Cluster Middleware

Avikalp Srivastava Anurag Bhardwaj Madhav Datt

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
{avikalp22, chatrasen, madhav}@iitkgp.ac.in

Abstract

Computing clusters can be developed by orchestrating a networked collection of inexpensive commercial off-the-shelf workstation computers or computing nodes. These can be designed to achieve very high levels of performance, greater computing power, better reliability, availability, and fault tolerance as compared to traditional systems. Such clusters can be made to function approximately like a single large cohesive computational resource through the use of a software layer, called cluster middleware, for system resource management.

In this paper, we extensively analyze the design of cluster middleware systems, examine design challenges and discuss trade-offs involved in job submission, job-system matchmaking, static and dynamic load balancing, job runtime limits, process preemption and migration, avoiding starvation, and fault tolerance. We then present a comparison between various approaches to addressing these challenges and their implementations. Further, as case studies, we go on to study the design philosophies, architectures, design decisions, implementations, and limitations of two real life cluster computing systems - Condor and Google Borg.

1 Introduction

Workstation clusters formed using relatively inexpensive commercial off-the-shelf workstations are being extensively used for distributed computing, in both academic and industrial applications. Cluster computing provides many advantages over traditional computing systems. For example, they provide for scalability by allowing additional nodes to be added to the system, to improve fault tolerance, performance, availability and redundancy, as an inexpensive alternative to scaling up single very powerful nodes or vector supercomputers. Clusters connected with high speed local area network may also be used as inexpensive parallel computers. Another popular use includes running batch jobs, where clusters can provide faster turn-around time as compared to supercomputers [7]. A number of computing nodes (for example, personal computers, dedicated servers etc.) are connected together over a network to form a computing cluster that can function, to a reasonable extent, as a single cohesive computing unit with the use of system management

software layer. The activities of computing nodes are managed by this software layer referred to as "cluster middleware".

Popular middleware systems for cluster computing include systems such as Condor [12], Borg [13], Portable Batch System [6], NEST [4] etc. Condor works on the principle of distributing jobs around a loosely coupled workstation cluster, attempting to use idle CPU cycles available on some machines to provide additional computing resources to other users, thus working on the concept of users donating their idle CPU cycles ("cycle stealing") while still maintaining instant access to their machines/personal computers. IBM LoadLeveler [11], which builds on Condor, is focused on maintaining a balanced load, optimal resource usage and fairness in job scheduling while allocating computing resources. Resources are allocated using a greedy method, where jobs are submitted to the computing node that meets all user specified requirements, but also has the lowest load/resource usage. The Portable Batch System (PBS) provides an external scheduling module to overcome limitations of scheduling jobs using sets of queue controls. This separate scheduling program has complete information of all the available jobs in the queues, currently executing jobs, and resource usage at systems. This allows for policies to make scheduling decisions over the full set of jobs "regardless of queue or order" [6]. CODINE (COmputing in DIstributed Networked Environments) brings together the queuing framework of Distributed Queuing System and leverages Condor's checkpointing method to try and ensure optimal computational resource utilization in heterogeneous environments (i.e. the computing nodes may have varying architectures and operating systems). It is particularly effective in case of workstation clusters with integrated servers such as vector supercomputers, CPU-GPU clusters etc.

In this paper, we analyze the design decisions and challenges involved in building cluster computing systems. Particularly, we look at the decisions and trade-offs involved in designing components for job submission, job-system matchmaking, static and dynamic load balancing, job runtime limits, process preemption and migration, avoiding starvation, and handling node and central server failures. We then compare a variety of cluster management/middleware systems based on their approaches to addressing these design challenges and implementation. Further, we go on to analyze and discuss the design decisions, architecture, trade-offs, and limitations of cluster computing systems, Condor [12, 8, 3] and Google Borg [13, 2, 10] in detail, as case studies.

2 Model Architecture

Computing nodes are connected over a local area network and are in a fully connected topology. A central master node to schedule and manage jobs may exist. We primarily examine systems where a central master node/server is present and is used for job scheduling, load balancing etc. (i.e. not fully distributed load balancing models). Jobs may be submitted by the user at any computing node, including the central server. Both user and jobs may have priority values associated with them. This would allow the scheduler/load balancer to take the urgency/importance of a user/job into account while scheduling the job's execution. The system is made fault tolerant with respect to crash faults only. Specific fault models, approaches and trade-offs involved are discussed in section 3.2.

Also, the user may specify, either directly through an interface or using a job submission file, the computing resources required for the submitted job. This would allow the user to request a specific amount of memory, swap space and disk space, or specify preference for a particular

computing node. In the event that the resources requested by the user are too large, the job gets delayed till the required compute resources become available, either through the preemption of lower priority jobs or completion of other jobs. Jobs might have dependencies, which can be specified through makefiles and included with the job submission file.

3 Design Challenges and Approaches

3.1 Scheduling and Load Balancing

On a single machine, the execution of a job can be started as soon it is submitted if the system is idle. However, extra efforts need to be taken in a cluster to find an appropriate candidate machine. Several factors need to be considered while making this decision.

3.1.1 Job Submission

Based on the underlying architecture, multiple job submission models can be used. Allowing distributed job submission involves accepting jobs at each node, while a centralized submission requires direct communication with a central node for any job submission. Another issue is whether to provide fully distributed handling, where no central node exists, or to provide distributed submission along with a central manager. Absence of a central node requires each machine to monitor the state of neighborhood machines, and make decisions on matching jobs to resources, load balancing, and job dispatch.

Approaches: First approach is submission to only the central server which keeps the incoming jobs in a queue [13], [1] providing simplicity and reducing communication overhead, but reducing system flexibility and places constraints on user access and architecture. Distributed submission [8] keeps a local job queue at each machine, which in turn communicate with the central server, specifying their idle jobs and availability. The results on job completion can be returned to the submitting machine for user's convenience.

