contact the primary master. The rest of the arguments, such as target number of workers and the first unique ID of a master's workers, are added for further modifications of communication protocols.

After submission of master jobs, the script acts as a coordinator to help replicated masters establish connection among themselves. As indicated in the nested for-loops in Figure 3.4, the script does not control how masters communicate with each other. It simply synchronizes masters such that the replicas can establish communication among themselves as they see fit[2].

3.5.2 Master

The master portion of M/W' can be roughly divided into three layers (Table 3.1). Similar to M/W, the top layer, which consists of application specific functions, is provided by the user. In fact, M/W' is built such that all the interfaces to the user-defined functions in M/W remain intact. Thus, existing user code can be used without change.

The bottom and the middle layers are called the *base communication layer*

[2]The communication links established among masters depend on the protocol these replicas use; this is explained in more detail in Section 3.5.2 and Chapter 4.

Table 3.1: Overview of the new master template.

Application Layer	User-defined functions, such as create_initial_tasks(), from the original master template; M/W library functions (code that calls such user-defined functions).
Protocol Layer	Customized functions that dictate how masters should communicate with other masters.
Base Communication Layer	Includes all of the communication functions in the original template and also contains new functions to facilitate communication among masters.

and the *protocol layer* respectively. The *base communication layer* is extended from M/W and is virtually unchanged. The ability to connect to and communicate with replicated masters is added. Other functions in this layer, such as basic communication functions between a master and a worker are left intact. The protocol layer is a new layer designed to handle information that is passed among masters. This layer is responsible for handling the merging of information from other masters and the transmission of information to other masters.

Figure 3.7 gives the pseudocode of our master template. The actual template is written in C++ and is 4756 lines long. The behavior of each replicated master is similar to that of the original master. In M/W', once started, each replicated master connects to the script program. With the help of the script, each master connects to other replicated masters listed in its master IP address list. Each master also listens for connections from other replicated masters. After this initiation stage, the masters negotiate among themselves to determine their task responsibilities, i.e., which master controls the execution of which tasks and in what order. This function is provided by the protocol layer. As in M/W, each master then creates several workers. The difference with M/W' is that each master passes a list of master IP addresses to its workers. This list may simply be a master's own list of master IP addresses or a variation of it[3]. Each master then loops similarly to that in M/W. Each master waits a certain amount of time

[3]This is discussed further in Chapter 4.

for a worker to contact it. When contacted, the master sends the worker a task. The master then waits to be contacted by the worker again to receive results for the task. One difference with a new master is that it has the ability to accept results from workers it did not create. Consequently, it also has the ability to accept results of a task which it did not know was distributed (to a worker.) Each master periodically sends out state information, such as the new results it obtained, to replicated masters on the master IP list, and listens for state information from other masters as well.

3.5.3 Worker

Figure 3.8 presents the pseudocode of our worker template. The template is written in C++ and is 899 lines long. The worker template is essentially unchanged from that in M/W. Each worker is created by a master, along with a list of replicated master IP addresses. After the initial administration setup procedures, each worker loops similarly to M/W. It contacts a master and receives a message from the master. The difference in this case is that each worker generally has more than one master to contact should it fail to receive a message from the master it contacted. It simply goes through the list of master IP addresses in order until a master is contacted successfully. When a message is received, it complies with the message, be it to terminate or to execute a task. When a worker completes a task, it again contacts a master to transmit the results. The critical innovation in this new template is the fact that a worker does not wait on one single master forever. Therefore, when a master receives a message from a worker, it is required to either comply with the request or send an ACK message. The worker in turn uses this fact that there should always be a message returned from a master within a reasonable amount of time to determine whether or not to go on to the next master.

3.5.4 Master IP Address List

Throughout the previous sections, one common theme that occurs repeatedly is the list of master IP addresses. It is important to note the significance of the lists of IP addresses passed on to the masters and workers. In addition to facilitating communication among processes, these lists can be manipulated to provide different replicated masters communication protocols and to facilitate load-balancing during master failures or network partitions. Further discussion on the IP address lists is provided in Chapter 4.

3.6 Summary

In this chapter, we provided an overview of the original template. We show that, based on the original tool, we are able to construct a fault-tolerant M/W template that is compatible with the old one. The following chapter discusses in more detail our design decisions and goal to provide a practical template that can accommodate different needs.

```
begin Master
   Read in the arguments;
   //E.g., store the number of masters that I should listen to.
   numRepMasters = argv[x];
   Based on the arguments I received, prepare a copy of master IP list
      for me and another copy for my workers.
   connect() to script;
   loop
      recv() a message from script;
      if message tag is CONNECT
         int rMaster = unpack an int from buffer;
         if master 'rMaster' is in my master IP list, connect() to it.
         send() CONNECT_DONE message to script;
      else if message tag is LISTEN
         for (int i = 0; i < numRepMasters; i++)
            accept();
         end for
         send() LISTEN_DONE message to script;
      else if message tag is DONE
         break;
      end for
   end loop
   initial setups;
   create_initial_tasks();
   sched_pushTodoTaskList ();    //used to schedule tasks.
   system ("condor_submit worker_processes");
   begin loop
      if (sched_toExit())
         break;
      sched_getMoreTasks();
      try to recv() a message from a worker and update my status;

      //The body of the loop is similar to that in Figure 3.2.
      //The difference is that if the master receives a RESULTS message
      //from a worker, it sends out that RESULTS message to all the
      //masters in its master IP list.

      //Check messages from other masters.
      merge_mergeInfo();
   end loop
   terminate workers;
end Master
```

Figure 3.7: Sample new master template.

```
begin Worker

   connect() to the master and send initial information;
   send a REQUEST_TASK message to a master;

   begin loop

      recv() a message;

      if message tag is DO_THIS_WORK
         unpack data in this message;
         execute_task ();
         pack results into buffer;
         send a RESULTS message a master;
         loop until receives an ACK
            if no ACK
               send the RESULTS message to another master;
            end if
         end loop
      else if message tag is KILL_YOURSELF
         terminates;
      end if

   end loop

end Worker
```

Figure 3.8: Sample new worker template.

4

Separation of Mechanism and Policy

There are many ways to organize replicated masters, with different configurations performing better in different situations. This led to our decision to separate mechanism from policy, so as to allow users to tailor our tool for specific needs imposed on their applications and execution environments. In this chapter, we classify our strategy into three policies, which are listed in Table 4.1. For each policy, we provide a brief description of the rationale behind it and our implementation of it. We also provide examples from two working protocols we have implemented. We call these two protocols the *primary/backup* masters and the *parallel* masters.

4.1 Sample Protocols

Using the template we built, we have successfully created two very different protocols to configure the replicated masters. Functions we have written for these two protocols are used throughout this chapter to further explain our template. The following sections introduce these two protocols. When necessary, we use M_i to denote the master with unique ID i, IP address IP_{M_i}, and master IP list $MIPList_i$. The IP address list that M_i passes to its workers is denoted as $WIPList_i$. When relevant, master zero (M_0) is the primary master.

