Distributed Systems: A Modern Approach

Prashant Shenoy

University of Massachusetts Amherst

5. Cluster Computing and Distributed Scheduling

Scheduling is a fundamental problem in operating and distributed systems and one that has been studied since the early days of computing. Processor scheduling has first studied in the context of time-sharing in mainframe computers in the 1960s. Early processor scheduling techniques were designed to schedule a set of jobs on a uniprocessor system using. As mainframe and minicomputer systems gained multiprocessing capabilities, numerous multiprocessor scheduling algorithms were designed. Some of these techniques targeted parallel computing on multi-processors using scheduling algorithms such as gang scheduling.

The continued advances in hardware technologies resulting from Moore's Law led to the era of workstations. It became possible to give each user their own workstation computer on their desk and users no longer needed to share a single backend mainframe or minicomputing system. Users could run jobs locally on their machines. Workstations were interconnected over a local area network using technologies such as token ring and ethernet. This interconnected network of workstations opened up new opportunities in scheduling and led the evolution of the field of distributed scheduling. A job could be scheduled onto any workstation in the network and was not limited to executing on the local workstation where it was initially submitted. Systems such as Sprite Sprite, System V, and Condor designed new techniques for distributed scheduling in such environments.

Cluster computing became popular in the 1990s with the emergence of server clusters that consisted of a set of servers interconnected by a high-speed local area network switch. Users could submit batch jobs to the cluster and the cluster scheduler would schedule jobs by assigning them to a server within the cluster. Cluster computing led to the design of numerous scheduling techniques for executing batch jobs on a cluster of servers.

As virtualization techniques emerged in the late 1990s and early 2000, the clusters become virtualized. This led to the design of cluster managers that used scheduling techniques to schedule virtual machines and containers onto a virtualized cluster.

This structure of this chapter follows the evolution of distributed scheduling and cluster computing. We first discuss distributed scheduling in a network of workstations. Next, we discuss cluster scheduling techniques to schedule batch jobs on a cluster of servers. Finally, we discuss cluster resource management techniques for scheduling and managing resources in virtualized clusters that run containers or virtual machines.

Distributed Scheduling in a Network of Workstations

Workstation-based desktop computing became popular in the 1980s with the emergence of a number of commercial workstation products from Digital, Sun Microsystems, HP, SGI, and IBM. With a workstation on every user's desk, it became feasible for users to run their applications and jobs locally on their workstations. However, researchers found that many workstations remained idle over the course of the day. Studies showed that around 30% of the workstations were idle at any time during the day and a higher fraction were idle

during evenings and nights. Harnessing idle workstations therefore became a popular research topic in the 1980s and 1990s.

The scheduling problem in a network of workstation environment was as follows. Given a newly arriving process at a workstation, should the process be executed locally or on a remote workstation? The answer would depend on the type of application and the utilization of the local and remote workstations. In case of interactive applications, running the application locally was desirable to ensure good response time and to avoid the network latency of interacting with a remote workstation. In case of batch applications, the system should strive to minimize the job completion time. In this case, if the utilization of the local workstation is low, the job should be run locally. If the utilization is high and a remote workstation with low load is available, it is better to execute the job remotely since it will result in a lower completion time.

The decision of whether to execute a job locally or remotely can be made manually by the user submitting the job or by a distributed scheduler. In the former case, a user can decide to execute a job remotely using the ssh command (the rsh command in UNIX operating systems was previously used for this purpose).

ssh hostname command

To run a batch simulation on a remote machine "alice-workstation," user bob would need to issue the command ssh alice-workstation job.sh

Manual scheduling of jobs is cumbersome since users need to keep track of idle workstations. Further if the user of the remote workstation began using their machine, the workstation would no longer be idle and the job would need to be moved to another machine or the user would need to allow the job to continue and accept lower performance. Distributed scheduling algorithms overcome this drawback by making the decisions on behalf of all users and their jobs. At a high level, a distributed scheduler that is designed to harvest idle cycles on workstations have four components that are part of its decision-making.

(i) Transfer policy, which determine {*when*} *to transfer a process to a remote workstation. A common type of transfer policy is to use a threshold on the local workstation load to determine whether to transfer a process and execute it remotely. The measure of load were metrics such as CPU utilization or the job queue length. Such a policy is purely local since it can make this decision using local information. An alternate policy is to use global information of the load on all workstations to determine if the local workstation is too loaded relative to other workstations.*

(*ii*) *Selection policy, which determines which process to transfer. It is simple to transfer a new job that has yet to begin execution since that job can started on a remote host* (*similar to the use of the above ssh command*). *Some distributed schedulers could also transfer a currently executing process to a remote host. This can be done by terminating the process and restarting it on a remote machine. This is straightforward to implement but wastes CPU cycles spent on the job so far. Alternatively the process could be migrated to a new host by suspending its execution, transferring its state to the new host and resuming execution on that host. Many research efforts have focused on designing OS-level process migration techniques but transparent process migration remains challenging to this day* [*SEE process migration box*] *The modern approach is to use virtual machine migration, which is discussed in sec X, to migration an application and its state, since VM migration overcomes most of the drawbacks of OS-level process migration.*

(*iii*) *information policy, which keeps track of the state of various workstations in the system. Specif-*

ically, the policy keeps information about which machines ones are idle or busy as well as the load on each workstation.

The information policy can be centralized or decentralized. In the centralized approach, a central manager node keep track of state of various node. This can be done by periodically polling the nodes or having a monitoring agent on each node send updates to the central node.

In the decentralized approach, there is no central node and each workstation is responsible to keeping track of the state of various nodes in the system. Decentralized information updates can be push or poll-based. In the push based method, each machine can periodically broadcast its state to all nodes, indicating whether it is busy or idle and its current load. An alternate approach is to broadcast updates only when there is a state-change, which means a workstation only broadcasts an update when there is a change in state(e.g., it becomes idle after a busy period, or vice versa).

Poll-based methods involve sending queries to various workstations to determine their state. Upon receiving a query, the workstation can report its current state (e.g., busy, idle etc) to the querying workstation. Polling can be done periodically to keep an up-to-date state table about the system state or only when the machine needs to transfer a remote job.

(iv) Location policy, which determines where to transfer the process. Since the goal of the distributed scheduler is to harness idle workstations, the location policy uses the information policy to find an idle workstation to transfer and execute the selected process.

A workstation can be deemed idle if it has no user logged in or if there is a user logged in but there has been no keyboard or mouse activity for a period of time and no foreground application is running. The main principle is to use idle workstations for remote jobs so long as they are not being used by their respective users. If an idle workstation is chosen for executing a remote job and the user begins using the machine again (which means it is no longer idle), then the workstation is immediately returned to the user by suspending the remote job and transferring it to another machine. This ensures that users are not inconvenienced due to the execution of remote jobs and those jobs only run when the user is not actively using their machine.