3.1.2 Matchmaking: Priority and Quota

Submitted jobs to the cluster can involve differing priorities, requiring system decision policy on how to handle and share cluster resources based on these priorities. Along with this, users and user groups can also have different priorities, which brings issues such as deciding the hierarchy, ensuring fair share, strict ordering, or fractional ordering depending on application. Each job, considered in the order decided by priority, is matched for execution on a member machine. The issues here include ensuring that the requirements of the job (such as memory, disk space, cpu cores required etc.) are met with the resources available at matched machine. Handling job preferences is another issue, where a policy for selection among candidate machines is required. All this is to be done while ensuring a load balancing policy that helps achieve high throughput.

Approaches: A priority queue for jobs is most common, and jobs are considered in this order for dispatch [5]. Eligible systems are found by matching the hard requirements of the job with each machine's resource description [8] [2]. Ranking among eligible stations can be done in different ways - by taking into account the preference policy described by the job and selecting the machine

providing just enough resources, or the one providing best possible set of resources [12]; by considering a load balancing policy and aptly selecting a candidate machine [1] [4]. Load balancing and job preference can also be used together, with a trade-off policy in place for balancing effects of both.

3.1.3 Static and Dynamic Load Balancing

Load balancing forms a crucial component of cluster middleware, as it provides the benefits of improving performance at each node and resource utilization, reducing job idle time and response time, with higher throughput and reliability [5]. Selection of an appropriate load balancing depends on application requirements, and parameters such as balancing quality, information available from application, load patterns and communication overheads tolerance. Some of the most important issues to consider involve *resource utilization*, *overhead*, *thrashing*, *reliability* and *response time*.

Approaches: Static load balancing (SLB) involves assigning jobs during scheduling according to the performance of the nodes and their current loads. Once the jobs are assigned, no change or reassignment is done. In other words, there is no need for load monitoring, and subsequent reassignments. Dynamic load balancing (DLB) focuses on transferring from heavily loaded nodes to the lightly loaded nodes, depending on periodic information about system state and load. It is especially advantageous in heterogeneous system consisting of nodes with variable speeds, communication link speeds, memory sizes, and different external loads.

Multiple trade-offs can be discussed with these two strategies in the general case. The system incurs a lower *overhead* with SLB, which is higher with DLB due to process redistribution. *Resource utilization* is better with DLB compared to SLB, *thrashing* possibility is much lower/non-existent in SLB, while it can be a considerable problem with DLB due to multiple migrations. While DLB helps with more effective resource utilization, it can increase the *response time* for jobs.

3.1.4 Process Preemption and Migration

In a system where jobs have priorities, it is always important to decide if a high priority job can preempt a low priority one. While scheduling a high priority job, we need to consider the trade-off between scheduling it on a low-preference machine (as per user specification and the matchmaking policy) and preempting a low-priority job on a high-preference machine. Also, appropriate measures need to be taken to reschedule the preempted job.

Approaches: The preempted job can be stored in the stable storage of the machine where it was submitted and rescheduled later on the same machine. It can also be sent to the central server which can then schedule it on an appropriate machine. The job can either be restarted from the beginning, from a checkpoint, or based on the number of time-quanta for which it has had access to computing resources. This involves a trade-off between storage space and execution time.

3.1.5 Fairness and Starvation

While scheduling jobs on computing nodes, it might be necessary to maintain some notion of fairness within the same user and job priority levels, in terms of compute time/resources allocated

to jobs and users. The middleware system must also ensure that low priority processes do not get repeatedly preempted, that is, there should be no starvation in jobs.

Approaches: Fair-share scheduling among users at the same priority level could be used to ensure fairness among users. A round-robin scheduling strategy, where each job is given computation resources for a fixed time-quanta before being preempted and added to the back of the pending queue, could be used to introduce fairness among jobs at the same priority levels. To make sure there is no starvation in low priority jobs due to repeated preemptions because of higher priority jobs, the priority level of a job can be incremented each time it is preempted.

3.2 Fault Tolerance

Cluster middleware systems are used by production critical applications with millions of clients. Hence, even a small fault (crash failure) may have drastic consequences on the working of the system. It therefore becomes extremely important to make these systems fault tolerant. In this section we will be discussing the major problems involved in detecting and resolving faults and some approaches to solve them.

3.2.1 Node Failure

The first kind of failure we will be discussing is the crashing of a non-central node. At first, we need to know that a node has failed. In this detection step, there is always a trade-off between the time required for detection and the amount of network traffic generated because of the detection measures. If a central server is present, it needs to know that a particular node has failed and has to take measures to reschedule the jobs running on that node. While rescheduling, it has to decide whether to restart the jobs from the beginning or to start them from a certain checkpoint. This is again a trade-off between time and space. On the other hand, handling a node failure in a completely distributed system will either require nodes to remember which node failures they have to handle or there will be a huge communication overhead to resolve the fault.

Approaches: For failure detection, the central server can periodically send a heartbeat message to all the other nodes and if they fail to reply, it knows that something is wrong. This approach can be extended to work in the case of a completely distributed system. The node can also send its checkpoint along with the reply which the central server can use if the node fails. However, as the chance of failure of a node is very low, it is better to restart the jobs running on the failed node that bearing such heavy communication overheads.

3.2.2 Central Server Failure

The entire system is dependent on the central server and hence its failure can totally disrupt the system. The first step here as well is detection. The central server failure will lead to an enormous information loss and the system needs to handle it. The jobs sent between failure and its detection also need to be handled.