4.1.1 Primary/Backup Masters

Our primary/backup protocol is designed to be deployed on a local area network in which machines rarely partition away from each other. Our goal for the primary/backup protocol is to build a protocol that is centered at the primary master, but can still tolerate primary master failure and backup master failure. Albeit rare in a local area network, this protocol also tolerates network partitions between any two masters. This protocol is designed such that the primary master is deployed on a reliable machine that is well-connected to all the backup masters. We designed the backup masters to play a supporting role in the application that ensures that the primary master is as well-informed as possible, in terms of results of tasks. As the name suggests, all processes, both worker and master processes, identify at any time one master, the master with unique ID zero, as being the primary master. In fault-free conditions, the primary master handles all tasks, distributing tasks and receiving results from workers it created as well as workers the backup masters created, and propagates the results to the backup masters. In the case in which a backup master, b, partitions away from the primary master, p, other backup masters that detect this network partition forwards information between the two masters, p and b, to ensure that p receives as much information as possible. Backup masters are also responsible for re-sending any old results the primary master missed to the primary master upon request. In the event that the primary master fails, our application continues to make progress and completes the job. In the case of a crashed backup master, the job can still be completed by the workers and the remaining masters. Because the backup masters are created to support the primary master, the primary master terminates the backup masters once it has received all results.

4.1.2 Parallel Masters

Unlike our primary/backup protocol, there is no discernible primary master in the parallel masters protocol. This protocol is suitable in cases where cooperating machines frequently suffer from disconnections. In this case, we require each master to act as the primary master to its group of workers. Each master is also designed to terminate only after receiving all task results. As described in Section 3.5, each master creates its group of workers. We assume that the workers created by any particular

Table 4.1: Overview of our three policies[1].

Policy	Description	Functions Used
System View	Facilitates communication among masters and workers. Well-designed algorithms may also provide load-balancing in the event of failures.	view_createWMasterIPList() view_updateMasterState() view_updateMasterView()
Scheduling	Determines the orders in which tasks are scheduled. An additional function to determine the termination of a master is also provided to allow user more flexibility in programming the degree of cooperation among masters.	sched_pushTodoTaskList() sched_getMoreTasks() sched_toExit()
Merging	Handles information sharing among masters.	merge_propagateResults() merge_mergeInfo()

master are in the same Condor pool, and hence, the same local area network, as that master. Such a group of workers identify their creating master as their primary master during failure-free executions. Upon receipt of a task result, each master propagates the results to all other masters. In the event of a master failure, workers contact other masters in a pre-determined order that best suits the network topology, and thus, can be tuned for network partitioning. Since each master is contacted by its workers for work and to return results, and each master forwards all results to all other masters, information flows among all the masters. Should a master partition away from another master but remain connected to other masters, it still receives information. Therefore, with no extra precautions, we expect the application to carry on smoothly with little duplicated work in the event of network partitions. In case of master failures, performance of this protocol may be slightly worse than that of the primary/backup protocol. Since every master in this protocol is actively involved with workers, the failure of any master affects some workers, which results in wasted time while these workers search for another master.

4.2 System View

4.2.1 Policy

Although our decision in Section 2.3.1 was not to employ any group membership protocol to dynamically provide processes in this tool a view of the system, each instance of a M/W job is started with an initial view of the system. Each master has knowledge of the total number of masters in the system, and the IP addresses and unique IDs of each replicated masters. Some or all of this information is passed to worker processes. This information can be manipulated in certain ways to provide specific views of the system, and thus, offers different communication patterns among masters and between masters and workers.

4.2.2 Mechanism

To this end, the template initially requires the IP addresses of some or all masters and their respective unique IDs. This information is made available to the program as arguments. Listed in Table 4.2, the function, `view_createWMasterIPList()`, converts this information into a list of masters to which a particular master propagates information. Alternate lists of master IP addresses are also created to pass on to the worker processes. These lists of IP addresses give masters and workers a view of the available masters.

We also provide two additional functions, `view_updateMasterState()` and `view_updateMasterView()`, to allow for greater latitude in manipulating the behaviors of workers and masters. `view_updateMasterState()` is provided to change the state of a master. This is needed for protocols that have both active and dormant masters in which a dormant master may become active during execution. `view_updateMasterView()` is designed to prompt workers to update the knowledge of the available masters.

The knowledge of available replicated masters is important because both masters and workers contact masters sequentially according to the list of replicated masters they have. By customizing `view_createWMasterIPList()` and providing certain IP ad-

[1] The function names given in this table have been modified from those in the actual program to improve presentation and for clarity.

Table 4.2: New functions.

Function	Description
view_createWMasterIPList()	This is called by a master to create a list of IP addresses to be given to its workers.
view_updateMasterState()	This is called when a master receives a message, either from a worker or another master. It is used to update the system view of the master.
view_updateMasterView()	This is called when master determines that a certain worker should change its system view, usually in response to a change of network connectivity.
sched_pushTodoTaskList()	This is called to create the initial `todo` task list, which is a list consisting of tasks a master is scheduled to process when a worker contacts.
sched_getMoreTasks()	This is called when there are no more tasks in the `todo` task list. It retrieves more tasks from the complete task list.
sched_toExit()	This is called to determine the termination of a master. Depending on the communication protocol used, each master can terminate under different conditions, as programmed by the user.
merge_propagateResults()	This is called when a master receives a result from a worker. The master uses this to share this result with other masters in the master IP list.
merge_mergeInfo()	This is called to receive results from other masters.

dresses as arguments, users can influence the behavior under failures. With the help of `view_updateMasterState()` and `view_updateMasterView()`, the dynamics of network communication can be captured in a timely manner. We illustrate the usage of these functions in the examples below.

4.2.3 Examples

4.2.3.1 Primary/Backup Masters

For the primary/backup masters protocol, all processes, both masters and workers, have the same view of the system. There is one single primary master, M_0, at the start of execution whose identity is known by all masters. The order in which a backup master becomes a primary master is also initially agreed upon: the available master with the lowest unique ID becomes the primary master. Thus, the IP address lists given to each master is identical. They consists of IP addresses of M_0, M_1, M_2, and so on in the order of the unique IDs. Each master subsequently passes this list on to its workers. With this single version of IP address list, all attempts of communication with a master begin with M_0. If communication were unsuccessful, a second attempt of communication is made to M_1 and so on. Using this scheme, we have essentially created a primary/backup application.