The location policy can simply use the information policy, discussed above, to determine a list of idle workstations and then choosing a particular machine from the list to run the remote job. Once an idle workstation has been selected, it can be queried to ensure that it is still idle before sending it a remote job. This can avoid a situation where many workstations choose the same idle workstation to send their remote jobs causing it to become overloaded.

These general concepts for a distributed scheduling policy can be translated into several types of policies, depending on whether the sender, receiver, or both initiate the request for a remote job. A sender is a workstation that has a job to transfer to another workstation. A receiver is a workstation that is willing to receive a remote job since it is idle.

Sender-initiated Policy: In the sender-initiated policy, the sending workstation initiates a transfer of one or more jobs whenever its local load increases beyond a threshold. In this case, the transfer policy is a threshold on the CPU utilization or the number of queued up jobs at the sending workstation. When a new job arrives, the scheduler uses this threshold to determine whether it should execute the job locally or remotely.
If remote execution is desired, the selection policy needs to select a job for remote execution. While any job can be transferred, it is straightforward to transfer the newly arriving

job since it is yet to start execution and can be started as a new process on the remote process. An existing process can be chosen only if the OS supports process migration, which we discuss below in the context of Sprite operating system. In the absence of process migration support, only newly arriving jobs can be transferred.

The location policy needs to find an idle workstation to execute the selected process. In a decentralized approach, the sender queries other workstations in the network to find an idle one. Queries can be made sequentially or in parallel to K workstations at a time. If multiple idle workstations are found, one can be chosen at random. In case of a centralized approach, the cluster manager keeps track of the state of all workstations. The sender queries the cluster manager for a list of idle workstations. Any workstation can be chosen from this list, but the sender should first query that workstation to ensure that it is still idle (in case the manager has stale information that is yet to be updated). If the workstation has become busy since reporting itself as idle to the cluster manager, another idle workstation should be queried until an idle workstation is found.

Once a job is transferred to a remote workstation, it runs there until completion. However, if the user returns to their workstation and starts using it again, the workstation should be immediately returned to the user. In that case, all transferred processes are suspended and become candidates for transfer to another machine via process migration. If process migration is not supported, these suspended processes are terminated and then restarted (as new processes) on another idle machine using the transfer policy.

Receiver-initiated Policy. A receiver-initiated policy is opposite of its sender-initiated counterpart, where idle workstations become receivers and proactively look for jobs to run. In a receiver-initiated policy, the transfer policy can either be low threshold on the load such as CPU utilization or any other metric that is used to determine if the workstation is idle (e.g., no user logged in or no mouse/keyboard activity for a period of time and no active foreground jobs). Once a workstation becomes are receiver as per the transfer policy, it needs to look for and overloaded workstation and request a job to execute. The location policy is used to locate an overloaded workstation. Like before, a decentralized policy can be used where the receiver polls other machine to see if they are suitable candidates for transferring jobs. In a centralized policy, the receiver polls the cluster manager to request a list of overloaded workstation and selects one of them for transferring a job. The transfer policy can then select a new job queued up at the overloaded workstation and transfer it to the receiver for remote execution. If process migration is supported, an existing process can be transferred. A previously transferred process that was suspended due to the user's return to the workstation can also be transferred (either via process migration or as a new process after termination of the suspended process).

Symmetric Policy. A symmetric policy combines the sender- and receiver-initiated policy. In this case, all workstations with load above a high threshold become senders, while all workstations with load below a low threshold become receivers. Workstations with a load between the high and low threshold are neither senders not receivers. A sender actively looks for a receiver to offload one or more of its jobs for remote execution, while receivers active look for a sender to take on a remote job. To locate a sender or a receiver, the information policy is used to poll other workstation or the cluster manager, in case of a centralized approach, to generate a list of senders and receivers. A sender can choose a random workstation from the list of receivers, while the receiver can choose a random machine from the list of receivers as its location policy.

Having discussed the basics of distributed scheduling in a network of workstations, we now discuss three prominent systems that were were among the first to introduce these concepts into a network of workstations.

System V

System V was a distributed operating system developed at Stanford University in the 1980s. It introduced the notion of premptible remote execution where a user could specify remote execution of a job (e.g., using a command such as @). The use of a wildcard "*" for the machine name indicated that the system could choose a suitable idle workstation for remote execution. A sender-initiated approach is used where the workstation sends a query to all workstations to see if they have sufficient resources to execute a remote job. While many workstations may respond, the one responding first is chosen for remote execution of the job. While this approach works well for small networks, the overhead of broadcasting queries can be reduced using a state change information policy where machines broadcast a change of state when they become a receiver node. Each sender maintains a list of the least loaded receivers and can only query this group rather than all workstations to determine a suitable receiver.

System V was an early system that introduced the notion of process migration. A program could be migrated by invoking the "migrateprog" command, which would migrate the process and all child processes that it created during its execution. System V explored two different process migration methods. The first method involved freezing the process (and sub-processes), copying the memory state of the process to a new machine, and then unfreezing the process and deleting the old copy. In this case, a receiver node needs to be located like before that is willing to accept the migrated program and its process. However, since freezing the process during its migration causes downtime, System V introduced the idea of pre-copying state. In this case, the bulk of the process state was transferred to the new host before the process was frozen and the remaining state was transferred before resuming execution. Since the executing process can continue to modify its memory pages, the pre-copy process involved iterative transfer of pages, where each round sends pages that were modified in the prior round until a small number of dirty pages remain. At that point the process is frozen and its remaining dirty pages are sent. This notion of pre-copy has now been widely adopted in live migration of virtual machines (see Sec X) and was first introduced by Theimer for live process migration in System V.

The main challenges of migrating a process to a new host are as follows: 1) If the process is accessing hardware devices on the host, it is challenging to migrate the process to a new host. System V avoided transferring processes that accessed hardware resources. Later systems have explored providing remote access to such resources from the new host. For example, access to a disk file could be provided using remote file access or a networked file system. 2) If the process uses network communication, its network socket connections are bound to a IP address that is specific to the host. Migrating the process will cause these network connections to fail since the new host will have a different IP address. System V and others used network packet redirection to redirect messages from the original host to the receiver host running the migrated process. This leaves a dependency on the original host where the network connections were created.

These issues remain a challenge for process migration to this day. Virtual machine migration which we study in Sec XX addresses these issues through virtualizing resources and using a VM-specific IP address that can move with the VM and is separate from the host IP address.

Sprite

The Sprite Operating system was a research operating system developed at UC Berkeley. It had a number of features to utilize idle workstation. A key design principle in Sprite was the the onwer of a workstation was to be respected above all. If a workstation was idle for more than 30 seconds due to lack of keyboard and moused activity and the number of active processes was less than the number of processors, the workstation became a receiver. A centralized coordinator kept track of receiver workstation through a state change-based policy, where the workstation notified the coordinator if it became a receiver. A user could submit a job for remote execution and the system could also choose to migrate a currently executing process to a new node. If the user returned to their workstation, the machine was returned to the user by suspending remotely executing processing and migrating them to a new receiver workstation.

