DESIGN AND IMPLEMENTATION OF A FLEXIBLE CLUSTER-SCHEDULING FRAMEWORK

BY

ESTEBAN TRISTAN PAULI

B.S., University of California, Davis, 2004

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

In the past, centralized supercomputers were the main source of computing power for those needing hundreds to thousands of processor hours. The schedulers for these systems were usually first-in-first-out (FIFO) queues, with reservations for special allocations having to be done by contacting the administrators. As powerful workstations became more common and people realized how many cycles were going unused, systems such as Condor [?] came about to take advantage of this by harvesting idle cycles. Now, however, small clusters of twenty to a hundred or more dedicated compute nodes are becoming more common. These clusters are owned by diverse organizations with varied scheduling needs. Trying to use the FIFO schedulers of supercomputers or the cycle-harvesting schedulers used for idle workstations often leads to scheduling policies that are less than optimal for the owners of the cluster. For this reason, we have designed and implemented a flexible cluster-scheduling framework. This framework allows for easy implementation of different scheduling strategies. It provides a robust system for changing the information stored about jobs, changing how jobs are scheduled, and changing how jobs are monitored. Furthermore, it allows for the implementation of scheduling strategies which understand the run-time systems of the applications running on the cluster to allow for advanced features such as checkpointing and shrinking and expanding of jobs to make the best scheduling decisions possible.

To Jill Marie Heidinger

Acknowledgements

I would like to thank my advisor, Professor Kalé, for his guidance in this research effort. I would also like to thank Eric Bohm and Greg Koenig for their assistance in the design of the FCS framework. They played a large part in this process, and their experience was invaluable. I would also like to thank the other members of the Parallel Programming Lab, who helped with all the complexities of the system programming which was required to implement FCS. Finally, I would like to thank all the people who designed and implemented the original Faucets system. Without all their hard work on the original system, we may never have started on the new one.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Since early in the history of computers, scientists and engineers have wanted to dedicate thousands of processor hours to solving difficult problems. Initially, they turned to supercomputers as their main source of computing power. These machines had simple first-in-first-out (FIFO) schedulers. Since most users found this policy to be sufficiently fair and efficient on supercomputers, this practice stuck and is still common practice today. Despite the popularity of the system, it does have some drawbacks. If, for example, a cluster has ten nodes available and the next job in the queue requires twenty, no small jobs can be scheduled while waiting for more nodes to become available. Although the scheduling policies have become a bit more advanced (the problem described above has been solved by backfilling [?], for example) and utilization has improved, there are many cases where maximizing a machine's utilization does not maximize the owner's utility.

As workstations started to become more powerful and more common, people started to realize that much of the time these machines were idle and the lost cycles could be harvested. Systems such as Condor [?] came about to make this possible. However, this system had some shortcomings. The main one was that jobs could only use one processor at a time: there was no support for parallel jobs. Furthermore, as soon as a user directly started using a machine on which a Condor job was running, the job would be checkpointed and not restarted until later. While this is a good policy for cycle-harvesting, it makes it very difficult for the owner of the job to know when their results will be ready.

In addition to harvesting idle cycles and using supercomputers, users in the last decade have found a new option. With the rapidly increasing power of commodity processor and interconnects, coupled with the rapidly decreasing cost of these components, it has become common for small organizations to have clusters of tens or even hundreds of dedicated compute nodes. Unfortunately, the above scheduling models are not always a good fit for the organizations which own these clusters. For example, certain organizations might want to give users priorities based on rank, need to access the cluster, or any other variety of reasons. Furthermore, if the cluster is collectively owned and financially maintained by distinct entities (such as different departments inside a university), a policy in which users from no department can monopolize the use of the resource might be desired. Similarly, an organization might want to make sure no single user can monopolize the cluster for too long.

No single scheduler or policy can meet such diverse scheduling needs. Writing a scheduler from scratch can be prohibitively expensive, time consuming, and difficult for most of the owners of small clusters. For this reason, we have designed and implemented FCS, a Flexible Cluster-Scheduling framework. FCS allows a wide variety of scheduling policies to be easily implemented. The implementer of a policy can easily change what information is stored about jobs, how jobs are scheduled, how jobs are monitored, and how users interact with the system. This allows organizations to more easily implement customized scheduling policies with minimal attention having to be paid to stability, security, and maintenance.

The rest of this chapter describes Faucets, the predecessor to FCS. Chapter 2 describes the design and implementation of the system. Chapter 3 describes several scheduling strategies implemented in FCS and how users can implement their own. Chapter 4 describes the features of other cluster schedulers and how those features could be implemented in FCS. Finally, chapter 5 draws conclusions and suggests directions for future work.

1.2 Faucets

Faucets is the predecessor to FCS. Its cluster scheduler was originally described in Sameer Kumar's Master's Thesis ([?]) as a scheduler capable of taking advantage of applications written using the Charm++ language and run-time system ([?]). Jobs specified a minimum and maximum number of processors they could use, and the scheduler would dynamically change the number of processors used by jobs in order to make better scheduling decisions.

A later paper by Kalé et al ([?]) described how additional components could be added to the system in order to create a meta-scheduling framework capable of allowing owners of different clusters to join a federation and share their resources with each other. In order to accommodate this, the scheduling strategy in the cluster scheduler had to be tweaked to be able to ensure users' Quality of Service (QoS) contracts were met.

As Faucets became a more mature software package, it was deemed ready to be used by a real group of users. It was installed on a cluster of over one hundred nodes owned by a group of professors in the Department of Computer Science at the University of Illinois. This group of users ran mostly single processor jobs which sometimes came in bursts of hundreds of jobs from single users. These jobs would oftentimes request several days of run time. This allowed single users to monopolize the cluster, even at times of high demand. For this reason, a new scheduling strategy was implemented which limited the number of long-running jobs any single user could have on the cluster at a given time.

All of the above situations made the Faucets software large and unmanageable. It was unclear which variables were valid for jobs depending on which strategy was being used. This caused many bugs and made debugging efforts difficult. Combined, the changes made to the scheduler made clear the need for a flexible scheduling framework. It was determined that this framework should keep scheduling strategies distinct, make changes to the information stored about jobs separate for each scheduling strategy, and should allow scheduling strategies to change as little or as much as needed. These main goals drove the redesign and reimplementation of the system. The resulting FCS framework is described in detail in the following chapters.