As mentioned in Chapter 3, these master IP lists are passed to the masters using the submit description file. For example, a two-master primary/backup application based on our template, with masters on `ophelia.cs.wisc.edu` (IP: 128.105.167.49) (primary) and `beak.cs.wisc.edu` (IP: 128.105.146.14), can be started with the two submit description files shown in Figure 4.1 and Figure 4.2. Note that the only difference between the two files is the first parameter in the arguments line. In Figure 4.1, the first parameter in the arguments is zero, indicating that the master process created using this submit description file has unique ID zero. On the other hand, the master process created using Figure 4.2 has unique ID one. The figures show that each master IP list consists of two IP addresses, `128.105.167.49` followed by `128.105.146.14`. Furthermore, since masters pass their lists in their entirety to their worker, all the workers have this same list. Consequently, the function `view_createWMasterIPList()` simply passes on a master's IP list to its workers. However, since the list of IP addresses given to a master also indicates the replicated masters each master should connect to when a CONNECT message is received (Figure 3.7), the master IP list each master retains is the original list without its own IP address to avoid a master connecting to itself.

```
  getenv = True
  universe = Scheduler
  executable = master_process
  arguments = 0 2 128.105.146.14 8992 2 128.105.167.49
128.105.146.14 0 1 30 5 1 10 0
  input = input_matrix
  output = product_matrix
  error = error_log
  log = work_log
  notify_user = ichen@cs.ucsd.edu
  queue
```

Figure 4.1: Example of Condor submit description file for ophelia in a two primary/backup masters system.

```
  getenv = True
  universe = Scheduler
  executable = master_process
  arguments = 1 2 128.105.146.14 8992 2 128.105.167.49
128.105.146.14 0 1 30 5 1 10 0
  input = input_matrix
  output = product_matrix
  error = error_log
  log = work_log
  notify_user = ichen@cs.ucsd.edu
  queue
```

Figure 4.2: Example of Condor submit description file for beak in a two primary/backup masters system.

With information disseminated only by the primary master, this protocol is not tuned to handle network partitions between a backup master and the primary master. However, [2] shows that in a group of strongly connected machines, it is possible for these machines to continue to communicate with each other in the face of one broken link among them because of path redundancy. To incorporate this idea to tune our primary/backup masters protocol for network partitions, we use the function `view_updateMasterState()`, illustrated in pseudocode in Figure 4.3. This function is called when a master receives a message. Each backup master normally receives messages only from the primary master. When a backup master receives a message from a worker or another backup master, it indicates that there may be a link failure between a backup master and the primary master. In this case, instead of remaining dormant, the backup master takes up the responsibility of the primary master and forwards messages between the two non-communicating masters. For example, Figure 4.4 illustrates the actions taken by the replicas in the event that M_0 partitions away from M_1. In this case, workers created by M_1 attempts to contact the next master in the list, M_1. When M_1 receives a message from a worker, it decides that it has temporarily become the primary master for that particular worker. Hence, the state of M_1 is now in the primary master state. M_1 begins distributing tasks to its workers and receiving results from them. Upon receipt of results from its workers, M_1 propagates them to all other masters, M_0 and M_2. Since M_1 has partitioned away from M_0, only M_2 receives results from M_1 directly. Upon receipt of results from M_1, M_2 determines that M_1 has partitioned away from M_0 and also changes its state to begin forwarding results between M_0 and M_1.

In the event that M_0 did not fail but simply partitioned away from M_1, the state of M_1 would revert back to backup master state once the partition ceased. Thus, when M_1 detects the presence of M_0 by receiving a message from M_0, M_1 changes its state back to the backup master status and informs all other backup masters of this change by calling `view_updateMasterView()`. The function `view_updateMasterView()` (Figure 4.5) is called by M_1 at this point to send a message to its workers informing them about the presence of the primary master M_0.

Clearly, this implementation of the two functions, `view_updateMasterState()` and `view_updateMasterView()`, together with the initial master information, allows an application programmer to structure a general M/W template into a primary/backup application that not only tolerates master failures but is also tuned to tolerate network partitions.

```
begin view_updateMasterState
   //myid is the unique ID of the calling master.

   if contacted by master i's worker where i != myid
      decide to become master i's primary master;
   if contacted by a worker created by the calling master
      decide to become a primary master;
   else if received results from master i where i != 0
      //for the purpose of forwarding results
      decide to become master i's primary master;
   else if received results from master 0 (i.e., primary)
      //this indicates connection with primary master exists
      if currently in primary master state
         inform workers to begin contacting master 0;
         inform all other backup masters to stop being
    forwarding results from master 0;
      end if
   else if received request from master i where i != 0
         stop forwarding results from master 0
      stop forwarding results to master i;
   end if

end view_updateMasterState
```

Figure 4.3: Sample primary/backup view_updateMasterState() code.

4.2.3.2 Parallel Masters

With the parallel masters protocol, our goal is to build a protocol that performs well when cooperating machines frequently partition away from each other. The design is based on the assumption that when a master creates its set of workers, Condor allocates these workers to machines in the same local area network as the master. This is because we assume that each Condor pool consists of machines in the same local area network. Thus, when a master submits worker processes, it contacts the central manager of its pool. This central manager then allocates machines in the local area network to run worker processes. With this assumption and the goal to specifically tolerate network partitions, we can manipulate the argument IP address lists to provide masters and

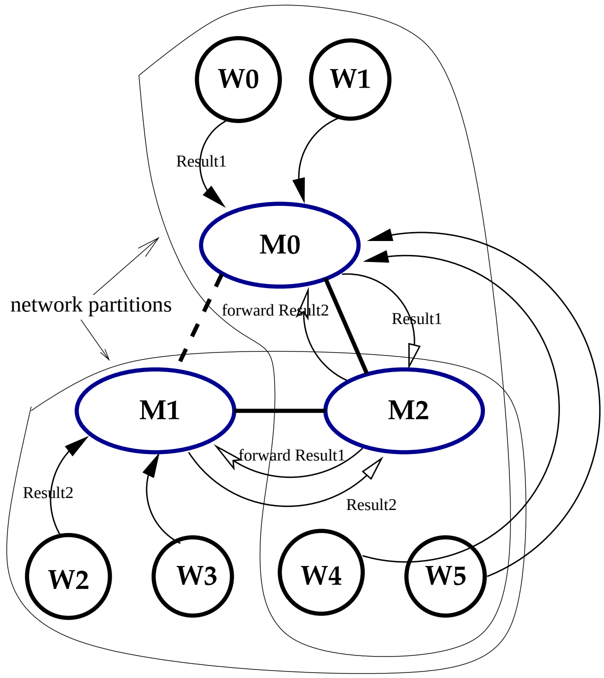

Figure 4.4: Primary/backup masters partition example

```
begin view_updateMasterView

   send each worker created by the calling master a message
      informing it the presence of master zero, and that the
      next time it communicates, it begins contacting the
      masters from the beginning of its master IP list until
      it finds an available master.

end view_updateMasterView
```

Figure 4.5: Sample primary/backup view_updateMasterView() code.

workers a view of the actual networks.

Instead of assigning an arbitrary master to be the primary master to all processes, primary masters are assigned in terms of networking distances. For example, we provide M_0 with $MIPList_0 = list(IP_{M_0}, IP_{M_1})$. On the other hand, M_1 has $MIPList_1 = list(IP_{M_1}, IP_{M_0})$. Similarly, $WIPList_0$ has the value $list(IP_{M_0}, IP_{M_1})$

and $WIPList_1$ has the value $list(IP_{M_1}, IP_{M_0})$. M_0 is seen as the primary master by workers in its local area network and M_1 is also seen as the primary master by workers in *its* local area network.