Sprite introduced the notion transparent process migration, where executing a process had the same behavior whether the process executed locally or remotely. The execution environment of the process (files, working directory, device access) remained identical even with process migration. This was achieved using several mechanisms. First, Sprite used a distributed file system that was uniform across all machines, so file access and directories were unchanged (had the same namespace) across machines. Second, each process had a home machine where it originated. When the process invokes a kernel call, it would normally execute on the current workstation–unless its output is machine dependent, in which case it was forwarded to the home machines for execution. The ensures that the results of execution are identical regardless of where the process executes and also takes care of hardware dependencies due to its ability to execute kernel calls on the home machine.

Unlike system V that uses a pre-copy method to migrate the process, Sprits uses a virtual memory-based suspend-resume approach. To migrate a process, the process is suspended and its memory image (and dirty pages) are written to a disk file (i.e., disk swap space). The receiver node can resume execution of the process by loading its pages from disk—using standard virtual memory mechanisms to load a process's pages on-demand as it resumes execution. While this process is slower in time due to the use of disk, it involves less data copying than the pre-copy method which needs to iteratively (and repeatedly) copy dirtied pages to the destination workstation.

Condor.

Condor was a distributed scheduling system developed at the Univ of Wisconsin. Unlike System V and Sprite which were new operating systems, Condor was designed to run on existing UNIX operating systems. While System V and Sprite had many design goals in addition to distributed scheduling, Condor was explicitly designed to utilize idle cycles in a network of workstations.

It used a centralized coordinator that polled each workstation every two minutes to see if the workstation had idle cycles to service remote jobs (i.e., was a receiver) and which ones were senders with queued up jobs awaiting remote execution. The central coordinator than makes capacity on receiver nodes available to the local schedulers on senders, which trigger the execution of job on those remote machines. If the local user resumes using their workstation, the local scheduler preempts execution of all remote jobs. Jobs are checkpointed periodically to disk so that they can be resumed on another node.

Condor was also one of the first systems to distinguish between different types of users

and their jobs. In particular, it introduced the notion of heavy and light users and short and long jobs. It allowed heavy users to use remote capacity while allowing fair access to light users. As we will see next, this notion of differential treatment of different classes of users and different types of jobs is used in all modern batch schedulers. The Condor project was quite successful and has ported to many different operating systems over the years. It continues to be active project at U. Wisconsin even today and can be used in variety of distributed computed settings to run batch jobs on remote machines (see HTcondor page).

Volunteer Computing

The notion of using idle cycles from a network of workstations on a local area network has been extended to the Internet using a volunteer model. Volunteer computing allows any user with a PC connected to the Internet to donate unused cycles on their PC for any large-scale computational task. The motivation behind volunteer computing is similar to that for using idle workstation cycles in Condor, Sprite, and similar systems. A typical PC is used for less than 30% of the time and the unused cycles of a typical PC can be harnessed over the Internet to run large computational or data processing tasks. In the volunteer computing model, a user who wishes to donate their PC's unused cycles needs to download and install volunteer computing application. In the simplest case, this software runs as screen saver application on the machine. When ever the screen saver activates, the desktop is assumed to be idle and the application contacts a coordinator server over the Internet for some work. The coordinator has sends a small task to the desktop and waits for the result. Once the task completes, the next task is sent and so on. If the user returns to the computer, the screen saver application deactivates and the partially completed task is killed.

Some of the early successes with Volunteer Computing involved projects such as SETI@Home and Folding@Home that involved processing a large amount of data to detect signs of extra terrestrial intelligence and to analyze protein folding for discovering drugs. Both projects are embarrassingly parallel, which means they can run on as many parallel machines as are available, and also amenable to be partitioned into large number of small tasks (e.g., processing a small piece of data from a large dataset), each of which can be sent to a volunteer computer to perform a short computation. There are two benefits to this approach. Since each task is relatively short, terminating it if a user returns to their desktop does not waste too many cycles. Further, since a volunteer computer is inherently untrusted and can send back erroneous or bogus results, each task can be sent to multiple independent machines and the results can be compared and accepted only if the majority of them match. This is easier to do if the application can be split into many small tasks that can independently processed and replicated.

The Berkeley Open Infrastructure of Network Computing (BOINC) project was instrumental in popularizing the volunteer computing approach over the Internet in the early 2000s [Anderson paper]. It was a cross-platform middleware that could harness compute cycles from Windows, Macs and Linux computers, and has since been extended to tablets and mobile phones. In its heyday, BOINC hosted a large number of research projects, including SETI@Home, and these projects could harness idle cycles from tens of thousands of computers over the Internet. The BOINC project is still active and is being used for research on climate change, discovering pulsars and diseases.

Scheduling Batch Jobs in a Cluster of servers

Despite the research on harnessing idle cycles from computing devices, a hybrid model emerged where desktops are used by their users to run foreground applications and local jobs and all remote jobs are run on dedicated pool of server machines (i.e., a server cluster) designed to run batch-orientation computational and data processing jobs. The notion of using idle cycles on desktop computers is now used in limited scenarios such as volunteer computing. This model, also known as the processor pool model, assumes that it is acceptable to waste unused cycles on desktops and use a dedicated cluster of (powerful) servers to run remote jobs. In this model, user run interactive and smaller batch jobs locally on their desktops. To run compute-intensive jobs, the user logs into the primary ("head") node of dedicated server cluster and submits the job to the cluster. Newly submitted jobs are queued up and a batch scheduler assigns these jobs to a node (e.g., underutlitized or idle node) that has sufficient resources to execute the task. This model is common in enterprises, where a cluster of powerful servers is dedicated for executing high-performance batch-oriented compute jobs. Batch workloads ranging from scientific simulations, large-scale data ("big-data") processing, and machine learning training are well-suited for this processor pool model where a cluster batch scheduler is responsible for assigning oncoming jobs to specific nodes in the cluster.

Batch jobs come in many flavors: * single batch job, which is a non-interactive job that executes as a unit. The job may consist of sub-tasks that execute sequentially. * parallel batch job, where the job consists of sub-tasks that that execute in parallel (either on different different processors of a server or on different servers) * distributed batch job with dependencies across sub-tasks, where the job can be viewed as a directed acyclic graph with each node representing a sub-task. The graph captures dependencies between tasks such that sub-task can execute only after all its parent nodes have finished execution. Regardless of the type, all batch jobs have similar characteristics: they are non-interactive in nature and and require the system to optimize their completion times or throughput. *** add more details ***

Over the past two decades, numerous batch schedulers have been developed for distributed scheduling of batch jobs on to nodes of a cluster. Early batch schedulers were developed for mainframe computers or large symmetric multiprocessor (SMP) servers, where a single SMP server (or mainframe) had a large number of processors, with each processor capable or executing a task. The batch scheduler was responsible for assigning jobs to processors on the server. As commodity x86 servers become popular, it was more cost effective to deploy a cluster of commodity servers over purchasing a single SMP server. In this case, batch schedulers were responsible for assigning jobs to nodes of the cluster (or to one or more processors on a specific node).