Approaches: Shadow master scheduler nodes or passive replication may be used [9]. The same approach for detecting failure of a node can also be used in this case, where the central server sends out a periodic heartbeat to its primary backup/replica servers. If this heartbeat message is not received for a certain period of time, the central server is assumed to have failed. After the central

server fails, a new leader can be elected among the remaining replica nodes, which can then start acting as the new central server. To handle the jobs sent after the failure, client would need to send missed jobs again after the new central master node starts functioning.

3.2.3 Node Recovery

After the failed node recovers, appropriate measures need to be taken again to make it a part of the system. The recovered node needs to inform the central server so that it can start considering it for task execution. In case of a completely distributed system, the neighbors of the recovered node need to be updated. If the recovered node was a central server, it needs to take appropriate measures to be a part of the passive replication system again.

Approaches: Addition of a non-central node to the system can be handled easily by communicating with all its neighbors. However, if a central node recovers, and the system has passive replication, the node needs to find out who the new leader is, and makes its state consistent with the leader. If our fault model assumes that there can be more than one fault at a time, every node must find out who the new leader is after it recovers.

3.3 Additional Issues

Dynamic Cluster: This includes support for addition and withdrawal of nodes from the cluster. Approaches similar to handling crash and recovery of a node can be adopted with changes.

Support for Multiple Job Models: User jobs can be serial or parallel in nature. Parallel job models include support for parallel virtual machines (PVM), and message passing interface (MPI).

Data Locality: The architecture can make use of a shared filesystem such as NFS, AFS. These, however, can degrade the performance as they must transfer information over a physical network, and reduce HTC. In case of no shared filesystem, remote system calls, file staging, file transfer protocols or RCP programs can be used.

Heterogeneous Support: Support for computing environment consisting of a number of computers with dissimilar architectures and different operating systems.

4 Case Study: Condor

Condor is a high throughput computing (HTC) system providing workload management for compute-intensive jobs. It provides the basic functionalities of job queuing, scheduling policy, priority scheme, fault tolerance with checkpointing, resource management and monitoring. Condor runs as a user-level process monitoring usage of each node in the pool, and matches waiting resource requests with idle resources. In addition, it provides fault tolerance in case of machine crashes or hardware failure through checkpointing running jobs.

An interesting and advantageous aspect of Condor is its ability to not only manage a cluster of dedicated beowulf nodes, but also to effectively harness the computation power of idle desktop workstations in the case non-dedicated nodes form a part of the cluster as well. The Condor system

only uses idle machines, and migration takes place when a non-dedicated machine is claimed by its user, limiting the throughput achieved by Condor.

4.1 Architecture

4.1.1 Cluster Setup

The Condor pool consists of a single machine acting as the central manager, along with an arbitrary number of other member machines. As mentioned earlier, these member machines can be both dedicated (eg. Beowulf nodes) or non-dedicated (eg. desktop workstations). Effectively, the pool is just a collection of resources (machines) and resource requests (jobs). The user can submit jobs to any node in the cluster, with each node maintaining its own local job queue.

The central manager is connected with every other node, and each node of the system sends periodic updates to this central manager. The manager also periodically assesses the current pool state and carries out matchmaking and scheduling.

4.1.2 Job Description

A job is described as the most basic unit of computation request submitted by a user, as compared to some other middle-ware systems where jobs can further be divided into subunits such as *tasks*. An application to be run in the Condor system must be able to execute as a batch job. A Condor job is characterized by the following traits:

(a) Each job has a priority number (-20 to +20), along with its submitting user's priority. The user's priority is pre-decided by the system admin.

(b) Should specify all jobs it is dependent upon. A job j_1 is said to be dependent on another job j_2, if completion of j_2 is necessary for successful execution of j_1.

(c) *Submit Description File*: Each job is accompanied by its description file, containing information such as what executable to run, files to use for logging, standard input & output, and the *ClassAd* data structure described later. Briefly put, the ClassAd data structure provides the resource requirements and preferences of the job for future matchmaking with a machine.

4.1.3 Condor Daemons

Daemons are background server processes running on each node of the pool. Depending on the role of each node, a different combination of daemon processes will be running on them. These processes are responsible for handling communication, job scheduling, spawning user application on execute machine, pool monitoring on the central manager etc. The specifics about the multiple daemons and their roles are described in section 4.2.2.

4.1.4 File System

Condor can work in both the presence and absence of a shared file system. However, a distributed file sharing system protocol such as the network file system (NFS) or the Andrew file system (AFS) can significantly limit the number of eligible workstations, and hence can hinder high throughput

computing. To maximize throughput, Condor runs any application on any remote workstation of a given platform, without relying on a common administrative setup. This also provides the benefit that Condor doesn't require a user to possess a login account on the execute workstation.

4.2 Associated Data Structures and Procedures

4.2.1 The ClassAd Data Structure and Matchmaking Algorithm

ClassAds are structures associated with both a job and a machine, and serve as detailed descriptions of the characteristics and constraints of the job/machine they are associated with. Using named expressions with attribute name and attribute values. ClassAds allow for matchmaking - a process where idle resource requests are matched with idle resources by matching requirements in the ClassAd, and by ranking ties using preferences in the ClassAd. We make the concept clearer using the example shown in table 1.