Chapter 2

Design and Implementation

2.1 Architecture

The original Faucets system consisted of a database, a client, and a multithreaded program which acted as a host, scheduler, job monitor, and cluster monitor. This made programming error-prone and debugging difficult. The threads in this program protected access to shared variables with a single lock. This negatively impacted performance as much time was spent trying to acquire this lock. For example, if a user submitted many jobs automatically with a script, the scheduler would start trying to schedule the first jobs while the rest were being submitted. Since there was only one lock for the entire system, while the scheduler was launching jobs from the front of the queue, the host could not add jobs to the end of the queue. It was not uncommon for users to have to wait more than a minute for all their jobs to be enqueued. While some might not find this to be an inconvenience, many users found it to be less than desirable.

Another problem was that all strategies used the same Job class. Whenever a new field was needed, it was added to this class. As people tried to implement more scheduling strategies, it became more difficult to tell which fields were and were not valid when the scheduling strategy was switched. This made it difficult to implement new strategies and maintain existing ones, especially for people not familiar with the code. A large percentage of bugs was attributed to this. For these reasons, the entire architecture of the system was

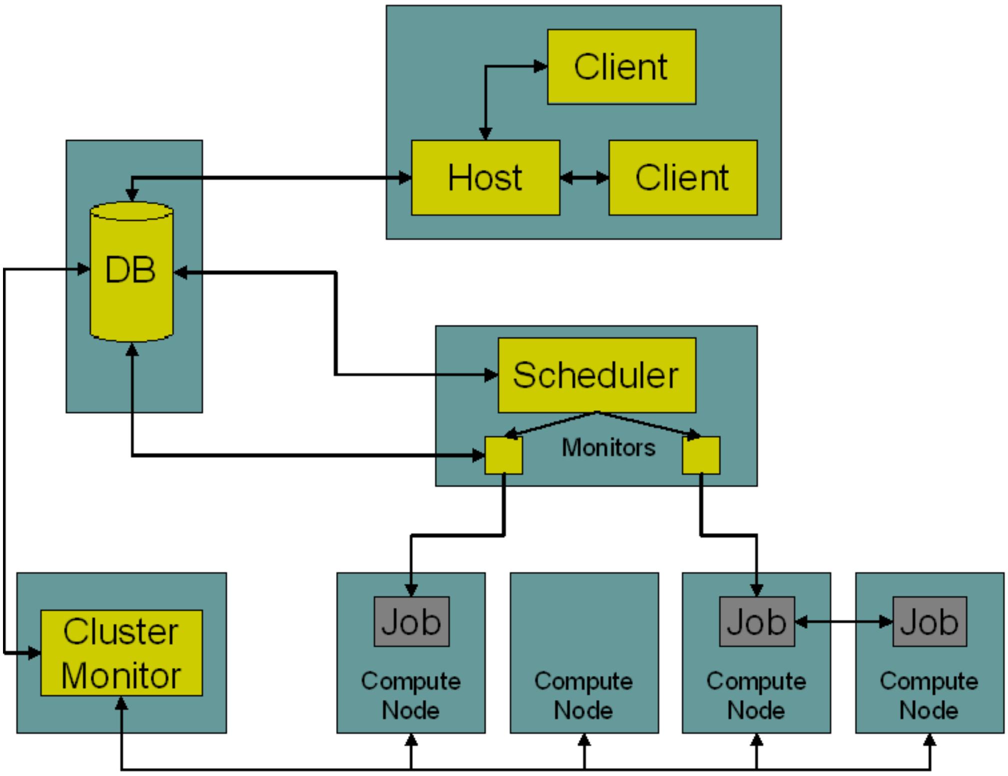

Figure 2.1: FCS Architecture

redesigned with the experience from Faucets in mind.

The FCS framework consists of several different programs. These are depicted in Figure 2.1. The outer boxes represent different physical machines. The inner boxes represent different processes. Arrows represent communication channels between processes and machines. The system can be configured so that the database, cluster monitor, scheduler, host, and clients are all on the same physical machine: the figure simply shows which components must be on the same machine.

2.1.1 Database

The central component of FCS is the database, which stores all information about the system. The main tables in the database are the job tables. These store all the information about every job in the system (such as if the job is queued, when the job was submitted, and the resources requested for the job). Even after jobs are finished, their information is stored so that system administrators can track what happened to jobs in the case of errors. In addition to the job tables, the database stores tables about the state of the cluster. This includes the current state of compute nodes and a history of their status. In another set of tables, the database stores information about the maximum resources users can request for their jobs. For example, this can include the maximum number of nodes and time which can be requested. Finally, the database has a set of log tables. Rather than using a simple file for logging system events, database tables are used to allow easier searching of events.

One advantage of storing all the information described above in the database is that it gives users and system administrators great flexibility in changing the behavior of the system and the status of the job queue without shutting down the system. For example, if a user realizes they did not request enough time for their job and it is still queued, they can issue a command to increase it. Another use might be for the system administrator to dynamically change the maximum amount of resources users can request for their jobs without interrupting the availability of the system or affecting jobs already in the queue.

A perhaps even greater advantage of storing everything in a database is that if any of the other components crash, the critical state of the system is not lost. If the scheduler crashes, for example, all submitted jobs are still stored in the database and users can keep submitting and monitoring jobs. If the host crashes, the scheduler can keep launching jobs. After any component has crashed, it can read the database and figure out what it was doing and resume where it left off.

In addition to storing all information about the system, the database also serves as a communication point. The other components in the system cannot directly communicate with each other. Rather, they write and read the database. For example, the host writes submitted jobs to the database. The scheduler then reads the list of jobs, makes a scheduling decision, and updates the database to reflect what it did. If the user wants to see the state of their jobs, they issue the appropriate command to the host, which then fetches the information from the database (reading any changes made by the scheduler). This helps isolate the components from each other, making it less likely that a bug in one component will adversely affect another. Furthermore, it makes it difficult for a malicious user to crash the scheduler or cluster monitor since they can only directly communicate with the host.

The database used in the current implementation is a MySQL [?] database. This was chosen for several reasons. First, we wanted a well-tested, robust permanent storage system which could be quickly read and written. We also wanted to allow multiple concurrent readers and writers. For these reasons, we decided to use an existing system rather than design our own. A relational database was chosen over persistent object databases such as SHORE [?] or ORION [?] because although we do write mostly objects into the database, we want users of FCS to be able to easily write their own scheduling strategies. More people are familiar with SQL and relational databases than with object databases. Also, we did not want to limit how these strategies can interact with the database and felt the relational database model would allow for more unforeseen possibilities. Of the available relational databases, we chose MySQL because of its good performance and stability [?]. Furthermore, it is free, so users of FCS are not required to purchase any software to use FCS.