These include batch scheduling frameworks such as IBM Load Leveler (designed for parallel SMP servers), IBM Load Sharing Facility (which adapted batch scheduling to server clusters), and Maui (which was originally designed for SMPs and later adapted to HPC clusters). Newer batch scheduling framework such as Moab HPC cluster scheduler, Distributed Queuing System (DQS), Sun Grid Engine, and Simple Linux Utility for Resource Management (SLURM) all focus on server clusters and especially heterogenous environments.

Let us first consider a simple FCFS batch scheduler, which is conceptually how all batch schedulers work. Consider a cluster of N homogeneous servers that can execute compute-intensive or data-intensive batch jobs. The scheduler runs on a primary node that manages the remaining pool of N-1 servers. The scheduler maintains a job queue where users sub-

mit their jobs and jobs are queued in FIFO order. The scheduler always assigns the jobs at the head of the queue to an idle server, if one is available, where it runs to completion on that node. If all servers are busy, the job must wait. When a server completes execution of an existing job and becomes idle, the scheduler schedules the next job at the head of the queue onto the server.

Modern batch schedulers have many additional features beyond simple FCFS scheduling. We discuss three key aspects of such schedulers: cluster heterogeneity, different job types, and different user groups.

Our simple FCFS scheduler assumed that all servers in the cluster have identical hardware configurations and it was feasible to schedule a waiting job on any server. In practice, this homogeneous assumption may not hold. Cluster may receive upgrades from time to time, where new servers with newer processors or more more resources such as memory or disk are added to the cluster. In this case, the cluster will have groups of servers of varying vintage and different hardware configurations with varying number of cores per server, different processor speeds, and different amounts of memory. Alternatively, the cluster may have mixed configurations, where some servers are equipped with special hardware such as expensive GPUs, while rest lack such features.

When the cluster has heterogeneous or mixed hardware configurations, the batch scheduler can no longer perform simple FCFS scheduling and will instead need to consider both the job requirements and server configurations when scheduling jobs. When a user submits a job to the cluster, the job will need to specify its resource requirement (e.g., minimum memory requirement) as well as specific hardware needs such as whether the job needs to run on a GPU-equipped server. These specifications enable the scheduler to determine whether a specific servers from the cluster has the correct type of hardware and software resources to execute the job. When a node becomes idle, the scheduler scan the queue of waiting job from the head to the tail and schedules the first matching job on that server. A matching job is one where its requested resources can be satisfied by the idle server.

Further, if each server have multiple cores or processors, it is capable of running more than one job at a time. The scheduler needs to consider the amount of unusued memory and cores on each server to match a job that will fit on the server and execute concurrently with other jobs.

Job types: The simple FCFS policy does not distinguish between job types and treats all jobs uniformly. Many studies have shown that the majority of the batch jobs are short jobs (finish in a few minutes), while a small fraction of jobs are long jobs. Long jobs take up majority of the compute cycles in the cluster. In an FCFS policy, long jobs that are executing on the cluster can increase the waiting jobs for short jobs—since short jobs may have to wait for a while until a long job finishes. To avoid such problems, the scheduler should treat long and short jobs differently. This can be done by maintaining two queues, one for long and one for short jobs. A user should submit short jobs to the short job queue and long jobs to the long job queues. To avoid starvation or high waiting times, the cluster can be partitioned across job types. For example, short jobs will execute on the server pool reserved for those type of jobs.

Each queue places a maximum limit on the amount of time a job can execute after which it is killed. Thus if a user erroneously submits a long job to a short job queue, it will be terminated when the maximum limit on short job execution is reached, which incentives the user to choose the proper queue for running their jobs.

Fair share / user partitions - multiple queues

A cluster of server is typically shared by many users who submit their jobs to the cluster. If the cluster employs FCFS scheduling to execute jobs, then a user's job can be starved if some other users have submitted a large number of jobs that are all waiting for execution in the queue. In shared academic clusters, it is not uncommon for a job to wait for hours or days to be executed if queue has a large number of jobs submitted by other users. This is a common problem with shared queues where heavy users of the cluster can starve other users.

One approach for addressing this issue is to allocate a certain share to each user, or an organizational unit such as a department or a research group that consists of a group of users. The cluster capacity is then shared between users in a fair share manner. In this each, each user, or group, is assigned their own queue. Jobs for various queues are scheduled onto the cluster is a fair share manner. For example, each queue is assigned a weight, which determined a guaranteed fraction of the cluster's compute cycles. When all queues have waiting jobs, each queue is allocated cluster capacity in proportion to its weight. The fair share job scheduler is work conserving in nature, which means that if a user's queue is empty, it's unused share is allocated to other queues with waiting jobs in proportion to their weights. If one or more jobs are submitted to this queue, the guaranteed share is returned to the user's queue. This can be done by preempting jobs from queues that are temporarily using more than their fair share (e.g., by using unused shares from empty queues). A job can be preempted by either by suspending it, which enables it to resume elsewhere at a later time, or by terminating and inserting it back its queue to restart it.

A fair share scheduler can also be used to schedule long and short jobs. This avoids having to statically partitions the servers into two groups and scheduling short and lob jobs in two different static partitions. For example, rather than statically partitioning the cluster as discussed above, the fair scheduler can be configured to give 60% of the cluster capacity to short jobs and 40% to long jobs (by assigning weights of 3:2 to the short and long job queue).

A fair share scheduler is more efficient that statically partitioning the cluster and using a FCFS scheduler within each partition. This is because a fair share scheduler allows each job type to use unused capacity for the other when available, while static partitions waste capacity within the partition if the corresponding job queue is empty. A fair scheduler also ensures that any unused capacity that is assigned to other jobs types or other user groups is immediately returned to the empty queue as soon as it becomes non-empty. This is done by preempting some jobs either by suspending or terminating them. The work conserving property of fair share schedulers ensures higher utilization of resources and more efficient use of servers. See Ch XX for a discussion of fair share schedulers.

In addition to using fair share scheduling to allocate a share of the cluster to each user group or job class, batch schedulers also support the concept of resource reservations. A resource reservation allows a certain number of servers in the cluster to be reserved or dedicated for a certain period of time. For example, if a research group needs to run a certain type of experiment, they can request 32 machines, each with at least 16GB of memory for 6 hours, starting at 5pm on a certain date. A resource reservation effectively creates a logical cluster of servers that is dedicated for that task for the specified time duration and blocks other users or jobs from using the reserved machines during the reserved time period. Since reserved machines may not be fully utilized by their users, the scheduler can employ a technique called backfilling to increase utilization and avoid wasting idle cycles, similar to how fair schedulers can reassign unused capacity to queues with pend-

ing jobs. Backfilling involves using unused server slots from reserved machines and using these slots to run short jobs from other users of the cluster, thereby filling "holes" in the schedule. Short jobs are preferred for backfilling since job submitted by used of reserved machines can simply wait for those jobs to finish to reclaim servers and not have to preempt the job.