In our implementation, to start a parallel masters application with masters on `ophelia.cs.wisc.edu` (IP: 128.105.167.49) and `beak.cs.wisc.edu` (IP: 128.105.146.14), one can use the submit description files shown in Figure 4.6 and Figure 4.7. Each master is given the IP address of the other master. To prepare a complete list of master IP addresses for its workers in the function `view_createWMasterIPList()`, each master inserts its own IP address into the list (Figure 4.8). With this simple change in master IP address lists, we have effectively provided a new protocol that is naturally attuned to the networking environment.

```
getenv = True
universe = Scheduler
executable = master_process
arguments = 0 2 128.105.146.14 8992 1 128.105.146.14
1 30 5 1 10 0
input = input_matrix
output = product_matrix
error = error_log
log = work_log
notify_user = ichen@cs.ucsd.edu
queue
```

Figure 4.6: Example of Condor submit description file for ophelia in a two parallel masters system.

Unlike the primary/backup masters protocol, each parallel master sends results to all other masters. Given a broken link between any two masters, all the masters continue to receive new information without much loss. Thus, we have elected not to require each parallel master to keep track of the conditions of other masters and do not provide `view_updateMasterState()` and `view_updateMasterView()`.

```
getenv = True
universe = Scheduler
executable = master_process
arguments = 1 2 128.105.146.14 8992 1 128.105.167.49
0 30 5 2 10 0
input = input_matrix
output = product_matrix
error = error_log
log = work_log
notify_user = ichen@cs.ucsd.edu
queue
```

Figure 4.7: Example of Condor submit description file for beak in a two parallel masters system.

```
begin view_createWMasterIPList

   wIPListPtr = a copy of mIPListPtr;
   prepend my IP address in to wIPListPtr;

end view_createWMasterIPList
```

Figure 4.8: Sample parallel master view_createWMasterIPList() code.

4.3 Scheduling

4.3.1 Policy

To complement the flexibility in communication described above, our tool also provides mechanisms for users to customize the scheduling of tasks. With well-scheduled tasks, an application can be both efficient and reliable. An M/W application begins with a fixed number of tasks known to all masters. Instead of simply scheduling all tasks among all masters, one can take advantage of the replication of masters and delegation of tasks to customize the tool to provide a certain degree of efficiency and fault tolerance.

4.3.2 Mechanism

We provide the function `sched_pushTodoTaskList()` for users to customize masters' `todo` task list. Each master has its own copy of `todo` task list. It is a list of tasks that a master should complete before attempting other tasks. When contacted by a worker for a task, the master retrieves the first task in its `todo` task list if the list is not empty. This `todo` task list can contain the complete list of tasks of the job or it can be half of the complete list of tasks in the reverse order, etc. We also provide an additional function, `sched_getMoreTasks()`, to offer more flexibility. This function is called when there are no more tasks in the `todo` task list. `sched_getMoreTasks()` attempts to retrieve unfinished tasks and places them in the `todo` task list. It can be crafted to balance the efficiency and redundancy of an application. To provide closer cooperation among masters, a user can use the function `sched_toExit()` to specify when a master terminates.

4.3.3 Examples

We continue the examples started in Section 4.2.3.

4.3.3.1 Primary/Backup Masters

Since our tool is primarily used to compute tasks that can be processed in parallel, we assume that merging these task results is relatively easy. Because our primary/backup protocol is tuned for master failures, we simply place all tasks in each master's `todo` list, expecting fault-free masters will have to handle all the tasks, and thus, doing away with function `sched_getMoreTasks()`. To reduce redundancy in case of network partitions, we use a simple scheduling scheme to somewhat randomize the order of the tasks in each master's `todo` list. Figure 4.9 shows the pseudocode we used, which works well in systems with as many as three masters. Because the backup masters are created to support the primary master, we require that each backup master remain processing until it is specifically told by the primary master to terminate, whereas the primary master can terminate after it has received all results and informed the backup masters to terminate. (If the backup master has received all results but has not received a termination message from the primary master, it terminates after a timeout period, which is currently set at 30 minute.) Thus, the function `sched_toExit()` prevents a backup master from terminating but allows the primary master to terminate when it has completed the job.

```
begin sched_pushTodoTaskList

   for each task
      if (myid % 4 == 0)
         append task to the list;
      else if (myid % 4 == 1)
         prepend to the list;
      else if (myid % 4 == 2)
         prepend odd number tasks to the list;
         append even number tasks to the list;
      else
         append odd number tasks to the list;
         prepend even number tasks to the list;
      end if
   end for

end sched_pushTodoTaskList
```

Figure 4.9: Sample primary/backup sched_pushTodoTaskList() code.

4.3.3.2 Parallel Masters

In our parallel masters protocol, designed to mainly tolerate network partitions, each master begins working on different tasks (Figure 4.10). In perfect conditions, masters process jobs in parallel, removing the need for all masters to attempt all tasks. In case of partitions, this is done in the hope that by the time each master finishes its portion of tasks, the partitions (if existing) would have ceased and the masters could simply merge their results and finish the job with little redundancy. Should a master have remaining undone tasks in the complete task list after finishing tasks in its `todo` list, it can call `sched_getMoreTasks()` to retrieve the remaining tasks. This function can simply refer `todo` to the remainder of the complete task list. Because this protocol is designed to be used by a group of machines that partition away from each other frequently, each master is designed to be responsible only for itself. Thus, to determine the termination status, each master calls `sched_toExit()`, which simply decides to terminate when all tasks are finished.

```
begin sched_pushTodoTaskList

   m = total number of masters;
   t = total number of tasks;
   firstTaskNumber = myid * (t/m);
   lastTaskNumber = firstTaskNumber + (t/m);
   if ((myid + 1) == m)   //assuming master IDs start at 0
      //I pick up the last of the tasks.
      lastTaskNumber = t%m + lastTaskNumber;
   for each task
      if task number is within my firstTaskNumber and my
         lastTaskNumber
         append the task to todo;
      end if
   end for

end sched_pushTodoTaskList
```

Figure 4.10: Sample parallel sched_pushTodoTaskList() code.

4.4 Merging

4.4.1 Policy

Finally, to further support replicated masters, we provide mechanisms for the masters to propagate and merge information. To tolerate master failures, it makes sense to propagate task results from a particular master to all masters whenever possible. In terms of tolerating partitions, it is also sensible to make as much information available to as many masters as possible because work may be saved with additional knowledge. With the extra information passing among masters, there is also a need for a method to merge and manage this information.

4.4.2 Mechanism

We provide two functions, `merge_propagateResults()` and `merge_mergeInfo()`, to facilitate the distribution and merging of information. Based on the nature of our M/W tool in which tasks are independent and can be done in parallel, and our decision not to provide dynamic system views during the execution of a job, the only state information that needs to be merged is task results. Therefore, these two functions are mainly used to propagate and merge task results. Similar to the mechanisms we provided for scheduling and determining system views, these two functions can be customized to fit different degrees of fault-tolerance and efficiency.