Since the database does not need any special system privileges, it can run as a non-privileged process on the system. In general, the same methods used for securing any database should be used to secure the FCS database. It should be ensured (as best as possible, anyway) that only other FCS components can read and write the database tables. If regular users are allowed to accesses them directly, they could put the database in an inconsistent state or put it in a state which would allow them to run their jobs as other users, gaining privileges they would not normally have. The stability, performance, and security of the database are critical. If the database fails, the system cannot function. If it is compromised, users can gain elevated privileges on the system. If it does not perform efficiently, not only will response time to the users be degraded, but more transactions might collide and fail, keeping the scheduler from making progress. For these reasons, proper configuration and maintenance of the database is crucial.

2.1.2 Host and Client

The host and client are the only way for users to interact with the system. The host acts as a server daemon, waiting for user requests. When a user wants to perform an action, they start a client process. This client connects to the host, sends the request, and waits for a reply. When the reply comes, the client reports it to the user and terminates. Depending on the parameters given to the client, it can be used to submit jobs, monitor queued and running jobs, and find information about finished jobs. Furthermore, the administrator can also use it to perform administrative duties on the database.

From the point of view of regular users of the system, the most important function of the host and client is to submit jobs. The standard way to submit a job to the system is to submit a job script. This script is a standard shell script with additional tags to describe the resource requirements of the job. This script is parsed by the client for any of the special tags. These are passed along with any command line parameters to the host. The host then checks this information against the maximum limits allowed for jobs (as currently stored in the database). If the job does not exceed the limits, it is added to the queue and a unique job ID is returned to the user. Otherwise, the reason for the job being denied is given to the user and nothing is stored in the database. When jobs are accepted, the script is copied for the scheduler to later read and execute. This is common practice in other schedulers, such as PBS [**?**].

In addition to submitting jobs, users also use the host and client to monitor the progress of their jobs. Users can ask for a list of all running and queued jobs, a completed job, or just specific jobs. The host fetches the current state of these jobs from the database and returns it to the client. The client then formats and outputs the information to the user. As described in Section 2.2, the information communicated about the jobs depends on the scheduling strategy. For example, in a priority-based scheme, the priority of each job can be communicated and displayed. This allows for some of the flexibility which was a key goal of the FCS system.

One final use of the host and client from the users' perspective is to manage their queued and running jobs. Regardless of the strategy, the client can be used to remove queued jobs from the system or to kill running ones. Depending on the scheduling strategy used, the user might also be able to change the resources requested for their queued or even running jobs (see Chapter 3). Additionally, the client and host could be used to allow the users to send signals to their running jobs. This is something that is not typically allowed by other cluster schedulers but could be used to do things such as forcing a job to checkpoint itself.

From the administrator's point of view, the host and client can be used to help maintain the state of the system. For example, the client can be used to issue shutdown commands to any of the components of the system. It can also be used to change the maximum resource limits allowed for jobs. Furthermore, administrative tasks such as removing old jobs from the database or removing old log entries can be done through the client. Finally, the administrator can request that the log tables be dumped into a file if desired.

To minimize the risk of security issues, the host process can be run without any special system privileges. The only privilege it needs is access to the database, which is controlled by the database software rather than the operating system. Clients are run with the privileges of the user starting them. To ensure users do not issue commands they are not allowed to issue, the client authenticates itself to the host. In the current implementation, the client and host communicate through UNIX domain sockets. The host asks the operating system for the identity of the user running the processes on the other end of the socket, making sure the client cannot fool the host unless it can fool the operating system. In a future implementation, the authentication could be changed to some other mechanism so that the implementation does not need to rely on UNIX domain sockets. So long as the host can verify who is running the client, any authentication mechanism can be used.

2.1.3 Scheduler

The scheduler is the component which does the most work and is at the heart of the system. Its main responsibility is to read the database, make a scheduling decision, update the database, and perform whatever actions are required by the scheduling decision made (launching or killing jobs, for example). The scheduler is also responsible for monitoring the jobs it launches. Because of its responsibilities, the scheduler is the component which is most heavily affected by the scheduling strategy used. This section described the generic steps; how different scheduling strategies come into play is described in more detail in Section 2.2.

The main loop of the scheduler consists of four main steps:

1. The first step is to read the database. Here, the scheduler reads the information about the current state of the cluster and the list of running and queued jobs. Since nodes might fail or be repaired while the scheduler is running, it is vital that the state of the cluster is read along with the list of jobs so that it knows which resources are currently available when making scheduling decisions.

2. The second step is to make a scheduling decision. Here, the scheduling strategy decides whether jobs should be launched, killed, checkpointed, or whatever else the strategy can do with jobs.

3. In the third step the new state of the system is written back into the database.

4. During the fourth step the scheduler to takes whatever actions it decided upon in the second step.

The first three steps all happen within a database transaction. If this transaction fails (if, for example, the cluster monitor simultaneously discovers a node failed and notes this in the database), the fourth step is skipped. This is to ensure jobs are not launched on faulty nodes, jobs are not killed unnecessarily, and other similar problems do not occur.

To launch a job during the fourth step, the scheduler first forks a child process which detaches itself from the process tree of the scheduler. This child will be responsible for launching and monitoring the job. The reason we detach the job monitor from the scheduler is that we want the job monitor to be able to keep running in case the scheduler is shut down or if it crashes (of course, the monitors can be shut down along with the scheduler if desired, with the understanding that this will kill the running jobs). Once detached from the process tree of the scheduler, the job monitor switches its user ID to match that of the user that submitted the job. It logs into the head node for the job (as decided by the scheduler), executes the job, and waits for it to finish. After the job finishes, the monitor switches its user ID back to its original ID and records in the database that the job finished. The job monitor then terminates.

To kill a job, the scheduler sends the `SIGINT` signal the job monitor. When the monitor terminates, the connection to the job's node is broken, killing the process tree of the job. Under normal conditions, this is sufficient to clear the node of any processes associated with the job. However, if any process from the job detaches itself from the main process tree, it will not die. For this reason, the scheduler must log into to every node assigned to the job and kill any process owned by the owner of the job. Unfortunately, further complications can arise. Many administrators allow the scheduler to assign more than one job to a node if the node has multiple processors. If two jobs belonging to the same user are assigned to the same node, we cannot log into the node and kill every process owned by the user since we only want to kill the ones associated with a particular job. In this case, we simply let the processes keep running until the other job finishes, at which point we kill all processes owned by the user on the node. Although not optimal, this provides a solution. Ideally, we would run all users' jobs inside virtual machines and simply kill the virtual machine to terminate the job (see Section 3.4).