Beyond FCFS scheduling: priorities and gang scheduling

Modern batch scheduler support a number of other scheduling policies beyond FCFS and fair share schedulers. The use of priority scheduling, which supports priorities for jobs, is desirable in many scenarios. In this approach, users can assign high priority to jobs with deadlines or completion time requirements. Jobs that are capable of running in the background whenever idle resources are available are assigned low priority. The scheduler uses a priority queue to queue various low and high priority jobs. High priority jobs are always scheduled first in FCFS order. Whenever the high priority job queue is empty, low priority jobs are executed on the cluster. If a high priority job arrives and no idle servers are available in the cluster, the scheduler preempts a low priority job and schedules the high priority job on that server. Priority scheduling is often used to running elastic tasks with loose completion time whenever possible to increase cluster utilization, while still being able to run high priority jobs immediately and ensure low waiting times for these jobs.

Gang scheduling is a scheduling policy that is specifically designed for running parallel jobs. Gang scheduling was originally designed for running parallel applications on multiprocessor servers. Parallel applications are commonly implemented using a middleware called Message Passing Interface (MPI) that enables efficient inter-process communication using messages (see Ch XXX for a detailed discussion of MPI). A parallel MPI application consists of multiple communicating processes. When these processes are scheduled independently by the CPU scheduler on various processors or cores, communication via message passing can cause blocking. For example, if a process that is running on a processor sends a message to different process that is currently not scheduled, the former process will need to wait until the latter process is scheduled on a processor and generates a reply after processing the sent message. The waiting problem is exacerbated when messages are broadcast to all other processes of the parallel application and each of then run asynchronously and processes the message at different times. Gang scheduling avoids such problems by scheduling all processes of a parallel application in the same time slot on different processors of a SMP server. The concept can be extended to a cluster where components of the parallel application, referred to as a gang of processes, are scheduled in the same time slot on various servers. Synchronizing the scheduling of the parallel application reduces blocking time resulting from message passing, since sent messages can be processed immediately by the recipients and response, if any, can be sent back. Unlike FCFS batch schedulers that schedule each job independently of other jobs in the system, gang scheduling requires coordination where a group of processes or jobs are always scheduled together on the cluster.

Batch schedulers offer numerous configuration options that allow various policies to be enforced on the same cluster. As an example, consider a server cluster at an university that is shared by three departments: Math, Computer Science, and Engineering. The scheduler can implement a separate queue for each department with fair sharing implemented across queues. Each department's share can be configured by assigning a weight. A weight of

1:1:2 indicates that Math and CS are a share of 25% each, while Engineering is given a share of 50%. Further, each department's share can be split between long and short jobs using a separate queue for each type of job and allocating a share to each type of queue. For example, 40% of each department's capacity can be allocated to short jobs and 60% to long jobs. Resource reservations can be used to set aside a certain number of servers for a specialized task such as running a large parallel application without any interference from other executing jobs.

SLURM

SLURM (Simple Linux Utility for Resource Management) is a popular workload scheduler for Linux blusters designed to run a batch and parallel jobs. As of 2019, it was the scheduler of choice on more than 60% of the top supercomputing clusters worldwide and is also common in smaller clusters in academic and enterprise environments. Like all batch schedulers, SLURM is designed to schedule a queue of pending jobs. It can provide jobs with shared or exclusive (reserved) access to nodes in the cluster and also provide support for running parallel MPI jobs.

SLURM consists of centralized control daemon to monitor nodes and jobs and to schedule jobs onto nodes. Each server node runs a local daemon that receives jobs to execute from the central daemon. SLURM also provides a REST interface that can be used by users to programmatically interact with the scheduler. SLURM allows nodes to be grouped into partitions, each with a job queue. A job queue can be configured to dictate what types of jobs and users can use that partition. This is done by specifying size and time limits for jobs and user group that are allowed to submit jobs to the queue. Jobs can have priorities and are scheduled onto nodes in priority order. Each job consists of a set of job steps, which are sub-tasks of the overall job. A sub-task can use all nodes allocated to the job (for a parallel task where sub-tasks depend on each other and run in parallel) or several sub-tasks can run concurrently on various nodes (for a task that consists of multiple independent sub-tasks).

SLURM supports variety of scheduling policies for queued jobs. Job scheduling can be configured using two parameters: priorityType and schedType. The priorityType parameter can be set to basic, which defaults to FIFO scheduling or multi-factor, which assigns priority to jobs based on various job characteristics such as age, job size, fair share. Given these job priorities, the schedType parameter can be used to choose a job using FCFS, backfill and gang scheduling. See [paper] for more details on SLURM.

Distributed Scheduling in CLusters with Multiple Applications Classes

Cluster schedulers such as SLURM are designed to schedule a single class of applications, namely batch jobs. While batch jobs represent a rich applications class for cluster computing, modern clusters also run other application classes. Interactive services with low response time requirements are one such application class. Example of interactive services include web applications that we encountered in Ch XXX, key value stores (Ch XXX) and database transaction processing. Distributed data processing applications such as Hadoop and SPARK are data-intesitive in nature and represent a different class of applications that exhibit characteristics of both batch and interactive jobs (Ch XX). Scheduling cluster resources across multiple application classes is more challenging than designing a cluster

scheduler for a single application class.

One common approach is to statistically partition the cluster into multiple groups, one for each application class, and to use a separate scheduler within each group of servers. However, static partitioning of servers in a cluster can be inefficient in the presence of dynamically changing workload demand within each application class. This can cause some partitions to be highly utilized, while other partitions have idle servers. Coarse timescale repartitioning of the cluster may be needed to handle changing demand within each application class.

To address this issue, recent cluster schedulers have been designed to handle multiple application classes by enabling fine-grain share of cluster servers across application classes. Fine-grain sharing allows the scheduler to flexibly vary the number of servers allocation to each application class depending on the number of tasks (i.e., workload demand) within each class. Such schedulers employ a hierarchical two-level approach as shown in Figure XXX. Rather than using a single scheduler to directly allocate server resources to tasks from various application classes, such schedulers allocate a certain number of servers to each application class. Each application class employs a second-level class-specific scheduler that then allocates servers from its allocation to individual tasks. Such a two-level approach enables each class to employ a scheduling policy that is appropriate for its tasks, rather than being constrained by a single global policy. The top-level scheduler is responsible to determine the aggregate allocation to each class at a fine time granularity and leaves the decision of how to allocate those servers to individual tasks within a class to the class-specific schedulers. Next we review four cluster schedulers

Mesos

Mesos is a cluster manager and scheduler designed for fine-grain sharing of servers between multiple cluster frameworks such as Hadoop and MPI. Mesos recognizes that no single cluster computing framework will be suitable for all applications. For example, a framework such as SLURM that is designed for batch jobs will not work well for MapReduce jobs. Hence an organization ends up with multiple cluster frameworks, one for each class of applications, by statically partitioning the cluster or using group of virtual machines. As noted above, this approach does not allow fine-grain sharing of cluster resources across framework.