4.4.3 Examples

4.4.3.1 Primary/Backup Masters

In the process of coding this tool and implementing the two protocols, we found that it is a good strategy to always send new task results to other masters. Thus, `merge_propagateResults()`, called by a master immediately after receiving a task result from a worker, sends results to all the masters (Figure 4.11).

As mentioned briefly in Section 4.2.3.1, to reduce the reliance on communication links to the primary master, we designed this protocol such that backup masters forward information to other masters when necessary. Thus, in `merge_mergeInfo()` (Figure 4.12), each master not only checks for results from other masters, it also forwards those results when necessary.

4.4.3.2 Parallel Masters

As mentioned in the previous section, we have decided that it is good practice to always send results a master received directly from a worker to all other masters in the master IP list. Hence, the function `merge_propagateResults()` for this protocol is identical to that in the primary/backup protocol, as shown in Figure 4.11.

As for the merging results from other masters, we have decided to keep it simple for this protocol since under normal circumstances, all masters are processing tasks and sending results to all other masters. The `merge_mergeInfo()` function (Figure 4.13)

```
begin merge_propagateResults

   for each master listed in my master IP list
      send the results received from worker to the master;
   end for

end merge_propagateResults
```

Figure 4.11: Sample merge_propagateResults() code for both primary/backup and parallel masters protocols.

```
begin merge_mergeInfo

   for each master listed in the master IP list
      check to see there are results messages from it;
      if there are,
         handle the results;
         update all the task lists;
      end if
      if there are results from master 0, the primary master,
         send these results to each master not in contact
            contact with the primary master;
      else if there are results from master i, i != 0
         forward these results to master 0 if necessary;
      end if
   end for
   send old results to the primary master upon request;

end merge_mergeInfo
```

Figure 4.12: Sample primary/backup merge_mergeInfo() code.

simply accepts results from other masters.

```
begin merge_mergeInfo

   for each master listed in the master IP list
      check to see if there are new results;
      if there are,
         handle the results;
         update all the task lists;
      end if
   end for
   re-send old results when requested;

end merge_mergeInfo
```

Figure 4.13: Sample parallel merge_mergeInfo() code.

4.5 Summary

In this chapter, we have shown the power of templates and separation of mechanism and policy through our implementation of two very different protocols. We showed that we were able to build these two protocols using only a few simple custom-built functions and certain system information. Together with Chapter 3, we described the entire design of our partition-aware M/W template. We have shown it to be compatible with the original M/W template and we have also shown its flexibility through our examples.

5

Evaluation

In this chapter, we provide some observations on Condor that we made when testing our tool. We also provide the behavior and performance of our primary/backup protocol for three specific situations.

5.1 Submitting Jobs in Condor

To test our tool, we first created a M/W application in which each task waits 120 seconds before contacting a master to return results. Thus, the expected duration of a job consisting of 15 such tasks is easy to calculate. However, when running such an application on Condor, where a job begins getting processed only after a worker process is allocated a machine, we discovered that the duration of the job is much longer than expected. Thus, we built a simple test program that submits ten jobs on the Condor pool at University of Wisconsin, Madison. Each job, once it has started processing on a machine, connects to the test program that started it and waits for a termination signal from that program. The test program records the time between the submission of the ten Condor jobs and the time it is contacted by each job. This test program terminates the ten jobs as soon as all ten jobs report in. Figure 5.1 shows the results of this experiment run ten times. We see in this plot the long lengths of time it takes to allocate ten suitable idle machines and the wide range of starting time of each job, particularly the first job of each run. We also plotted the time between the start of each job in Figure 5.2. Figure 5.3 shows the line plots from Figure 5.2 without run four and run five. Again, we see variations in the times between the start of each job although the three figures indicate that for each run, the latter nine jobs are usually assigned machines shortly after the first job has been assigned a machine. We suspect Condor may be using a work conserving algorithm to allocate machines to jobs. At this point, we have sent the data we gathered to the Condor developers for clarification but have yet to receive an explanation.

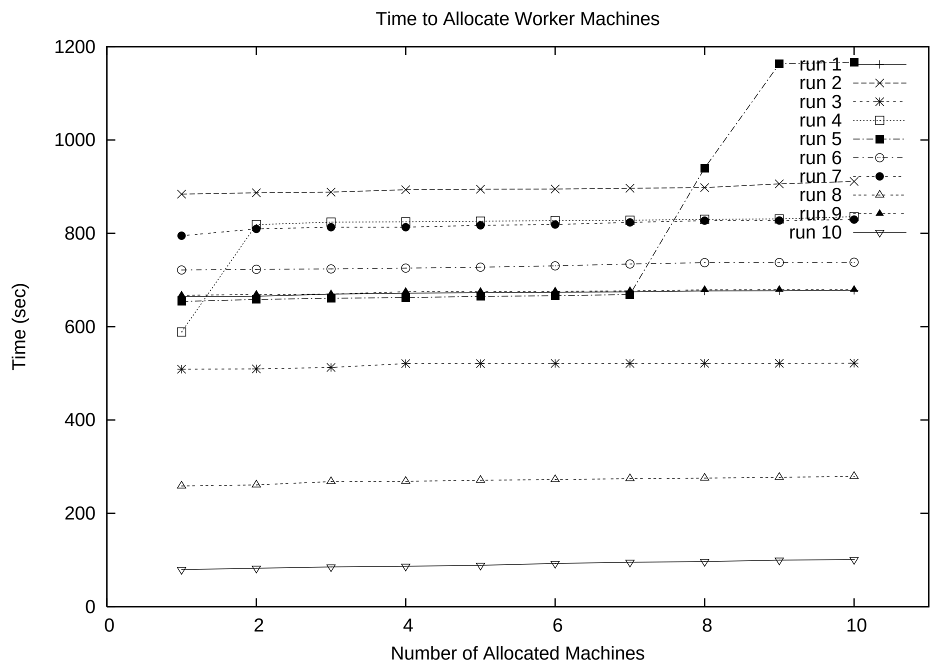

Figure 5.1: Plot of time it took for Condor to allocate ten machines for ten jobs.