Due to the wide variety of tasks and system calls the scheduler must perform, it needs to run with more privileges that the other components. Specifically, the job monitors must be able to log in to the compute nodes as any user. In the current implementation running under a UNIX system, this means being able to do a `setuid(2)` call to set its user ID to any other user on the system, typically requiring the scheduler to be run as the root user. Additionally, once the process has switched its user ID, it must be able to connect to the compute nodes without prompting the user for a password. This can be accomplished by using `rsh`, or the preferred method of using `ssh` and requiring all users to properly configure their authorized keys file. Other than the above, no special privileges are required. However, future scheduling strategies might require more privileges, so the implementation of the framework should not limit this possibility.

2.1.4 Cluster Monitor

The cluster monitor is a simple, yet crucial component. As its name suggests, it is responsible for monitoring the state of the cluster. It periodically tries to connect to every node and see if it is running correctly. By default, this means being able to `ssh` into the node and execute the `date` command. However, this can be overridden. For example, trying to execute the `ls` command in a random user's home directory would let the scheduler know if NFS directories are being mounted correctly. If the monitor finds any node to not be behaving as required, it marks this in the database. The next time the scheduler performs its main loop (described above in Section 2.1.3), the scheduler will see that a node has failed and will not schedule jobs on it.

If the writer of a strategy wishes (see Section 2.2), the cluster monitor could theoretically be made to perform more complicated tasks. For example, if the cluster is commonly used to run large parallel jobs and has a high-performance interconnect connecting the nodes in addition to a standard Ethernet (as is commonly the case), the cluster monitor could test the links in this network. If a certain link is found to be faulty, the scheduler could use this information to not launch a parallel job that requires multiple nodes on the nodes at the end of the link. However, it would know that it could launch single-processor jobs on these nodes as these jobs would not need the high-speed interconnect.

Like the host, the cluster monitor needs no special privileges as far as the operating system is concerned. It only needs to log into the nodes of the cluster as a normal user. The cluster monitor also needs permission to access the database, but that is controlled by the database software and requires no extra privileges at the operating system level.

2.1.5 Adding Components

Although we have taken care to carefully design the FCS framework so that the behavior of the above components can be easily changed (as we will see in Chapter 3), it is certainly possible that some users will want behavior which cannot be easily integrated into any of the above components. Since all components communicate through the database, adding a new component can be quite straightforward. The new component can be a standalone application which communicates with the other components through the database. It can use the same `Job` and `Strategy` classes (described below) to make the implementation simpler. To communicate with other components, it can access the same database. Taking this into account, writing a new component should not be difficult.

2.2 Strategies and Jobs

In order to make FCS a flexible cluster scheduling framework, all of the components described in Section 2.1 had to be implemented in a flexible way. To achieve this, we used an object-oriented representation of jobs and scheduling strategies. All jobs in the system (except when stored in the database), are represented by instances of the `Job` class (or one of its subclasses). Similarly, each component of the system has an instance of a `Strategy` (sub)class. By writing subclasses of these two classes, we can easily change the behavior of the different components to create a wide variety of scheduling strategies. Below we describe how these two classes interact to achieve this.

2.2.1 Database Interaction

Since all the important state of the system is kept in the database, one of the most vital ways in which the `Job` and `Strategy` classes interact is the manner in which they access the database. The three operations which a strategy can perform are adding jobs to the database, updating a job already in the database, and reading a job from the database. Each of these operation is described in turn below.

When a scheduling strategy has created a new job (for example, when the host has received a request to add a job to the queue), it must add it to the database if it wants any other component to be able to see it. The first thing the strategy must do is read in the `Limits` tables from the database. These tables describe the maximum resources any job may request (for example, number of processors and time). A strategy may have additional limit tables from which it can read additional data (such as maximum jobs allowed per user at any one time). With the `Limits` object, the strategy checks the job against the resource limits. If the job passes the check, the strategy passes a buffer to the job into which the job object writes SQL code to insert itself into the database. If the strategy is using a job class different from the default, the implementer must make sure the buffer is passed to all necessary levels of the class hierarchy so that the entire state of the job is recorded. The strategy then passes the buffer to a database interaction routine which executes the query, adding the job to the database.

Updating a job is similar to inserting it. When a scheduling strategy has modified a job (by launching it, killing it, or observing its successful completion, for example), it must write the new information about the job back into the database. As when the job was inserted, a buffer is passed to the job object, into which the job object writes SQL code for updating its information in the database. Like before, this buffer must be passed to all necessary levels of the job object's class hierarchy if it is not an instance of the default `Job` class. The strategy object then takes the query buffer and passes it to the database API so that the job's new state will be recorded in the database.

Reading jobs from the database is essentially the reverse of updating them. When it needs a particular list of jobs (like all queued and running jobs when the scheduler tries to launch more jobs), the strategy class creates a query for reading these jobs from the database. Here, the strategy object has to make sure it reads every table associated with all necessary levels of the class hierarchy for the strategy object if it is using any additional data not included by the default `Strategy` class. The strategy object then passes the query to the database as when writing. Now, however, it receives a number of resulting rows from the database (one per job). The strategy calls a virtual function which takes this row and creates a new job object out of it. This job is now ready to be manipulated and stored back in the database as necessary.

2.2.2 Scheduling

The most important interaction between jobs and strategies occur when the FCS system tries to make a scheduling decision. In fact, many strategies can potentially be written by just changing the behavior described here. This is where the scheduling strategy must decide what to do with every running and queued job.

As described in Section 2.1.3, in its main loop, the scheduler first reads in the list of all running and queued jobs from the database. Using the methods described in the previous section, we use the results of the database query to create job objects of a given class. Since the scheduling strategy knows the class of these objects, it can use this information to make more advanced scheduling decisions. For example, the job objects might have a priority field. Using this field, the strategy can make a scheduling decision which will most benefit the organization owning the cluster by running the more important job first. Similarly, if the strategy knows the jobs use a particular runtime system, it might be able to use this to shrink and expand jobs to make better scheduling decisions (see Section 3.3).

Without getting into more detail about specific scheduling strategies, it is difficult to describe more precisely how the job and strategy objects interact to make scheduling decisions. The entirety of Chapter 3 is dedicated to this, so it will not be covered further here.