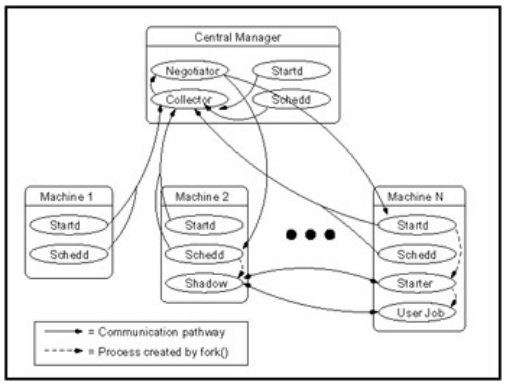
Table 1: Structures at the Central Manager for Matchmaking

Structures at the Central Manager		
Job ClassAd	Machine 1 ClassAd	Machine 2 ClassAd
Requirements: (OpSys = 'Linux'; Arch = 'Intel' Disk > DiskUsage)	**OpSys** = 'Linux' **Arch** = 'Intel' **Disk** = 3,076,706	**OpSys** = 'Linux' **Arch** = 'Intel' **Disk** = 2,016,502
Rank = Memory * 1,000 + KFlops	**KFlops** = 145,811	**KFlops** = 142,811
DiskUsage = 465	**Memory** = 511	**Memory** = 500

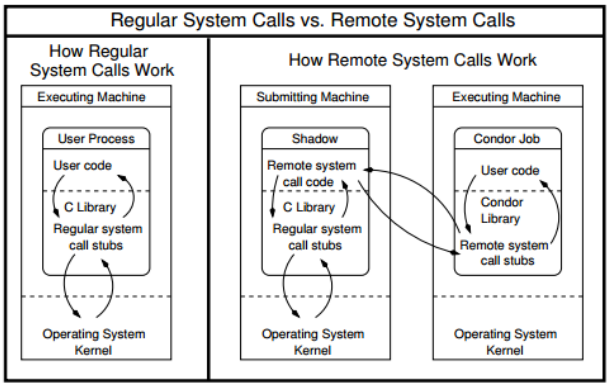
Here, the hard *requirements* of the job are first matched with each idle machine. Machine_1 and Machine_2 both satisfy the constraints, and therefore *rank* measure, based on a linear combination of *Memory* and *KFlops* (a metric of floating point computation performance) is used, where Machine_1 is found to be preferred over Machine_2, and is therefore assigned the job for execution.

4.2.2 Condor Daemon Processes

condor_master: Runs on every machine in Condor pool, simplifying system administration. It spawns and monitors all the other daemons running on the system, and restarts any faulty ones.
condor_startd: Represents a machine, and advertises its ClassAd. Running *condor_startd* enables a machine to execute jobs. It is also responsible for enforcing the policy under which remote jobs will be started, suspended, resumed, vacated or killed. When the *condor_startd* is ready to execute a job, it spawns *condor_starter*
condor_starter: Spawns remote condor job on a given machine. It sets up the execution environment, plus monitors the running job. Also detects job completion, sends back status info to submitting machine, and exits.
condor_schedd: Represents a job. Any machine accepting job submissions needs to have a *condor_schedd*. Users submit jobs to *condor_schedd*, where they're stored in a local job queue.
condor_shadow: Runs on the submitted machine, whenever that job is executing. Serves requests



(a) Inter-Daemon communication in Condor



(b) Remote System Call procedure internals

Figure 1: Communication and system calls in Condor. Figures from [12].

for files to transfer, logs job's progress, reports on completion.
condor_collector: Runs only on the central manager, dynamic database of classAds. Collects information about pool status. All other daemons periodically send ClassAd updates to the collector.
condor_negotiator: Runs only on the central manager. Responsible for matchmaking and enforcing user priorities, through communication with *condor_collector*.

4.2.3 Remote System Call Procedure

To provide access to a job to its data files, using file I/O calls such as *open()*, *read()*, *write()* etc., Condor needs a communication mechanism between the execute and the submit machine. This is achieved using the following procedure:

(a) Users need to link their program with Condor's caller library using command line interface.

(b) This library augments certain system I/O call function stubs to make them behave as remote system call stubs.

(c) The remote system call stub send a message with the system call number and apt arguments over the network to the *condor_shadow* process on submit machine.

(d) The shadow process carries out the system call on behalf of remotely running job, and sends the results via message to the remote call stub on the execute machine.

(e) The remote call stub returns the result, unaware that the call took place remotely, not locally.

4.2.4 Process Checkpointing Procedure

Periodic checkpointing takes place at nodes running jobs. Checkpoints involve snapshot of the program's current state, requiring the program to be relinked with Condor system call library.

Taking a checkpoint is implemented in Condor system call library as a signal handler; when condor sends a checkpoint signal to a process linked with this library, the provided signal handler

writes the state of process out to a file or network socket. The program state includes process's stack, data segments, all CPU state, register values, state of open files, all signal handlers and pending signals. By default, a checkpoint is written to a file on local disk of the submit machine. Condor may also be configured to have a checkpoint server to serve as a repository for checkpoints.

4.3 Condor in Action

In this section, we explain the procedure involved from submission of a job to its completion. The simplest path procedure is explained, while the mechanisms for events such as preemption and checkpointing have already been described in the section 4.2, and the conditions for those events have been discussed in section 4.4.

The steps involved in the process from job submission to its completion are as follows:

1. The job submission takes place via the command line interface, which reads a job description file, creates the job's ClassAd, and gives that ClassAd to the *condor_schedd* process managing the local job queue on that node.

2. The *condor_schedd* process, in turn, communicates with the central manager, triggering a negotiation cycle.

3. During the negotiation cycle, *condor_negotiator* queries *condor_collector* to discover all machines willing to perform jobs, and all users with jobs waiting for execution.

4. *condor_negotiator* communicates in user priority order with each *condor_schedd* with idle jobs in queue, and performs matchmaking via the algorithm described in section 4.2.1.

5. Once *condor_negotiator* makes a match, *condor_schedd* of the submitting machine claims the corresponding machine, and makes subsequent scheduling decisions about the order in which to run jobs.

6. When *condor_schedd* starts a job, it spawns a condor_shadow process on the submit machine, and *condor_startd* spawns a *condor_starter* process on corresponding execute machine.