Mesos uses a two-level approach where it allocates resources to frameworks and delegates control of scheduling these resources to those frameworks. Mesos introduces a new abstraction for allocating resources: resource offers, which is a bundle of resources from the cluster offered to a framework. Mesos decided how many resources to offer to a framework, based on organization-wide policies such as fair sharing, and offers these resources to the framework in the form of a resource offer. A framework can decide which offers to accept and which ones to decline. For accepted resource offers, the framework can then decide what tasks to run on them using a framework-specific scheduling policy.

Figure X shows the architecture of Mesos managed cluster. It consists of four key components: the Mesos coordinator, which is a cluster-wide entity that coordinates Mesos workers running on each server. The cluster also consists of onr or more frameworks that run tasks on server nodes. Each framework consists of a scheduler that registers itself with the coordinator for receiving resource offers and an executor that runs on server nodes to run tasks.

Mesos uses a four step process to enable a framework to schedule tasks in a Mesos clus-

ter, which we illustrate using an example. In step 1, when a worker node that becomes idle, it reports that it has 6 cores and 6GB of memory to the coordinator. The coordinator involves its allocation policy, which decides to offer all of these resources to framework 1. In the second step, the coordinator sends a resources offer that lists these resources to the framework. In step 3, the framework decides whether to accept or reject this offer. In this example, it decides to accept the offer and invokes its scheduler, which decides to schedule task 1 using 2 cores and 2 GB of RAM and task 2 using 2 cores and 3GB of RAM. It responds to the coordinator indicating which tasks it has decided to schedule and the list of resources allocated to the tasks using the accepted resource offer. In step 4, the coordinator sends the tasks to the worker node, which allocates those resources to framework's executor and the executor launches the task using the allocated resources. Finally, note that the framework has only accepted a subset of the resources from the resource offer and 2 cores and 1 GB of RAM were not used. These unused resources can be offered using a new resource offer to another framework.

Mesos uses a pluggable resource allocation module that can implement any allocation policy for various framework. Mesos implements two example allocation policies, fair share and priorities. In fair share allocation, the policy allocates a share of the cluster resources to each framework. This is done by making resource offers in line with a framework's share. Allocation modules can also revoke tasks if tasks hand or take inordinately long to complete. Mesos also uses a pluggable isolation module that can use any standard mechanism to provide performance isolation across frameworks and their tasks. For example, tasks can run inside OS containers which are isolated from each other using mechanisms such as cgroup and resource limits. This ensures that multiple executing tasks on the same worker node do not interfere with each other. Mesos ensures robustness using three simple mechanisms. First, it supports the notion of filters, which allow the coordinator to avoid making resource offers that do not match filters specificized by a framework. Doing so reduces the chances of making resource offers that are not useful to a framework and repeatedly rejected. An example filter could specify "only offer worker nodes with at least 4GB of RAM." Second, offered resources count towards a frameworks fair share allocation while they are pending an accept or reject decision, which motivates frameworks to respond quickly to offers. Third, pending offers time out after a threshold duration and are revoked if the framework does not respond with the maximum threshold duration. Finally, to ensure fault tolerance of the coordinator, the coordinator only maintains soft state that can be easily reconstructed if the coordinator fails. Standby coordinators are used, along with leader election, upon a coordinator failure. Frameworks can register multiple schedulers to deal with scheduler failures.

A key decision made in Mesos is that frameworks do not requests resources explicitly from the coordinator. For example, a framework can not request 4 nodes, each with 2 cores and 4GB RAM to run the next two tasks. Instead, the coordinator unilaterally makes resource offers, which the framework can accept or reject. Note that the coordinator makes resource offers without know anything about the current needs of the framework or its list of queued up tasks. The resource offers are made based on decision made by the allocation module using priorities or fair share. A key insight of the Mesos work is that making resource offers to frameworks without any explicit request for resources from the framework works surprisingly well when the workload consist of fine-grain tasks. Thus, Mesos uses a push-based allocation approach over a pull-based approach where framework request resources from the coordinator.

Borg

Borg is a cluster manager designed by Google to manage clusters in their data centers. Borg is designed to run hundreds of thousands of concurrent jobs from tens of thousands of applications that run on clusters, each of which may contain tens of thousands of servers. Many of the lessons learned from Borg were used to design Kubernates, a widely used open-source cluster manager. Hence, it is useful to discuss Borg before delving into Kubernates.

Borg had three design goals. First, it was designed to hide the details of resource management and failure handling from application developers so that developers could focus on designing their application and not worry about how to run the application in a large cluster environnement. Second, it was designed to operate with high reliability since it managed clusters that run most of Google's user-facing services such as google mail, google docs, and web search, which were all themselves designed to be highly available with minimal downtimes. Third, it was designed to scale to cluster sizes of tens of thousands of machines and run application workloads on very large clusters.

Borg provides users with an abstraction of a job to run their applications. Users submit jobs to the system and each job can consist of one or more tasks that all execute the same program. Borg is specifically designed to run two classes of applications that share cluster resources: interactive and batch. The first class of applications are long running applications that have high availability requirements and handle short-lived low-latency requests. This application class consists of Google's user facing services such as google mail, google doc and web search. The second application class are batch job that have durations of seconds to days. Borg assumes the relative proportions of interactive and batch jobs in a cluster will vary over time.
Borg assign priorities to jobs based on their class. Long running interactive jobs, also referred to as production jobs, are given higher priority. Batch jobs, also referred to as non-production jobs, are given lower priority than interactive jobs in Borg.

Borg uses the abstraction of a cell to group machines in a cluster. A typical cell consists of 10,000 machines, all of which belong to the same cluster. Each cluster resides in a data center. While a cluster can host multiple cells, a typical cluster consists of one large cell to run jobs and can consist of a few smaller cells for testing and other special needs. Machines in a cell are assumed to be heterogenous with varying CPU, memory, disk and network capabilities. Borg is designed to match jobs to machines in a cell based on their resource requirements so that developers do not need to worry about determine which machines to choose for their jobs. This is done by specifying resource constraints on jobs (e.g., the job should run on a machine with Intel process and needs at least 4GB RAM). Borg then runs the job on machines with matching attributes.
Figure XX shows a state diagram that shows the lifecycle og a job. A job's configuration can change at run time, and the user can instruct Borg to use a new configuration to run the job. Some configuration updates such as changing the resource requirement of the job (e.g., increasing the memory needed to run the job) may cause the job to need more resources than are available on the current server. In these cases, Borg will stop and restart the job on a different machine.