2.2.3 Job Monitoring

As described in Section 2.1.3 above, whenever a job is launched, a monitor process is spawned and detached from the main scheduler. This monitor process is responsible for launching the job, waiting for it to complete, and recording exit data about the job when it completes. With the base `Job` class, this is all the monitor can do. However, some strategies might find it useful for the monitor to have additional functionality. For example, if we know the job's underlying run-time system is capable of checkpointing the job and resuming it at a later time, a scheduling strategy might be able to take advantage of this to checkpoint low priority jobs and allow high priority ones to run. For this to happen, the implementer of the strategy would add some way for the scheduler to signal the monitor that it should checkpoint the job (such as through a system signal or a socket). The monitor functionality would be overridden to allow it to receive this signal and take the appropriate action. With such capabilities, many more complicated scheduling decisions could be made.

2.2.4 Other Functionality

Aside from the previously discussed functionality, the `Job` and `Strategy` classes also interact to achieve a few other miscellaneous features. For example, the `Job` class controls how jobs are printed when a user requests to see the current status of the queue. If a particular job class has extra fields it should print, it can override this functionality to display this data. Another way in which job classes can be extended is in reading command line parameters when jobs are added. All command line arguments not understood by one level in the class hierarchy are passed on to the next. In this way, a new class which needs to understand an argument such as "-priority n" can override the argument-checking function to just understand this argument and then pass the rest to the base job class to parse. This makes specifying job parameters simple for the users and the strategy writers. There are other small things like this throughout the implementation of the code to support all the features described in this chapter in a flexible and extensible way. In this manner, the work required for writing new scheduling strategies can be minimized.

Chapter 3

Scheduling Strategies

In order to demonstrate the flexibility of the FCS system, this chapter is dedicated to describing several different scheduling strategies and how they would be implemented in the FCS framework. We begin with a simple first-in-first-out strategy (FIFO) and work our way up to a priority-driven strategy which shrinks and expands or checkpoints and restarts jobs to run the most important jobs first. Where the implementation of the strategy is particularly simple or instructive, we show the source code for it.

3.1 Simple FIFO Strategy

In order to implement a simple FIFO strategy for the FCS scheduler, one has to do a very minimal amount of work. All the strategy needs to do is run jobs in the order in which they arrive. The only thing a user can request is a specific number of nodes and the maximum time for which it will use these nodes. Since the default `Job` class records the time every job was submitted and the number of nodes it requested, the FIFO strategy can use this class without any additions. For this strategy, the default behavior for interacting with the database and for monitoring jobs will suffice, so this does not have to be overridden. The only thing that has to be written is the function which given a list of queued and running jobs, decides which to launch and which to kill.

For our simple FIFO strategy, deciding which job to launch next is rather simple. We simply look at the next job in the queue, and if there are enough nodes available to launch it, we do so. If not, we look at the list of running jobs. If any of these has gone over the amount of time it requested, we can safely kill it without breaking any promise to the user. Therefore, as long as there are any such jobs and we still do not have any nodes available, we will pick one of these and kill it. If after killing all such jobs we still do not have enough nodes to run the next job, we must wait until other jobs finish or another job goes over its time limit. If we redefine what we mean by the "next job in the queue," we notice that this functionality will be common to many strategies. For this reason, this code is pulled into the main `Strategy` class as an auxiliary function.

Putting the above functionality in the main `Strategy` class makes implementing the FIFO strategy extremely simple. All we have to do is sort the jobs by submit time. The code for this strategy is given in Figure 3.1. The actual `FIFOStrategy` class only has one function, which uses the `C++` Standard Template Library's `sort` function to sort the jobs by submit time. The comparison is done by the `JobSortDec` class, which is only used as a functor class by passing it as a parameter to the `sort` function. From this implementation of a FIFO strategy, we can see how trivial it would be to implement a strategy that schedules jobs by another simple criteria. For example, by changing the `JobSortDec` class's `operator()`, we can easily schedule shortest jobs first, longest jobs first, largest (by number of processors) jobs first, and so on.

We could make the FIFO strategy smarter by only killing jobs which have gone over their time limit if doing so will allow us to launch another job. Also, if a large job (say one that requires the whole cluster) is at the front of the queue, but based on the time requested for the running jobs we know we will have half the cluster idle for at least another hour before the rest becomes available, we might be able to squeeze in a few small jobs (say one requesting only two processors for five minutes) without really affecting the time at which the large job will run. This technique is known as backfilling [?]. We would no longer have a FIFO strategy, but this behavior could be completely controlled in the `schedule_jobs` function that

```
class FIFOStrategy : public Strategy{
  public:
    FIFOStrategy(){}
    virtual ~FIFOStrategy(){}
    /* 'cluster' is the current state of the cluster.
     * 'jobs' is a list of all running and queued jobs.
     */
    virtual void schedule_jobs(Cluster &cluster, std::vector<Job *> &jobs){
      /* figure out order for jobs using STL sort */
      sort(jobs.begin(), jobs.end(), JobSortDec());

      /* Call Strategy's schedule_in_order function */
      schedule_in_order(cluster, jobs);
    }
};

class JobSortDec{
  public:
    inline bool operator()(Job *a, Job *b){
      if(a->submit_time == b->submit_time)
        return a->dbid < b->dbid; // break ties with ID
      return a->submit_time < b->submit_time;
    }
};
```

Figure 3.1: Code for the simple FIFO strategy.

we changed for our FIFO strategy (although we could no longer use the common auxiliary `schedule_in_order` function). The point is to illustrate that such features can be added by only changing the code in a small, isolated location. This allows researchers and production users to easily write different strategies to suit their needs.

3.2 Fair Scheduling: `LimitFIFO`

In the Department of Computer Science at the University of Illinois, there is a cluster known as the "Architecture Cluster." This cluster is owned and used by a group of professors' research groups (most of whom study computer architecture). Most of the jobs submitted to the cluster use input-level parallelism. They request a single processor per job, give each job a different input to the same executable, and run the job for a long amount of time (24 hours). Because of this, many users write scripts to submit their jobs automatically, and many long jobs arrive at once. If scheduled with a simple FIFO strategy, single users would monopolize the cluster for long amounts of time. Since these users oftentimes want to use the cluster at the same time and cannot wait (due to journal and conference deadlines), this creates a problem as there is high contention for the resource at a critical time.

Coming up with a solution to this was one of the firsts tasks of the original Faucets scheduler. The users of the cluster decided users with the least number of nodes currently dedicated to them should get the highest priority. Furthermore, it was decided that not more than one third of the cluster at any one time should be used for jobs that request 24 hours or more of run time. The Faucets scheduler was rewritten to enforce this new requirement. FCS will eventually replace Faucets on the architecture cluster, so below we describe how we would implement this strategy, which we call the `LimitFIFO` strategy.