7. *condor_shadow* transfers the job's classAd and data files required by *condor_starter*, which spawns the user's application on the execute machine. The *condor_shadow* process also serves RSCs as described in section 4.2.3.

8. On job completion/abortion, *condor_starter* removes every process spawned by the user job, and frees any temporary space used by the job.

4.4 Analysis of Design Decisions

4.4.1 Job Submission

Condor supports distributed job submission, meaning that each node in the cluster can accept job submissions, along with allowing a user to submit a job to the pool via their own desktop machines.

In case of a uniformly owned cluster of beowulf nodes, direct submission to the central manager takes place.

Advantages: Distributed job submission allows for ease of expanding the cluster, allowing inclusion of multiple local disk systems, submission sites, and more resilient safeguarding of data as compared to all job information being stored on the central server. It also allows to implement hierarchical priority scheme, where the central server enforces user priority, while each local submission machine then handles individual job priority.

Disadvantages: Compared to purely distributed submission and processing, presence of a central manager implies that all nodes need to be in constant and periodic communication with the central node, leading to communication overheads and network traffic. The system can incur heavy computation losses in case of a single point failure at the central manager, and requirement of passive replication for fault tolerance at central manager.

Decision Rationale: Since Condor was set up with the goal of utilizing compute resources of member workstations, while providing the services of the cluster to owners of the member machines, distributed submission is the ideal choice. The output results are returned to the submitting machine for user's convenience. A central manager is required for monitoring all idle stations and jobs. The central manager is designed to be highly reliable, to reduce the frequency of applying the heavy computations required to recover from single point failure at it.

4.4.2 Priority Scheme and Matchmaking

Condor implements a hierarchical, distributed scheduling architecture. In presence of two levels of priorities: user and job, the central scheduler enforces user priority and assigns an execute machine to the submitting machine, and the submitting machine then enforces job priority to control job execution order. The matchmaking ranking process has been described earlier.

Advantages: The hierarchical distributed priority scheme enhances Condor's scalability and flexibility. Fair share scheduling can then be imposed on top to ensure priority weighted distribution of resources to the users.

Disadvantages: Hierarchical ordering implies that a much higher prioritized job from a lower priority user will always be considered after even a very low priority job from a higher priority user.

Decision Rationale: Considering the public nature of access to Condor to users with differing importances within the system, it is important to have an initial priority over the users (assigned by the administrator). A very basic example is segregation into two user types - one submitting production level jobs (higher priority), and another submitting non-production level jobs.

4.4.3 Scheduling Policy and Preemption Scheme

Starvation of lower priority users is prevented by a fair share scheduling algorithm, which attempts to give users priority weighted machine allocation time over a specified interval. Priority calculations are based on ratios opposed to absolutes. So, if user A has priority twice compared to that of user B, then instead of starving user B by assigning machines to jobs by A, Condor will try to assign user A twice as many machines as user B on average.

Under its default preemption scheme, Condor doesn't allow preemption of lower priority jobs by higher priority ones, and instead will wait for a machine to become idle.

Advantages: Fair sharing prevents the starvation problem, and is effective in allocating resources in a reasonable method. Another point is that fair sharing algorithm can be applied at multiple levels of abstractions such as groups, users, processes etc. Absence of priority based preemption reduces overhead of checkpointing and migration, and simplifies implementation. It also eliminates possibility of problems such as thrashing, and improves system reliability.

Disadvantages: Fair share scheduling increases the response time for the jobs. Some fair share algorithms can be prone to bias to users with more submitted jobs, causing other users to wait for longer than expected time. For an application requiring strict ordering of user jobs, fair sharing fails to meet this requirement. Absence of priority based preemption means high priority jobs can be waiting behind lower priority ones. It also eliminates the possibility of imposing CPU run time limits to prevent large jobs from significantly increasing response time for smaller jobs.

Decision Rationale: Fair share scheduling prevents starvation while ensuring all resource requests get access to resource within a reasonable response time. This is a good design choice for Condor which includes users with varying priorities and multiple job submissions. Priority preemption is a design choice keeping in mind a lot of factors such as overhead, thrashing etc., however Condor provides an option to allow priority preemption as well, allowing administrator to configure according to application needs.

4.4.4 Handling Node Failures and Recoveries

The central manager periodically pings the member nodes to detect any crash faults. Along with this, periodic checkpointing is carried out to minimize computation loss in case of a fault at a machine running a job. The job is then simply put back in central manager's queue, and on dispatch, the submitting machine sends the checkpoint state data stored in its local disk to the execute machine. The job's state is then restored to this checkpoint and computation is continued. For node recovery, the rejoining node simply pings the central manager, which then adds it to the node set.

Advantages: Periodic checkpointing minimizes computation loss, specially for large jobs. There is a trade-off when deciding the time between checkpoints, between reducing computation losses and limiting network traffic.

Disadvantages: Simple checkpointing procedures lead to several limitations pertaining to parallel nature of jobs, and read-write of files. These are discussed in detail in section 4.5. Periodic checkpointing increases network congestion, and the storage of checkpoints use up local disk space on submitting machines. Procedure for fault at central manager is not specified by Condor, however, we can assume it to be similar to the techniques used in other systems, such as replication.

Decision Rationale: Considering Condor's standard universe, which is used only for serial jobs, many limitations of checkpointing are moot, while providing a simple method for both fault tolerance and migration. Network congestion is again a trade-off, and Condor can be configured to sacrifice fault tolerance by turning off checkpointing, with smaller network delays and traffic.

4.4.5 Tackling Data Locality

The following trade-off decisions can be made based on underlying hardware architecture:

(a) Condor can utilize a shared filesystem such as NFS or AFS permanently mounted across machines in the pool.