Borg users resource reservations when running jobs on machine. An reservation (called an alloc) is a set of reserved resources on a single machine, and a group of reservations across machines is called an alloc set. Once this alloc set reservation is created by Borg, jobs can be scheduled on those machines. Scheduling is done using a combination of priorities and quotas. If there are no free resources to run A higher priority job, one or more lower

priority jobs are preempted to free up resources to run the higher priority job. Premempted jobs are rescheduled on other machines when feasible. The concept of a quota is used to cap the maximum amount of resources allocated to a user's job at any time. A quote is a long-terminated reservation of cluster capacity for a job. If a high priority job is using less resources than its quota, it will be guaranteed to run, possibly by prempting low priority job. Once a job reaches its quota, it may need to wait if no free resources are available.

Figure X shows the overall architecture of Borg. Each cell consists of a coordinator called the BorgMaster. The coordinator consists of a process to user requests and a separate scheduler process. For high availability, the coordinator is replicated five times and keeps its state in a highly available distributed Paxos store. One of these processes is elected to be the leader and acquires a chubby lock (see Ch XX for leader election and Chubby locking, Ch XX discusses Paxos). Fail over is handled by electing a new leader.

The scheduler maintains a priority queue of jobs. The queue is scanned from high to low priority jobs and jobs are assigned to machines in the cell if there are resources available to run them. To schedule a task, a list of feasible machines is first computed and a scoring function is used to pick one of those machines. While feasibility is determined based on whether a machine has adequate free resources to run the job, the scoring function chooses a feasible machine based on a goodness criteria. Borg originally used a criteria that spread jobs across machines using a worst-fit criteria. Later on, the criteria was changed to use a hybrid combination of best-fit, which packs jobs onto a small set of machines and worst-fit, which spreads jobs across machines.

Since a single machine can run multiple jobs, Borg provides isolation across tasks using containers and cgroups and also provide security isolation using chroot to prevent a job from accessing data belonging to other tasks. More than 98% of machines in a cell run a mix of production (interactive) and non-production (batch) jobs. Each cell is shared by thousands of users and a cell can new jobs arrival rates of 10,000 jobs per minute. An experimental evaluation of Borg showed that it is more efficient to share a cell across different classes of jobs over an approach that statically partitions the cluster between job classes. The results showed that haring yields higher utilization and lowers hardware costs.

Kubernetes

Kubernetes (Greek word for pilot) is a popular open-source cluster manager for containerized applications. It was originally developed by Google in 2014 and is based on lessons learned from the Borg and Omega cluster managers that were used to manage servers in Google's production data centers.

There are two salient features that distinguish Kubernetes from other cluster managers that we have encountered so far.

While many batch schedulers directly run applications on bare metal servers, Kubernetes is designed primarily for a virtualized cluster environment. Specifically, it is designed for deploying, running and managing containerized applications. Kubernetes uses docker as its default container run-time and deploys dockerized applications onto clusters. Since docker is itself a popular platform for packaging and deploying applications, Kubernetes takes full advantage of the developer's familiarity with docker for application development and deployment. Kubernates also supports other container technologies in addition to docker.

Second, most of the cluster managers and schedulers that we have discussed so far have been designed for batch applications. Others like Mesos, Yarn and Borg support multiple

application classes. In contrast, Kubernetes is designed for interactive cloud services that are based on the micro-services architecture (see Ch XXX). While Kubernetes can also run batch jobs inside containers, many of its features are designed for interactive micro-services.

Like Borg, which inspired its design, Kubernetes is designed to scale to clusters with tens of thousands of servers and is able to manage hundreds of thousands of applications. It is designed to provide the following benefits. Some of these benefits are similar to Borg's design goals outlined in Sec XXX.

It simplifies application deployment by handling deployment of the application onto the cluster on behalf of the developer. The developer can focus on designing their application and can leave the details of what servers to use for the application to Kubernetes. Developers can specify constraints such as the amount of (processor or memory) resources needed to run their application or special needs such as a server with a solid state disk (SSD) or a GPU, and Kubernetes takes care of selecting severs that meet these requirements.

Second, it is designed to achieve high utilization of the cluster's hardware resources. This is achieved by packing applications onto servers so as to achieve high utilization while also meeting each application's resources needs. Since applications reside inside containers, Kubernetes has the ability to move containers from one server to another to achieve better packing and resource utilization.

Third, Kubernetes provide self-healing capabilities where it monitors the health of each deployed application and can handle application failures by restarting the application. The health of nodes is also monitored and node failures are handled by restarting all resident applications on other servers. This is a useful capability for reducing application downtimes since many types of failures can be handled by Kubernetes without any human involvement.

Fourth, Kubernetes has built in capabilities for auto-scaling where it monitors the application workloads and starts new application instances on new servers to handle a workload increase. In addition to horizontal scaling capabilities, Kubernetes also implements load balancing between replicated components of an application.

We note that the many of these benefits are specific to interactive services and are not found in batch schedulers. Since Kubernetes is designed for modern micro-service style interactive application that run in clustered or cloud environments, it has builtin support for these features.

Figure X shows the architecture of Kubernetes. Kubernetes consists of a control node that runs the Kubernetes control plate. The control node manages all of servers in the cluster, which are called worker nodes.

The control plane consists of four key components. - The Kubernates scheduler is reponsible for application placement. It assigns ("schedules'') application components onto worker nodes. - The Controller Manager perform cluster-wide tasks such as tracking the status and health of worker nodes, handling node failures, and replicating application components across nodes. - The API server exposes an REST interface to applications and to other control plane components. It provides a programmatic API to interact with Kubernetes. - etcd is a consistent and highly-available key value store that is used by Kubernetes as its backing store for all cluster data. The etcd key value store can be distributed across multiple control plane nodes.

Each worker nodes runs three Kubernetes components in addition to running application components inside containers. - Each node runs a container runtime to run containers scheduled on that server. Kubernetes uses docker as the default container runtime and is capable of running other container runtimes such as containerd and CRI-O. - Each node

also runs a kublet, which is a component that manages containers on the node using the underlying container run-time.
Finally each node runs a service proxy called kube-proxy that load balances traffic across application components.

Pods: Kubernetes uses the concept of a pod when deploying applications. A pod is a group of containers that are co-located on a node and belong to the same appliaction. Kubernetes uses pods as a building block for application deployments—an application is deployed by deploying one or more pods, each of which contains one of more containers. In the common case, a pod contains a single container. In situations where multiple components of the applications need to be co-located on the same node, the corresponding containers can be grouped into a single pod.

It is useful to understand why Kubernetes uses the pod abstraction rather than dealing directly with containers.

One of the lessons learned from the Borg scheduler was that applications deployed on the same server node can have network port conflicts, which makes application deployment more challenging. For example, consider two web applications that are deployed on the same node. If both applications (or their front end components) are deployed inside containers, the containers will share the IP address and network port addresses of the physical host. Since web services use port 80 as the default HTTP port, this will result in a port conflict since both applications can not use the same port 80 on that host. While Borg addresses this issue by explicitly scheduling ports and avoiding conflicts, Kubernetes has a more elegant solution. It groups one of more containers of the application inside a pod and gives each pod a distinct IP address. Since each pod is a logical host that has its own IP address, and this IP address is different from that of the physical host, applications residing on same host can freely use network ports of their choice without any risk of port conflicts. In the above example, each front-end container can use port 80 for their HTTP server since these ports are bound to two different IP addresses of the corresponding pod.