If we think about what fields are needed for a job, we realize that nothing different from the FIFO strategy is needed, so we can again use the base `Job` class. In order to implement the scheduling policy, all we need to change is the function which determines which job to schedule next. We could hard-code the values "1/3" and "24," but this would not be elegant or highlight any new features of FCS. Instead, we will add a new database table which will be read along with the state of the cluster during every iteration of the scheduling loop. The cluster state class has virtual members which are used for reading and writing the state of the cluster. We simply have to change these to read the additional table which has the values of the maximum percentage of nodes which can be used for long jobs, and what amount of requested time constitutes a long job. With this information, deciding which job to schedule next can now be easily done in the `schedule_jobs` function as it was for the simple FIFO strategy. The function now just has to be a bit more complicated to ensure the fairness policy, but all of this can be done in a few simple lines of C++ code.

Aside from allowing us to highlight another flexible component of the FCS framework, the above implementation has a distinct advantage over hard-coding the values or reading them in when the scheduler is launched. Since these values are read at every iteration of the scheduling loop, they can be changed dynamically by the administrator during run time without having to interfere with the scheduler. If the cluster is not busy and a single user has many long jobs in the queue which could potentially run since nobody else is running the cluster, the administrator can change the values in the database, allowing the user's jobs to run. At a later time, the administrator can change these values back when more users require the system. With minimal changes to the system, we are able to create a scheduling strategy which meets the user's requirements yet is flexible enough to ignore these requirements when it is safe to do so.

3.3 Priority-Based Scheduling

In most organizations, different tasks have different priorities. Yet many cluster schedulers do not have any notion of priority. In organizations where this is critical, this can be a major problem. Imagine a scenario where a distributed sensor network on a battlefield sends a large amount of data back to a centralized computer center. This data needs to be analyzed immediately so that a commander on a battlefield can use it to make a decision. If the cluster that will process this data is dedicated solely to this task and no other work is done on it, it is potentially a huge waste of resources. However, if the data arrives and the job has to sit in a FIFO queue waiting for its turn, the commander will not have data which could be crucial to analyzing the situation on the battlefield. To help solve this problem, we could take the simple approach of simply killing all non-critical jobs on the cluster when the critical job arrives. These other jobs could be restarted from scratch later. However, any progress is lost and we must restart from the beginning. One way in which FCS could address this is by implementing a priority-based strategy which understands the run-time system of applications and can checkpoint and restart or shrink and expand them.

For the priority-based part of the scheduling, we would need to make a few simple changes. First, we would need to know the highest priority each user can request for a job. To do this, we would change the `Limits` class used by the strategy. We would add a database table which is read by the `Limits` class during the scheduling loop which contains the highest priority each user can request. Having these values in the database would allow us to change them dynamically. This would be useful, for example, if people had different maximum priorities based on military rank, but the priorities were increased for commanders while on the battlefield. Next, we would create a new subclass of the `Job` class. This class would simply add a priority field and the mechanism for reading and writing this field from the database (it would be stored in a new table in the database). The new `Job` class would also override the function which reads command-line arguments to understand the new "-priority" argument. The new `Limits` class would make sure that the priority specified by the user does not exceed the maximum. Now, the scheduling function could be easily changed to schedule the jobs with the highest priority first.

However, we would like to go a step further and be able to make room for high priority jobs as soon as they arrive. We would like to be able to checkpoint jobs currently on the cluster with the lowest priority and run the high priority job. To do this, the scheduler would have to understand the run-time system of the applications, and the run-time systems would have to have a mechanism for checkpointing and restarting applications. There are currently several run-time systems which can do this, such as Condor [**?**] and Charm++ [**?**]. Usually, these mechanisms work by sending a signal to the application. To do this in our scheduler, we would change the job monitoring function of the `Job` class. Depending on the type of application (which could be specified by the user as "-jobtype charm," for example), the job monitor would register a signal handler which when invoked would signal the job to checkpoint itself. When the scheduler needs to checkpoint a job, it would signal the monitor, which would then signal the run-time system of the application to checkpoint itself. The scheduler would then be free to reuse the nodes which were being occupied by the job. At a later point, when the job is restarted, the launching function of the `Job` class would call an overridden function which specifies to the application that it should restart from its last checkpoint. Similarly, using the Charm++ runtime system jobs could be shrunk if the high priority job did not need the entire cluster. Afterwards, the jobs would be expanded to use their original set of processors.

As we can see, this priority based scheduler would not be too difficult to implement. It is a bit more involved than the previous examples, but highlights some important features. The most important is the ability to override the job monitoring and job launching functionality. We can leverage this to implement advanced features like checkpointing and restarting. We could also potentially change the job monitors to allow the application to give feedback to the scheduler. This could be useful if, for example, the application knew it would not finish in time and wanted to request some additional time before it is killed. With small changes, we could add this and other advanced features.

We have implemented a simple priority FIFO, and key parts of the implementation which highlight features of the FCS framework are discussed below. Figure 3.2 shows the constructor for the new job class we need, the `PriorityJob` class. This shows how simple it is to add new arguments which the scheduler uses to determine a job's features. The constructor of the job class is passed a list of all arguments which have not yet been consumed by somebody else. In the given example, the constructor for the `PriorityJob` class first lets the base `Job` class consume all the arguments it understands. It then scans the remaining arguments for a "`-priority`" flag. If the flag is present, the code reads the priority from the next argument, reporting an error if no such argument exists.

Another key feature of the job classes is their ability to read and write the database. Figure 3.3 shows how this functionality is overridden. First, the `update_DB` function is called on the `PriorityFIFO` class. This function calls the overridden function in the base `Strategy`

```
PriorityJob::PriorityJob(list<string> &args, uid_t uid):Job(args, uid){
  priority = DEFAULT_PRIORITY;
  for(list<string>::iterator it = args.begin(); it != args.end();){
    if(*it == "-priority"){
      it = args.erase(it); // 'it' now points to next argument
      if(it == args.end()){
        err = "-priority option not given a priority.";
        status = JOB_STATUS_CREATE_FAILED;
        return;
      }
      priority = atoi(it->c_str());;
      it = args.erase(it);
    }
    else
      it++;
  }
}
```

Figure 3.2: Constructor for the `PriorityJob` class which reads arguments given to the scheduler by a user.

```
void PriorityFIFO::update_DB(Job *job, ostream &o, const Cluster &cluster){
  Strategy::update_DB(job, o, cluster);
  ((PriorityJob *) job)->PriorityJob::update_DB(priority_table_name, o);
}

void Strategy::update_DB(Job *job, ostream &o, const Cluster &cluster){
  job->Job::update_DB(job_table_name, job_node_table_name, o, cluster);
}

void PriorityJob::update_DB(const string &table_name, ostream &o){
  o << "UPDATE " << table_name << " SET priority = '" << priority
    << "' WHERE jobID = " << dbid << ";" << endl;
}
```

Figure 3.3: Functions used for updating a Job's fields in the database.

class. This function calls the `update_DB` function of the base `Job` class (not shown), which writes the SQL code into the stream "`o`" for updating the job's information in the database. When this is complete, this same stream is handed off to the `update_DB` function of the `PriorityJob` class. Here, the code for updating the job's priority in the database is written into "`o`." The code in this stream will be used by whoever called the function to update the job's fields in the database. The process for reading a job from the database is analogous, except instead of passing around a stream into which SQL code is written, a row from a database table is passed around from which each class in the hierarchy reads its fields.