(b) In case of non-dedicated pool machines and shared filesystem, the job requirement file can specify that job should be run only on machines sharing the filesystem with submitting machine, or the requirement file can specify usage of Condor file transfer protocol for the same.

(c) In absence of shared filesystem, the system utilizes remote system call procedure described in section 4.2.3 to solve the problem.

Advantages: Can work in both the absence and presence of a shared file system, RSC procedure allows remote execution without requiring login credentials. File transfer protocol adds to the flexibility of the system.

Disadvantages: Shared file systems can degrade performance due to communication overheads and reduce high throughput capacity. RSC procedure requires users to re-link their programs with Condor's caller library.

4.5 Limitations

In this section, we explain some of the most important limitations of Condor. Please note that certain finer drawbacks have not been discussed for brevity.

(a) Due to the design decision of only selecting idle workstations for job dispatch, Condor lacks a load monitoring and balancing scheme. This can mean large queue of jobs at the central scheduler, increased response time, as well as assignment of job to a suboptimal machine.

(b) Since Condor relies on checkpointing for saving job states to be used in case of node crashes and job migration, the following limiting constraints need to be insured for implementing the simple checkpointing procedure used by Condor encompassing heterogeneous systems:

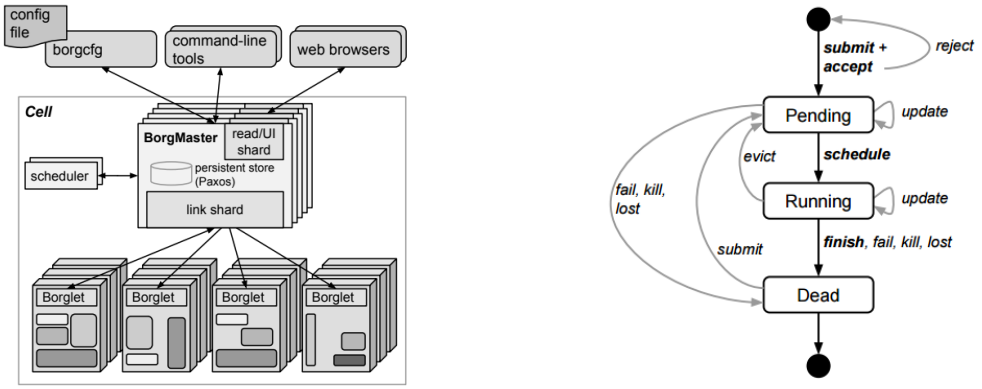
- Only single process jobs are allowed (eg. *fork()*, *exec()*, *system()* etc. not allowed).
- Signals or signal handlers are not supported (eg. *signal()*, *sigvec()*, *kill()* etc.).
- Interprocess communication is disallowed (pipes, semaphores etc.).
- Network communication should be kept as brief as possible. For example, a job making network connections using system calls such as *socket()*, delays in closing the network communication can adversely affect checkpointing and migration processes.
- All files must be accessed either 'read-only' or 'write-only'. If accessed in 'read & write' mode, then there are problems if a job must be rolled back to an old checkpoint image.

(c) Jobs must be linked to Condor's caller library for migration and checkpointing purposes.

(d) No CPU run-time limit exists for a job. This means that no preemption scheme exists to ensure fair sharing among large and smaller jobs, which can lead to small jobs getting stuck because of a similar priority large job.

(e) No support for parallel jobs with fault tolerance due to checkpointing problem.

(f) Recovery from single point failure at central manager requires dealing with the host of challenges brought on by passive replication.



(a) High level architecture of Borg

(b) State diagram for execution of submitted jobs

Figure 2: Borg architecture and job submission (Figures from [13])

5 Case Study: Google Borg

Google's Borg System [13] is a cluster middleware that efficiently runs thousands of jobs across hundreds of clusters. Two types of jobs run on the Borg system: *prod* (e.g. Google docs which are mostly user oriented) and *non-prod* (Google's internal jobs). The main benefit of using Borg is that it hides the details of resource management and failure handling so that its users can focus more on application development. The matchmaking procedure which is a part of its scheduling process helps it achieve high throughput and utilization. Its efficient distribution of tasks over multiple machines reduces the probability of correlated failures. Along with all these benefits, Borg has been made extensively fault tolerant and improves user experience my minimizing the fault-recovery time. Its own declarative job description language, real-time monitoring of jobs and tools to analyze and monitor the system's behavior have contributed to an exceptional user experience.

5.1 Architecture

Among the various categories of cluster middleware discussed earlier, Google Borg falls in the following category: "A centralized server for job submission and control". The entire Borg system is composed of cells. A Borg cell consists of a number of a machines, a centralized controller called the Borgmaster, and an agent process called Borglet which runs on every machine in the cell. Figure 2(a) gives a high level view of the Borg architecture.

5.1.1 Borgmaster

The Borgmaster consists of two main processes, one for scheduling and the other for job-submission and answering queries related to read-only jobs. The existence of these two independent processes significantly improves the performance of the Borgmaster. As the failure of the Borgmaster will totally disrupt the system, 5 replicas of the Borgmaster are present in every cell. Whenever one of the replicas goes down, a leader election is conducted as part of fault-handing and the elected leader serves as the primary Borgmaster to whom the user submits jobs. An important point to

note is that the state of the Borgmaster at any stage serves as the checkpoint of the system. The Borgmaster periodically records a snapshot of its state and stores it in a stable storage. This might be used in case of a recovery.