Fine-grain use of containers: Kubernates encourages applications to use per-process containers. Thus, if an application contains of multiple processes, the preferred approach in Kubernates is to encapsulate each process inside a separate container, rather than grouping all of them into a single container. Kubernetes discourages running multiple unrelated processes inside a container since it complicates replication and self-healing. In the process-per-container approach, if the process crashes, its container can be restarted. If multiple processes are grouped in a container and one of them crashes, it is harder to simply start a new container since the old container still has other running processes from that application.

The pod abstraction enables containers belonging to an application to be grouped together inside a pod. Since containers will typically contain a single process, a pod provides a convenient means to group an application's processes (and their containers) that are running on the node. As an example, consider an application consisting of two processes, one each for its front-end and backend tier. Fig XX(a) shows both processes inside a single container that resides in a pod. However, as explained above, this is discouraged to simplify failure handling and handle crashes. Fig XX(b) shows the front-end and backend tier processes inside separate containers, with both containers in a single pod. While this approach is suitable for application deployment, it is not desirable for applications that desire autoscaling. Auto-scaling requires the ability to scale each tier independently. Since pods are the basic unit of deployment, grouping both tiers inside a single pod will prevent each tier from being scaled independently. In this case, it is better to run each container inside a separate pod, which allows each tier to be scaled independently. A container-per-

pod model also allows the tiers to be mapped onto different machines, which is desirable for clustered multi-tier applications.

Application deployment:

To deploy and run an application in a Kubernetes cluster, the developer needs to first containerize their application and create a container image (e.g., a docker image). This image needs to be uploaded to a container registry that is accessible to the cluster.

The developer then creates an application description for the purpose of deployment. This description is written in a markdown language such as YAML[1] or JSON. The description contains information about the components of the application and the mapping to to pods which determines the components that should be co-located. The description can also many other details such as how many copies of an application to use, constraints on placement, ?? resource requirements <Check?>. See Figure X for an example.

This description is provided to the Kubernetes API server, which invokes the scheduler to determine a placement of components onto the nodes of the cluster. Once a mapping is determined, each node's kubelet can then use the container images to start up pods and containers on that node. If the application description indicates one or more components need to be replicated, the scheduler scheduler the desired number of replicas for each node.

Let us discuss the YAML description for the application as shown in Figure XX. XXX

Scheduling. The Kubernetes scheduler is responsible for placing applications (or more precisely, pods) onto cluster nodes. Like the Borg scheduler, it first determines a list of feasible nodes for each unassigned pod. This is the filtering step that yields nodes with adequate resources to run the pod. The scheduler then performs scoring, which assigns a score to rank these feasible nodes and choose the node with the highest score to run the pod. The scheduler supports scheduling policies that allow for flexibility in how scoring and filtering is done. This is done by specifying predicates that are conditions that must be satisfied for a node to be feasible during the filtering process. Scoring is implemented using different types of priorities that are then used to rank nodes. For example, morerequestedPriority favors nodes with most requested resources and packs pods onto the smallest number of nodes. ServiceSpeadingPriority attempts to spread a pods onto a large number of nodes (which roughly translates to using the least loaded node).

In Kubernetes, pods have priorities and can be preempted. IF a higher priority pod can not be placed into the cluster, a lower priority pod can be evicted to allow that pod to be placed onto the cluster. If a pod is chosen for pre-emption, it is given a grace period to allow for it to be properly shutdown. The pod is terminated at the end of the grace period. This policy is similar to how spot instances are pre-empted in the cloud.

In addition to scheduling-driven eviction, resource pressure on a node can also cause pod eviction. In this case, the kubelet monitors the utilization of various resources on a node. If the node utilization rises above a threshold, pods can be evicted to reclaim resources and avoid starvation. Pods are evicted in priority order when reclaiming resources under resource pressure.

Failure handling, Replication, and Autoscaling. Since Kubernetes is designed for interactive cloud services, it has built in features for handling failures and replication of application components.

Kubernetes is designed to automatically handle two types of failures: software failures of the application and node failures. The kubelet on each node monitors all pods deployed on it and containers within it. If the application process inside a container crashes, Kuber-

[1] YAML stands for Yet Another Markdown Language

netes will detect the crash as a result of its monitoring and restart it automatically. While crashes are simple to detect, the application process may encounter other types of failure. For example, the process may simply hang and stop responding or it may run out of memory due to a memory leak and start responding very slowly. To handle such failures, Kubernetes allows the application to specify a liveness probe where it actively checks the health and liveness of the application. This can be done by sending a HTTP GET request, establishing a new socket connection, or by running a script inside the container. If the probe is successful, the container is assumed to be healthy, otherwise the container is restarted if the probe fails.

While the kubelet can detect and handle software failures of the application through passive monitoring or active probing, a different method is needed to handle node failures. If a machine experience a hardware failure or an OS crash, all pods running on it will fail as well. By default, Kubernetes does not take any action when it detects a node failure. However, applications can use a mechanism called the replication controller to handle node failures. The Replication Controller is designed to create a designated number of replicas of an application component. For example, if a web service specifies a degree of replication five for its front-end component, and the application's YAML description specifies the use of the replication controller, Kubernetes will start five pods, each on a different machine, using the replication controller. Importantly, the Replication Controllers will continue to monitor the application controller to ensusure that five replicas are always running in the cluster. The number of replicas may drop below its specified value for two reasons. First, the application's pod may be evicted by a higher priority pod or by the kubelet during times of high utilization and resource pressure. Second, one or more nodes in the cluster may fail. In either case, the ReplicationController will detect that the pod is no longer running and fine a new node to run the pod. The pod is then restarted on that node. An applicaiton must specify that its pods should be mananged by the Kubernetes ReplicationController to take advantage of this feature. Unmanaged pods that are not under the control of the ReplicationController will not be restated if they are evicted or fail due to a node failure. In such cases, the application or the developer will need to handle the failure.

Finally, Kubernetes also supports autoscaling of the application components where the number of replicas is varied dynamically based on a metric. Kubernetes has rich support to monitor a number of applicaiton and system metrics. IT also allows external monitoring frameworks to be used to monitor application and even allows for custom metrics to be defined by an application. The built in auto scaler is designed to perform horizontal scaling based on specified metrics. For example, the application can specify that it should be scaled automatically by the Auto Scaler. To do so, it specifies the minimum and maximum number of replicas that should be maintained by the auto scaler. It also need to specify the metric to use for auto scaling and a threshold that triggers auto scaling. For example, the application can specify that the target CPU utilization should be 50% and new replicas should be created by the Auto Scaler, subject to the maximum allowed number of replicas, when this threshold is exceeded. Similarly, the number of replicas is scaled down whenever the utilization falls below the threshold.

YARN

Omega

Fair Share allocation:

weighted fair queuing, lottery DRF, proportional share, max-min fairness