5.2 Performance

As mentioned in the previous section, to test the performance of our template, we used a sample application which consists of 15 tasks, each of which consumes 120 seconds. This allows us to calculate the exact expected processing time of a job. As mentioned in the previous section, we observed that there is an overhead involving the allocation of machines in Condor. Thus, to test the performance of our tool, we have decided to record the duration of a job, which is defined to be the start of a master until the master terminates, and the processing time of a job, which is the time between the

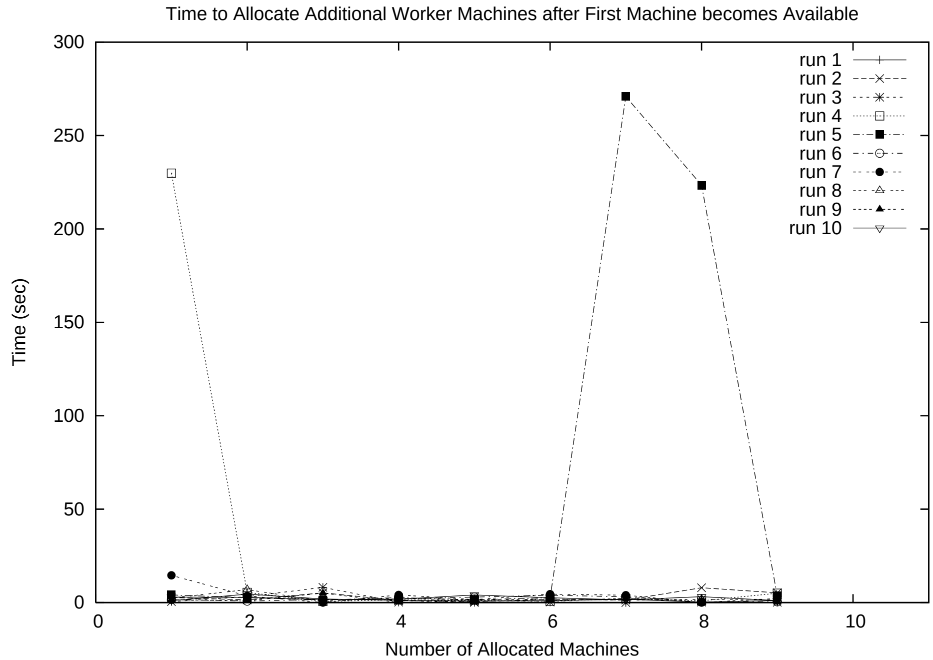

Figure 5.2: Plot of time between contact from one job to the next.

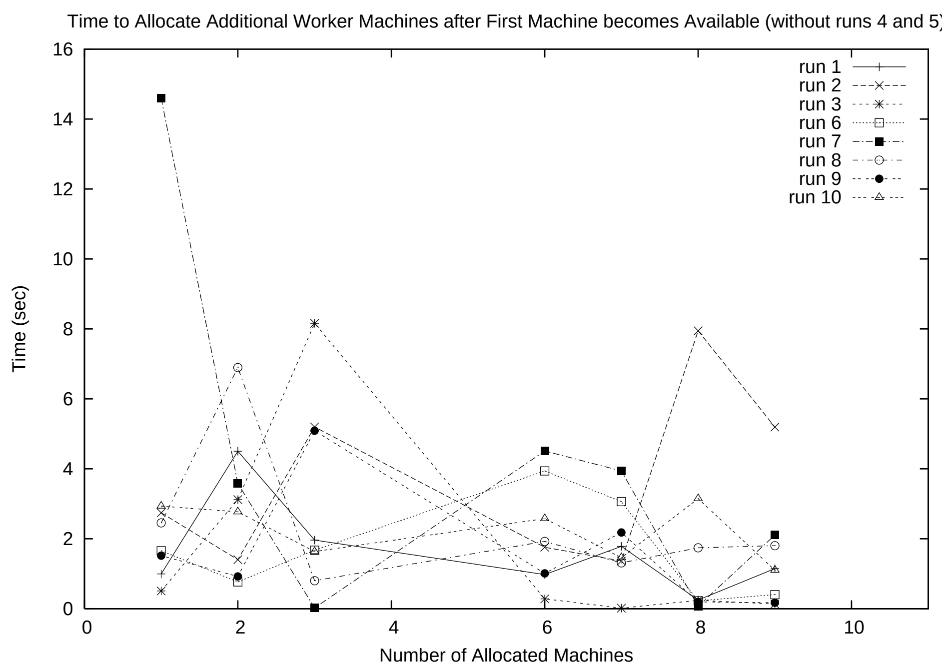

Figure 5.3: Plot of time between contact from one job to the next without run four and run five.

start of all worker processes on idle machines and the termination of a master. In this section, we show the performance and behavior of the primary/backup masters protocol. We picked three particular situations:

1. two masters in failure-free situation,

2. two masters initially partitioned away from each other but were able to communicate with each other after one of the masters received results from five tasks, and

3. three masters, denoted as M_0, M_1, and M_2 as described in Section 4.1, in which M_1 is partitioned away from M_0 for the duration of the job.

In the cases with two masters, we placed the primary master, M_0, on machine `mystere.ucsd.edu` in the Condor pool at the GRAIL at UCSD and the backup master, M_1, on `ophelia.cs.wisc.edu` in the University of Wisconsin, Madison Condor pool. In the case with three masters, we run the second backup master, M_2, on `beak.cs.wisc.edu` in the Condor pool at Wisconsin. The replicated masters each creates two workers in their respective Condor pools. We performed ten runs for each situation and report the average duration, processing time, number of results a master received from workers, denoted as number of local results, number of results a master received from another master, denoted as number of external results, and the total number of results a master received, which may include duplicates, in Table 5.1.

Table 5.1: Performance and behaviors of three test cases for primary/backup masters.

Master	Number of Workers Created	Duration (sec)	Error of Duration at 95% Conf. Lev	Processing Time (sec)	Expected Processing Time (sec)	Error of Proc Time at 95% Conf. Lev.	Number Local Results	Number External Results	Total Number Results
2 masters—fault-free									
M0	2	746.7690	103.0326	482.3720	480	0.752789	15	0	15
M1	2	747.7124	102.6890	479.8009	480	1.034220	0	15	15
2 masters— partitioned away from each other initially, partitions cease after a master receives 5 results									
M0	2	829.8953	137.1914	515.0557	510	1.263724	9	6	15
M1	2	832.6158	137.0459	513.3623	510	1.737202	6	5	11
3 masters— M0 partitioned away from M1									
M0	2	587.4344	110.9002	363.6658	360	0.696637	11	4	15
M1	2	598.3408	110.7957	369.6793	360	1.118856	4	7	11
M2	2	589.7910	110.4462	362.6799	360	0.911657	0	15	15

In the failure free case, we expect to see the four workers working directly with the primary master, M_0, in parallel. Thus, we expect to see M_0 receiving all its results directly from workers and M_1 receiving all its results from M_0. Since the four workers are working in parallel, and there are 15 tasks each taking 120 seconds to complete, the expected processing time of a job is $\lceil \frac{15}{4} \rceil * 120 = 480$ seconds. We expect the duration of the job to be larger than this number, but as mentioned in the previous section, we cannot predict the duration of the job. In Table 5.1, we see that M_0 indeed received all results directly from workers and M_1 received all results directly from another master, which in this case is M_0. We also see that the processing times for both M_0 and M_1 is very close to the expected 480 seconds. Since each master begins to record the processing time only when it detects all four worker processes have begun processing by receiving a connection from all four workers, the total processing time recorded by each master may vary. Furthermore, since each worker contacts M_0 first and then M_1, M_1 detects the start of all four workers later than M_0, effectively delaying the time of start of job and may reduce the total processing time recorded by M_1 in the meantime. As mentioned earlier, we cannot predict the duration of these runs, which is evident from the error shown in the table.