5.1.2 Borglet

Borglet is a local agent of Borg present on every machine. It starts and stops tasks, takes appropriate measures if they fail, manages local resources and monitors the state of the system. The Borgmaster communicates with the Borglet by sending a heartbeat every few seconds. The Borglet reports everything to the Borgmaster on receiving this message. This message is also useful for detecting that the Borgmaster is still alive. Most of the times, the state sent by Borglet is almost same as the one it sent in the previous iteration. Hence, The Borgmaster just needs to look at the changes and update them accordingly. To do this, every Borgmaster cell runs a *link shard* to handle communication with the Borglet. The link shard informs the Borgmaster about the changes it has to incorporate and thus reduces the load of the Borgmaster.

5.2 Borg in Action

In this section, we will describe all the things that happen starting from a job submission to its completion in Borg. We will be describing a simple run which does not involve a job preemption or the steps involved in checkpointing. The steps are as follow:

1. A client submits the job along with its name, his name and the number of tasks in the job to the Borg system. Jobs can have constraints forcing matches with machines having particular set of attributes (e.g. CPU cores, RAM, disk space, disk access rate, TCP ports etc.).
2. The Borgmaster takes a note of the job in its stable storage and adds it to the scheduler queue if the job can be admitted based on its quota. (More on this in section 5.3.2)
3. The scheduler processes its queue from high to low priority and selects the topmost job which is about to be dispatched for execution.
4. The scheduler then selects an appropriate system for the execution of this job using measures described in section 5.3.3
5. After the job (or rather the job's task) is dispatched, the Borglet of that machine takes over and handles everything related to the job.
6. The Borglet also sends periodic updates to the Borgmaster regarding it state.
7. After the execution of the task is complete, the Borglet informs the Borgmaster, who makes necessary changes to its stable storage, updates the state of the machine which finished execution and reports the results of the execution to the client.

5.3 Analysis of Design Decisions

5.3.1 Job Submission

In Borg, a job is always submitted to the same centralized server (or its replica in case a new leader is elected). Before doing anything else, the Borgmaster takes a note of the job as it has to handle

everything related to it. The submitted job is a collection of tasks which can be independently scheduled on different machines. A client can update the job even after submitting it. Figure 2(b) describes the entire life cycle of a job.

Advantages: As a single server is in control of the entire scheduling process, load balancing can be optimal and the scheduling overheads are minimum. The absence of a distributed submission reduces the communication overheads to a great extent. Also, it is easier to present a global view of the system to the user at any point of time. As the client can update the job at any point of time, he doesn't need to worry about any mistake he made during submission.

Disadvantages: No other machines shares the scheduling load with the Borgmaster. The communication links of the central server are also more crowded and may lead to a network congestion. As the entire info is present in one location, if it goes down, system breaks down.

Decision Rationale: Google uses Borg for a lot of user applications like Google docs, and hence latency is one of the major things it has to focus on. A centralized server based system implies that the communication overheads are kept to a minimum and hence the job gets scheduled probably the fastest among all the models. As Google (arguably) has access to the best servers and network links in the world, all other problems created by a central server based system are not that big of an issue. Also, the replication of the Borgmaster provides an efficient fault handling methodology

5.3.2 Priority and Quota

Each job submitted to the Borg system has an associated priority. Jobs get scheduled from high priority to low priority. A total order is enforced among jobs having the same priority by using a Round-Robin scheme based on arrival times. In case it is not possible to schedule a higher priority task in the system when a lower priority one is getting executed, the former can preempt the later. However, to avoid cascading, preempting a prod task by another prod task is not allowed. Along with priority, the user also demands (or buys) a particular amount of resources called quota. Borg admits a job for scheduling based on its quota. It might happen that at some point of time, the user's job's requirements (after admission) might exceed this quota. Borg handles such cases by reducing the priority of the job making sure that the product of the priority and (modified) quota is constant at every point of time.

Advantages: The priority scheme allows high priority jobs to be completed faster than low priority ones. The priority and quota based scheme ensures that the job of the user always gets executed irrespective of whether the user has the correct idea about the required amount of resources for his job.

Disadvantages: The low priority jobs may be starved due to two reasons: 1)They are not scheduled at all due to incoming high priority jobs, 2)Even if they are scheduled they keep on getting preempted. In some cases, it might happen that the low priority jobs have to give a part of their shareable resources to high priority ones which affects their latency.

Decision Rationale: Not all jobs are of equal importance and hence assigning priorities to jobs solves this problem perfectly. The use of quota for admission control ensures that jobs which don't have any chance of getting scheduled are not even present in the scheduling queue.

5.3.3 Scheduling Policy and Preemption Scheme

After a execution based job is submitted to the Borgmaster, the scheduler takes over. The scheduler operates on individual tasks and not the entire job. It adds the tasks to its pending queue and dispatches them in the order of their priorities. The scheduling algorithm primarily consists of two parts: *feasibility checking* and *scoring*. Feasibility checking involves finding machines which meet the tasks constraints and have all the resources the task desires for. Note that while calculating available resources, Borg also takes the resources already allotted to low-priority tasks into consideration as they might be preempted to make way for the higher priority task. After finding the feasible machines, "goodness" of each machine is determined. This phases is known as scoring. The scores depends on the user-specified preferences for sure but other criteria which affect it are: minimizing the number of preempted tasks, picking machines which already have the packages required by the task installed on them, spreading tasks across power and failure domains and distributing high and low priority jobs uniformly among all the machines. For generating a score, both best-fit and worst-fit strategies were tried, however, both have their own pros and cons. Hence, Borg uses a hybrid strategy for scoring, one that tries to minimize the amount of stranded resources i.e. resources which cannot be used because another resource on the same machine is fully allocated.

If a machine selected after the scoring phase doesn't have the necessary resources (at that moment), some low priority tasks are preempted. The preempted tasks are added back to the pending queue of the scheduler and are not hibernated or migrated somewhere else.