3.4 Virtual Machine Scheduling

When describing the checkpoint mechanism for the priority-based scheduling in Section 3.3, we said the scheduler would have to understand the run-time systems of the applications, and that they would need to have a way of checkpointing themselves. This can be a serious problem for users who want to use other run-time systems or programming languages. Furthermore, the scheduler needs to understand each and every one of these systems. One way to get around this would be to use virtual machines, such as the Xen virtual machine [?]. If we run all jobs inside a virtual environment, we can easily checkpoint this environment and then run it from its previous point without the application even knowing. Aside from simplifying the checkpointing and restarting for the priority-based mechanism, we could also leverage the virtual machine's capabilities to implement other advanced scheduling strategies.

One such strategy is one in which we allow anonymous users to access the cluster. Typically, every user must have an account to access the cluster. These accounts are only given to users who can either be trusted or be held accountable for their actions. This is usually not a problem as most users just want to run their jobs and get their results. However, if an organization wishes to share its cluster with another organization (to trade for time on their cluster, for example), the level of influence an administrator has over the users is greatly reduced. In order to increase security, all jobs could be run inside virtual machines. This would allow users to log into a shared guest account and launch their jobs like any other user. To implement this with FCS, we would only have to change the way jobs are launched. Rather than just running the user's application, we would launch a virtual machine and run the user's job in it. The use of a secure guest account could also be used to implement more advanced strategies as described below.

One of the original ideas of the Faucets system was that users should be able to view compute power as a utility. In fact, the name came from the idea that computing power should flow from the computational grid like water flows from a faucet. Service providers would make their resources available on the grid, and users would log in, run their job, and pay a fee. Although virtual machines might provide a sufficient level of security to try this, we would still need a scheduler which would be able to determine which jobs to run when to maximize profits. Undoubtedly, different profit centers would want to try different strategies to maximize their profits. Using all the techniques described so far, different companies could easily implement different strategies using the FCS framework. These strategies could be written to attract the highest number of jobs possible, or try to attract jobs with just specific characteristics to allow the cluster to find its own niche in the market.

3.5 Deadline-Driven Scheduling

In real life, deadlines play a huge role in the day-to-day workings of organizations. Cluster schedulers, however, tend to completely ignore this fact. The effects of this can have serious consequences. Consider, for example, a cluster shared by several research groups in a university. These research groups are from different departments and do not submit research papers to the same conferences and journals, so they rarely have the same deadlines. Because of this, most users are not bothered by waiting a day or two for their results most of the time. However, at times they would like to specify that they need their jobs completed in just a few hours. In order to accommodate this, the cluster scheduler must have some concept of deadlines for jobs which it can use to make scheduling decisions.

Like most of the other strategies we have discussed, to implement a deadline-driven strategy in FCS, we would have to add a new database table which records the deadline for each job. This has already been discussed, so we will not go into further details here. We have also discussed how to change the scheduling algorithm, so we will not describe that either. What we will discuss is the acceptance and rejection of jobs. We saw in Section 3.3 that the FCS framework allows the host component to read tables from the database which contain the limits of resources users can request. The priority strategy had overridden this functionality in order to read the maximum priorities allowed for each user. An administrator could change these values to allow different priorities to different users at different times. Each time a job was submitted, the host could check this value to either accept a new job or reject it based on the requested priority. Although useful for properties of the job (like priorities) which do not depend on the current status of the job queue, this mechanism is not sufficient for deadline-driven scheduling strategies.

In order to determine whether or not a new job can be run by the requested deadline, a scheduling strategy needs to consider the current state of the cluster (all running and queued jobs). In order to allow this kind of functionality, the FCS framework allows the host to read the current state of the job queue when accepting a job. Using this information, the strategy could determine whether or not the new job could be fit into the system with all the current jobs and be scheduled by its deadline (in general, this problem is NP-complete [**?**], so a reasonable strategy would probably decide to reject some combinations of jobs which could be scheduled together). Since in order to accept the job the host would have just found a plausible schedule, it could store this schedule in the database. The scheduler could then read it and use it as a starting point from which to decide which jobs to launch next (we wouldn't necessarily want to use the exact schedule suggested since some jobs may have finished early or may have been killed by whoever submitted them). Similarly, the scheduler could store its planned schedule in the database so that the host could use it to help figure out whether or not a job can be run by its deadline. Using these features of the FCS framework, the work of writing a deadline-based scheduler can be easily accomplished.

Chapter 4

Related Work

In the previous chapter, we saw how the FCS framework can be used to implement a wide variety of scheduling strategies. In this chapter, we will compare it to other schedulers and show how one might go about implementing their key features using FCS.

4.1 PBS

Like FCS , the PBS family of resource managers (PBS [?], OpenPBS [?], and Torque [?]) try to provide a way for owners of clusters to implement their own schedulers. The PBS resource manager acts as in interface for users to access the cluster. It can accept jobs into a queue and let users view the status of the queue. It also provides a default FIFO scheduler. This scheduler communicates with the resource manager through sockets using a well-specified protocol. The administrators of the cluster can write their own scheduler which uses this protocol to communicate with the resource manager. The scheduler reads the state of the queue, makes a scheduling decision, and informs the resource manager of its decision.

This behavior is similar to that of the FCS framework. However, FCS has some key advantages over PBS. For one, PBS provides no easy mechanism for allowing system administrators to write a strategy which can understand more resource requests than those in the default set. For example, there is no way for the writer of a scheduler to allow users to specify "-jobtype charm" or "-deadline 3:00" options. As we saw in Chapter 3, using this feature of FCS we were able to implement more advanced strategies where the scheduler understood to run-time system of the applications and used this to make more advanced scheduling decisions.