For the case in which two masters are initially partitioned from each other, but reconnect with each other after a master received five tasks, we expect the processing time to be $\lceil \frac{15}{4} \rceil * 120 + 30 = 510$. This is because the four workers are still processing tasks in parallel, with M_0 distributing tasks from the beginning of the task list and M_1 distributing tasks from the end of the task list as illustrated in Figure 4.9. However, there is a delay of 30 seconds because the workers created by M_1, which are in the same local area network as M_1 and are also initially partitioned away from M_0, attempt to contact M_0 first and waits for 30 seconds before they try to contact the next master, M_1, in their master IP lists. We see from Table 5.1 that the actual processing time we recorded is very close to the expected time. Since we expect M_0 to receive five results first, as it begins distributing tasks to workers first, by the time M_0 receives the 5^{th} result, M_1 would have already distributed six tasks because it is behind M_0 by only 30 seconds while each task takes 120 seconds. Thus, we expect M_1 to receive six results directly from workers, meaning M_0 would receive six results from M_1 when they reconnect with each other. M_0 should receive all other results directly from workers. Since our goal for this protocol is to simply ensure that the primary master, M_0, receives all results if it does not fail, all masters terminate when M_0 has received all results, regardless of the status of the backup masters. From Table 5.1, we see that the data we collected matches our expectations.

In the case with three masters in which M_0 and M_1 partitioned away from each other for the duration of the job, we expect the total processing time to be $\lceil \frac{15}{6} \rceil * 120 + 30 = 390$. With the forwarding we built in (Figure 4.12), although M_0 is partitioned away from M_1 for the entire duration of the job, M_0 still receives task results from M_1's workers via M_2. Thus, there is effectively six workers working in parallel, with the two workers in M_1's Condor pool lagging behind approximately 30 seconds, and hence the extra 30 seconds in the total processing time. Since M_0 has four workers to work with (M_0 and M_2 each created two workers), compared to the two for M_1, M_0 receives results twice as fast as M_1. As described in Figure 4.9, M_0's `todo` task list consists of all tasks in the order of task zero, task one, task two, and so on, while M_1's `todo` task list consists of all tasks in the reverse order. By the time M_0 received the first eight results and sent out the next set of four tasks, M_1 would only have received four results from the last four tasks and sent out the next set of two tasks. Thus, in the end, M_0 received eleven results directly from workers and four results forwarded by M_1. Since M_2 has connection to both masters, M_2 should receive all results directly from other masters. Table 5.1 confirms our speculations. We notice that the processing time for M_1 is six seconds longer than M_0. We attribute this to the fact that M_1 terminates only when it receives a termination message, which was originally sent out by M_0, but forwarded to M_1 by M_2. This message is received only after M_2 accepts all results sent by M_0 as the termination message is the last message sent out by M_0.

Overall, with the exception of the unpredictability of Condor as demonstrated in Section 5.1, our tool generally performed as we expected.

5.3 Implementation Change

In the implementation described in Section 3.5.1, the script program was originally coded to use the `-r` option in the `condor_submit` command to submit multiple master jobs to different machines. In the process of doing the evaluation described in the previous section, we discovered that the `-r` option was only intended for debugging purposes by the Condor developers and is not functional in the newest version of Condor. The submission of master jobs was subsequently modified to using `ssh` to directly log on to a remote machine and start the master process there. This implies that the user would need to have direct access to machines in different Condor pools, and therefore require user accounts on different networks. This differs from our original intention, which was to utilize Condor to manage the computing resources and to free the user from keeping track of several accounts. This led to the future work we propose for the Condor project in order to provide fault tolerance more gracefully. We realize that there are concerns in using the `-r` option to submit scheduler universe master jobs to machines that may only be committed to processing a Condor job at certain times. However, requiring the user to obtain user accounts in multiple networks was not the original intention of Condor or of M/W'. In light of this, we propose that a new class of machines be created in Condor to handle scheduler universe jobs. These machines will offer the commitment that is needed to provide the added fault tolerance.

6

Conclusion

As computational Grids span across wide area networks and computations grow larger to require hundreds of workers for longer lengths of time, there is a higher probability of machines from different local area networks partitioning away from each other and of machines crashing. Computational Grid developers have traditionally addressed the problem of making jobs resilient to machine crashes by restarting a job on a different machine. However, because of the sizes of the jobs running on Grid systems, simply disposing output from the crashed machines has become inefficient. The performance of this approach in terms of the completion time of a job can be improved by logging process state information to restart a process from the last checkpointed state, as is done in Condor. With Condor, however, checkpointing and re-submission of a process is provided only for jobs in certain universes, and these services are provided by the central manager of a Condor pool, which is a single point of failure. Jobs in certain universes, such as the scheduler universe, are not monitored by the central manager, and hence, would require a user to restart a job manually. The second problem in wide area M/W computations, the problem regarding two cooperating machines partitioning away from each other, has not been considered previously because Grid systems have not been deployed across a wide area network until recently. We believed there was a need to address these fault tolerance issues to ensure that processes continue to make progress with the limited resources available to them in any situation. In this thesis, we have built a M/W template that can tolerate network partitions as well as failures of machines running master processes in a wide area network on a popular platform, Condor.

Our M/W template is extended from that provided by the Condor developers. Since Condor is a popular platform and M/W applications are common, we designed our template to be compatible with the original Condor tool. To that end, we retained the same interfaces to user-specific functions as in the original tool. We preserved the structure of the original master and worker templates and added lower level communication mechanisms to support replicated masters. Our end product is a M/W template that supports replicated masters to tolerate both network partitions and failure of machines.

A M/W application based on our template can be started on several Condor pools, each of which is most likely a Condor pool deployed in a local area network that rarely suffers from network partitions but can partition away from other Condor pools often. A replicated master is placed on each of these Condor pools. Each master then submits worker jobs to the central manager in its pool, creating worker processes in the same local area network as itself. By starting each worker job in the vanilla universe, our tool inherently tolerates failure of machines running worker processes because of the monitoring of the Condor central manager, as discussed in Section 3.1.1. Furthermore, by placing replicated masters in different Condor pools, and thus, creating workers in different Condor pools, we have eliminated the single points of failure at the each of the central managers.

The replicated masters cooperate with one another by sharing results. Our tool tolerates master failures because of the replication master processes. With masters and their workers placed in different Condor pools, which should be contained in different local area networks, each unit of master and its workers continues to make progress in the face of lack of communication with units in other partitions of the network.