Advantages: A two-stage scheduling policy eliminates the need to score all the available machines. Compromising on user's preferences (at some stages) significantly improves the performance of the system as a whole to a great extent. Operating on tasks instead of tasks ensures that only a part of the job gets affected even if there is a fault in some part of the system. The distribution of tasks across power and fault domains better equips the system in case of a fault.

Disadvantages: The low priority jobs may starve. Even if they are scheduled quickly, their turnaround times might be pretty large in situations when they are preempted. From the user's point of view, he might not be able to get the expected performance as his preferences are overlooked at times by Borg.

Decision Rationale: The major disadvantage of this scheduler seems to be the starvation of low -priority jobs. However, Borg tries to solves this problem in the scoring phase. One of the major factors involved in scoring is the even distributing of high and low priority machines across all machines. Due to this, the high priority jobs are completed faster than if they were concentrated on a single machine and thus lower-priority jobs have a better chance of getting scheduled. This factor also improves the turnaround times for high priority jobs. Also, as the scheduler tries to minimize the number of preempted tasks, the starvation of a task due to preemption is eliminated to a great extent.

Enhancements: The following three things have significantly improved the performance of the scheduler:

- *Score Caching:* Instead of calculating the scores for all the tasks again and again, Borg caches the scores until properties of a task or a machine change.
- *Equivalence classes:* Borg classifies tasks having identical requirements and constraints into an equivalence class. Feasibility checking and scoring are done only once per class.

- *Relaxed Randomization:* The scheduler examines machines in random order until it finds out a sufficient number of candidates for the given task instead of examining all the machines.

5.3.4 Handling Node Failures and Recoveries

Fault detection and recovery depends entirely on the heartbeat message which the Borgmaster sends to the Borglets. Similar communication also takes place among all the Borgmaster replicas. If a Borglet crashes, the Borgmaster detects it and reschedules all the tasks which were running on it from the last updated state for the crashed machine. On the other hand, if the Borgmaster itself fails, the replicas elect a new leader among themselves. The new elected leader communicates with all the Borglets and restores the state of the system (other than the Borgmaster). As stated earlier, the Borgmaster periodically takes a snapshot of its state and stores it in a stable storage. The new leader can make use of this checkpoint to restore the state of the Borgmaster (if it has missed some recent updates to the old Borgmaster).

Advantages: Almost all kinds of crash failures get handled as the chances of all the Borgmaster replicas crashing simultaneously are very low. Periodic checkpointing minimizes the recovery time. One major advantage is that fault in one part of the system doesn't affect the tasks in the other parts and they can carry on with their execution. Hence, distributing tasks (of the same job) across machines makes the job fault tolerant to a great extent in this model.

Disadvantages: A lot of computation power and time is wasted for updating the Borgmaster replicas. Some of the tasks in the failed machines have to be restarted. A lot of space is wasted in storing the checkpoints and most of the time they are of no use as the chances of a system failure are very rare.

Decision Rationale: Google can't afford a fault intolerant system as it has millions of users worldwide. For a middleware involving a centralized server, passive replication is one of the more practical solutions to making the system fault tolerant, with respect to crash faults.

6 Conclusions

In this paper, we extensively analyze cluster middleware software systems, examine design challenges and discuss trade-offs involved in various approaches to job submission, job-system matchmaking based on priority, resource quotas and computing node preferences, algorithms for job scheduling based on resource availability, static and dynamic load balancing, job runtime limits, process preemption and migration, avoiding starvation of low priority processes, and fault tolerance in case a computing node or the central server fails. We then present a comparison between various approaches to addressing these challenges and discuss some aspects of their implementations. As case studies, we go on to give a detailed analysis and discussion of design philosophies, architectures, design decisions, implementations, and limitations of two real-life cluster computing systems - Condor and Google Borg. Particularly, we look at their different approaches towards scheduling policies, fairness in scheduling and preemption schemes, and handling node failures and recoveries.

References

[1] Brett Bode, David M Halstead, Ricky Kendall, Zhou Lei, and David Jackson. The portable batch scheduler and the maui scheduler on linux clusters. In *Annual Linux Showcase & Conference*, 2000.

[2] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):10, 2016.

[3] Dick HJ Epema, Miron Livny, René van Dantzig, Xander Evers, and Jim Pruyne. A worldwide flock of condors: Load sharing among workstation clusters. *Future Generation Computer Systems*, 12(1):53–65, 1996.

[4] Ahmed K Ezzat. Load balancing in nest: A network of workstations. In *Proceedings of 1986 ACM Fall joint computer conference*, pages 1138–1149. IEEE Computer Society Press, 1986.

[5] Dror G Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer, 1997.

[6] Robert L Henderson. Job scheduling under the portable batch system. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294. Springer, 1995.

[7] Joseph A Kaplan and Michael L Nelson. A comparison of queueing, cluster and distributed computing systems. 1993.

[8] Michael J Litzkow, Miron Livny, and Matt W Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111. IEEE, 1988.

[9] Najme Mansouri, Gholam Hosein Dastghaibyfard, and Ehsan Mansouri. Combination of data replication and scheduling algorithm for improving data availability in data grids. *Journal of Network and Computer Applications*, 36(2):711–722, 2013.

[10] Mina Sedaghat, Eddie Wadbro, John Wilkes, Sara De Luna, Oleg Seleznjev, and Erik Elmroth. Diehard: reliable scheduling to survive correlated failures in cloud data centers. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 52–59. IEEE, 2016.

[11] Joseph Skovira, Waiman Chan, Honbo Zhou, and David Lifka. The easy-loadleveler api project. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 41–47. Springer, 1996.

[12] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor: a distributed job scheduler. In *Beowulf cluster computing with Linux*, pages 307–350. MIT press, 2001.

[13] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.