In addition to the above advantage of being able to specify extra job options, the object-oriented design of FCS makes it easier to implement different strategies. To write a different scheduler for PBS, all message types must be handled by the scheduler. While it is true that the code for parts which do not need to be changed from one scheduler to the next can be reused, the FCS framework makes this code reuse more explicit through the `Job` and `Strategy` classes. As we saw in Chapter 3, a wide variety of scheduling strategies could be easily implemented with just a few minor changes in isolated locations. This certainly makes the process of implementing strategies simpler and is a key advantage of FCS over PBS.

4.2 Maui and Moab

The Maui Cluster Scheduler [?] is intended to work with the Torque [?] resource manager (as mentioned above, Torque belongs to the PBS family and shares its API). It comes with a wide variety of scheduling policies to try to accommodate different scheduling needs. Using the techniques described in Chapter 3, these scheduling policies could be implemented using the FCS framework by creating the appropriate strategy classes. If a policy found in the Maui scheduler meets the needs of a particular organization, then there is no good reason not to use it. However, using FCS, once we have the strategies implemented, we can easily tweak them to have policies which are customized to a higher degree.

The Moab Cluster Suite [?] is the successor to Maui. Among other improvements, it provides a large set of graphical tools which help administrators monitor the state of the cluster and the queue. Currently, FCS does not provide this. However, due to the design of FCS, comparable standalone applications could easily be written. In Section 2.1.5, we mentioned how extra components could be added to the framework. By requiring that all information about jobs and the state of the cluster be stored in the database, we left the possibility for such components to have easy access to all the information the other components have. By simply reading and writing the database, we can make graphical tools similar to those in provided by Moab. These tools could even be strategy-dependent, allowing administrators to monitor and change information contained in strategy-dependent fields of the `Job` classes.

4.3 LSF

The LSF License Scheduler [?] provides a very interesting feature: it takes into consideration which jobs require licenses for particular applications when making scheduling decisions. It is often the case that organizations will use software which is too expensive to buy for all their employees, but not all their employees will need at once. Software vendors accommodate this by allowing these organizations to install their software everywhere, but control how many instances of it can be run at once by having the application contact a server and ask for a license before running. If such an application is used by batch jobs and the scheduler is unaware of it, it might schedule more jobs that require the application than the organization has licenses. If the application is configured to wait until a license becomes available, cluster resources would be wasted by blocked jobs. If the application exits when no license is available, users might quickly become frustrated as their batch jobs keep failing due to a lack of licenses. The LSF License Scheduler allows jobs to specify which licenses they need and only schedules jobs which can get the licenses they require. It even takes into account job priorities so that more important jobs get the licenses they require sooner.

The behavior could be implemented using the FCS framework without too much difficulty. We already saw how to make the scheduling strategy understand additional parameters in Section 3.3. We could use the same technique to add a "-license" flag so that jobs can specify they require certain licenses. Using the same technique we used for storing the information about long jobs in the `LimitFIFO` strategy in Section 3.2, we could add cluster state tables to the database showing how many licenses are available for each application. Combined with the techniques for specifying maximum priorities for jobs from Section 3.3, we could specify the maximum priority each user can request for jobs using each type of license. Although a bit more involved than some of the simpler strategies, we can see that the FCS framework provides all the necessary machinery for duplicating this feature of the LSF License Scheduler.

Chapter 5

Conclusions and Future Work

5.1 Summary

This thesis gave an overview of the design and implementation of the FCS framework. This framework provides a flexible platform with which researchers and cluster-owners can easily implement different scheduling strategies. In Chapter 2, we discussed the design of the system. We saw that the system is composed of several stand-alone applications that communicate through a shared database. These components are implemented using the `Job` and `Strategy` classes. In Section 2.2 we saw how these classes interact to access the database, schedule jobs, monitor their progress, and perform a variety of other tasks. In Chapter 3, we saw how to override the behavior of these classes to implement a variety of different strategies. We saw how to implement strategies based on submission time, priorities, and deadlines. We also saw how to take advantage of virtual machines and run-time systems to checkpoint and restart, as well as shrink and expand jobs to make better scheduling decisions. Chapter 4 highlighted some of the feature of other cluster schedulers and how one might implement those using the FCS framework. Through this process, we saw how the FCS framework provides simple, yet powerful ways to change the the behavior of the scheduler. We saw how to change the interactions with the database, the scheduling policy, the commands understood by the scheduler, and various other capabilities.

5.2 Future Work

With the framework developed, the next order of business is to implement different scheduling strategies. Simple FIFO and simple priority-based strategies have already been implements. The `LimitFIFO` strategy described in Section 3.2 should take very little work and help test different capabilities of the framework. Implementing the more advanced deadline-driven strategies and strategies relying on checkpoint and restart or shrink and expand will further test the robustness of the design. During this process, it might become apparent that some parts of the system are not as flexible as others and need to be redesigned. However, due to the overall design of the system and its separation into different applications, these changes should have a minimal and confined impact on the system.

With different strategies implemented, a whole new realm of possibilities for research becomes available. By testing different strategies under real workloads, we could study which provide the highest throughput of jobs, the shortest waiting times, and a wide variety of other variables. To make the research process faster, a simulator which uses the same `Strategy` and `Job` classes could be provided. Using this simulator, system administrators could write a few different strategies, see how they perform under simulated workloads, and install them on their clusters with minimal efforts.

As mentioned in the first chapter, one of the original goals of the Faucets system was to provide a way for different organizations to share their clusters with each other. These clusters would form a federation to which users could submit their jobs. The cluster which could best meet the requirements of the job would then run it. With the new FCS framework, implementing different strategies for different clusters should become much easier. Also, adding components to the system is simple since they can communicate with the other components through the database. We would like to revive the original goals of the Faucets system and meet them using the FCS framework as a foundation.

One very important piece of work which still needs to be done is deployment of the system. The current plan is to implement the LimitFIFO strategy and deploy the system on the Architecture Cluster mentioned in Section 3.2. This will help test the core functionality of the system and will undoubtedly help make it more robust. Once this is complete, we hope to spread the use of the system, and lay the foundation for the development of a cluster-sharing federation as we intended with Faucets.

5.3 Concluding Remarks

As small, dedicated clusters keep being built, more varied scheduling solutions will be needed. The FCS framework provides a platform for the administrators of these clusters to implement scheduling solutions to meet their organization's needs. It is a flexible, robust system capable of being changed in subtle or complex ways. It provides the capability for overriding all the functionality of the system, as well as for adding future components. With proper maintenance and publicity, the FCS framework could become the standard choice for cluster scheduling.