Because of our goal of providing a practical tool, our template is built such that it only dictates the general behaviors of a master and a worker in failure-free conditions and in undesirable situations such as the failure of a master. We provide guidelines for the actions to be taken by a process in a dynamic environment. As illustrated in Chapter 4, we extracted several important components in a fault-tolerant wide area M/W application. We believe a well-constructed wide area distributed M/W application should solve problems in three areas: the knowledge of the machines involved in the application (system view), the ability to determine how a job is to be delegated among the processes (scheduling), and the ability to share and merge information (merging). We believe these are general policies that a distributed application should consider and we have incorporated these abstractions into our template. We have shown in Chapter 4 that we have successfully built two different protocols, the primary/backup protocol and the parallel masters protocol, based on our template. Although topics such as knowledge of process group, scheduling of tasks, communication protocols for coordinating processes, etc, are important aspects of a fault-tolerant tool in a wide area network, we understand that these are problems with various solutions for different conditions and user needs. Thus, we have elected to provide only interfaces to these features in our tool to allow a user to customize a M/W application that best suits his/her needs.

Condor is a unique computational Grid middleware that scavenges idle computing power. Condor's central manager ensures that a machine servicing a Condor job is returned to its owner upon the return of the owner or based on other requests, such as time limitations, specified by the owner. A job on such a machine is then migrated or restarted by the central manager on another available machine. Since both the master and worker processes in our tool require communication with each other, one of the challenges we faced was providing a mechanism for processes to communicate with each other despite the migrations of processes that cannot be predicted. Our solution is to submit multiple master processes in the scheduler universe each on a different machine. This eliminates the migration of the masters in addition to tolerating failure of any particular machine running a master process. Worker processes, on the other hand, are given the relevant IP addresses of the machines running master processes, and they are submitted to the vanilla universe. These worker jobs are processed subject to the availability of suitable machines and are restarted when needed. They maintain communication with masters using the IP addresses given.

From our experience described in Section 5.3, we conclude that fault tolerance implemented in software requires the cooperation of a few machines dedicated to fault tolerance. Our design of M/W template that tolerates master failures and network partitions started with the decision to replicate masters. However, with the unique environment Condor provides, we realized that these master processes would have to be placed on machines that were committed to completing these jobs as opposed to machines that would return to their owners whenever needed. Hence, we believe that a new class of machines, one such that each machine is dedicated to completing a job, is required in Condor if masking of failures and undesirable conditions is to be provided. The current fault tolerance Condor provides involves having the central manager monitor jobs and logging process states to allow for manual re-submission of a job that the central manager does not monitor and migrate. Requiring the central manager to monitor jobs indicates a single point of failure at the central manager, and manual re-submission of a job does not mask failures from users at all. Our version of M/W template, on the other hand, masks failures and undesirable network conditions by placing master jobs on specific machines that complete these jobs regardless of the presence of their owners. Hence, to effectively tolerate failures, we believe that a certain degree of cooperation from hardware is required, and in the case of Condor, a new class of machines should be defined.

During the testing of our template, we noticed that it took an unexpectedly large amount of time for the Condor pool at University of Wisconsin, Madison to allocate suitable machines to handle vanilla universe jobs even though there seemed to be more than enough suitable machines that were idle at those specific moments. A few samples of this unexpected finding are shown in Chapter 5. As stated before, we have sent the data we collected to the Condor developers, but we have yet to hear back from them. A possible explanation is the use of a work conserving algorithm in the process of assigning jobs to idle machines.

Finally, in the process of evaluating our tool, we discovered that the experimental runs with simulate network partitions we performed could not scale to the sizes of real-world parallel applications due to resource and time constraints. We find that it would be useful to provide some standard benchmarks, a practical simulation package, and consensus in certain desired characteristics of a parallel distributed application in the future to help the development of a parallel application, as opposed to the current trial-by-error approach.

Bibliography

[1] AppLeS Master/Worker Application Template at `http://gcl.ucsd.edu/amwat`.

[2] D.G. Andersen, H. Balakrishnan, M.F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Symposium on Operating Systems Principles*, pages 131–145, 2001.

[3] APST project webpage at `http://grail.sdsc.edu/projects/apst`.

[4] Ö. Babaoğlu, A. Bartoli, and G. Dini. Programming Partition-Aware Network Applications. In *Advances in Distributed Systems*, pages 182–212, 1999.

[5] Ö. Babaoğlu, R. Davoli, and A. Montresor. Partitionable Group Membership: Specification and Algorithms. Technical report, University of Bologna, Bologna, Italy, 1997.

[6] Ö. Babaoğlu, R. Davoli, A. Montresor, and R. Segala. System Support for Partition-Aware Network Application. Technical report, University of Bologna, Bologna, Italy, 1997.

[7] Karan Bhatia. *Dynamic Replication in Wide Area Environments Using Message Logging*. PhD thesis, University of California, San Diego, 2001.

[8] K.P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53 and 103, December 1993.

[9] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of Supercomputing*, 2000.

[10] Cleo home page at `http://www.lns.cornell.edu`.

[11] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth Symposium on Principles of Distributed Computing*, pages 1–12, 1987.

[12] D.H.J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. Technical Report DUT-TWI-95-130, Delft University of Technology, The Netherlands, 1995.

[13] X. Evers. Condor Flocking: Load Sharing between Pools of Workstations. Master's thesis, Delft University of Technology, The Netherlands, 1993.

[14] Globus project webpage at `http://www.globus.org`.

[15] J. Goux, J. Linderoth, and M. Yoder. Metacomputing and the Master-Worker Paradigm. Technical report, Argonne National Laboratory, February 1999.

[16] GriPhyN home page at `http://www.griphyn.org`.

[17] J.J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, U.S., 1991. ACM Press.

[18] Legion project webpage at `http://legion.virginia.edu`.

[19] J. Linderoth, S. Kulkarni J. Goux, and M. Yoder. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing*, pages 43–50, August 2000.

[20] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor – a Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[21] G.G. Malewicz, A. Russell, and A.A. Shvartsman. Distributed Cooperation During the Absence of Communication. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 119–133, 2000.

[22] K. Marzullo, M. Ogg, A. Riccardi, F.A. Calkins, and E. Rothfus. NILE: Wide-Area Computing for High Energy Physics. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 49–54, 1996.

[23] MCell Home Page at `http://www.mcell.cnl.salk.edu`.

[24] NILE home page at `http://www.nile.cornell.edu`.

[25] J.K. Ousterhout, H. Da Costa, D. Harrison, J.A. Kunze, M. Kupfer, and J.G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Symposium on Operating Systems Principles*, pages 15–24, 1985.

[26] M. Satyanarayanan. A Study of File Sizes and Functional Lifetimes. In *Symposium on Operating Systems Principles*, pages 96–108, 1981.

[27] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[28] The SETI@home project webpage at `http://setiathome.ssl.berkeley.edu`.

[29] J.B. Sussman and K. Marzullo. The Bancomat Problem: An Example of Resource Allocation in a Partitionable Asynchronous System. In *International Symposium on Distributed Computing*, pages 363–377, 1998.

[30] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, and C.H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated System. In *15th Symposium on Operating Systems Principles*, pages 172–183, 1995.