Distributed Systems

Arthur de Fluiter
afr480, 2536812
afr480@student.vu.nl

Eline van Mantgem
emm300, 1508512
emm300@student.vu.nl

Jelmer Mulder
jmr252, 2526631
j.mulder@vu.nl

Tim Veenman
tvn680, 2515737
t.veenman@vu.nl

Glenn Visser
gvr340, 2555282
g3.visser@student.vu.nl

Joost Heitbrink
jhk810, 2517669
jhk810@student.vu.nl

December 15, 2017

Support cast
A. Iosup
A. Uta
L. Versluis

Abstract

In this paper, we describe an experiment we did in distributed computing. We have designed a scalable and fault tolerant distributed simulator of a distributed grid-scheduler, which we partially implemented. The model offloads tasks amongst clusters of varying sizes, simulating loads that real massive distributed scheduling systems (e.g. DAS, Kubernetes) have to deal with.

Introduction

Modern cloud workloads these days are forced to be distributed by their sheer scale. In addition, many cloud vendors such as Google Cloud Platform and Amazon Web Services run shared workloads on their datacenters to maximize efficient hardware utilization. In order to ensure hardware is uniformly used and no bottlenecks arise, the need for distributed schedulers has risen. These schedulers are efficient in not only scheduling jobs fairly, ensuring a uniform distribution of the workload, but also handle faults in either jobs or hardware gracefully, without compromising the overall integrity of the system.

Prior work

As stated before, distributed scheduling systems are not new. Both in research and commercial contexts distributed schedulers are widely used. We will look at two schedulers: the DAS-5, and Kubernetes.

DAS

The DAS is a distributed cluster computer, spread over six sites in the Netherlands. Each site holds a cluster of servers with industrial-grade CPU's and additional GPU clusters for fast parallel computations. Fast internet connections ensure that each site can communicate with other clusters at high speeds.

This enables users to run large workloads not only in parallel, but independent of geological location. The DAS servers do not enable users to pick nodes on their own, instead users have to use the DAS's scheduler to ensure maximum performance for the specified job. In order to achieve acceptable tenancy as a multi-user distributed research cluster, the DAS-5 has a strict time limit on batch jobs of 15 minutes.

Another interesting design decision of the DAS is that each of the six sites has a so called head-node. This is a special node of the DAS that has no GPU's or special CPU's attached, but instead serves as the sole entry point to the actual DAS worker nodes. Because there is no special hardware on these head nodes, no work can or may be run on them.

This is an interesting departure from a traditional distributed system, since it restricts at which node users may enter the system, and discerns different node types, while most distributed systems try to make a case not to distinguish different nodes.

Kubernetes

While running virtual machines in datacenters, Google quickly found the need to develop a distributed system that could maximize the utilization of the physical hardware, whilst taking into account the different requirements of different customers and their specific needs. The initial scheduling system that Google created, codenamed 'Borg', was a prototype that handled batch jobs and long-running processes[1]. However, due to the developing and more complex needs that arose, Google's improvements to Borg resulted in a second system, called Omega. Omega was built with the same requirements as Borg, but from the ground up, using a more robust Paxos-based transaction store.

However, as cloud computing became more prevalent, a new paradigm for deploying software arose: containers. These containers are best compared to lightweight virtual machines. Furthermore, parties other than Google became interested in deploying containers at scale. Thus, the Kubernetes project was created. While at its core similar to Borg and Omega, the Paxos-based transaction store was swapped out for a more generic distributed key-value store in the form of etcd, which is in turn built upon the raft consensus algorithm.

Background on Application

The Virtual Grid System Simulator

The virtual grid system simulator needs to effectively be able to simulate various distributed workloads, and show how the system performs under various workloads, ranging from long running batch jobs to short lived but frequently arriving jobs. It also needs to be fully distributed, meaning that while the system might run across different nodes, the unexpected termination of one node may not compromise the integrity of the system in any way or form. Furthermore, jobs may not be lost as a result of a node unexpectedly going down.

Job Dispersal

The VGS needs to be able to effectively disperse the incoming jobs to the wider system, so as to not overload one node simply because it receives a lot of requests at a certain point. The job dispersal needs to not only be implemented for efficiency and scalability, but for fault tolerance as well. If a job is not dispersed properly, it might end up being lost if the node it resides on goes down.

Fault Tolerance

The grid must be tolerant of both grid schedulers and resource managers (and their corresponding nodes) leaving the grid without notice. The system must also accommodate newly arriving nodes in an efficient manner. Jobs must also be saved in some way in the system, as to prevent jobs getting lost when a node goes down.

Scalability

Parallel to the fault tolerance requirement of arbitrary nodes arriving in the system, the scheduler must not degrade in performance as the amount of nodes, attendant resource managers and grid schedulers grows. The main challenge here is that while each grid scheduler should be able to communicate with each other grid scheduler in the system, it would at large scales be infeasible to for example have each grid scheduler communicate with all individual nodes.

System design

We have decided to implement our system in Java, to allow us to reuse the provided skeleton code. Additionally we have decided to use Java RMI to facilitate any communication between different hosts. Java RMI takes cares of all the networking details, allowing us to focus on the high level design of the system and quickly develop new features.

The high level overview of the system is as follows: The entire system consists of a number of clusters. A cluster is a number of compute nodes and a resource manager that takes care of scheduling jobs on that cluster. The system also contains grid schedulers, which are responsible for scheduling jobs between clusters. All grid schedulers are aware of each other and can thus communicate with each other. Each grid scheduler is uniquely primarily responsibly for a number of resource managers. Each resource manager communicates mainly with one

Discovery

When a new grid scheduler joins the system, it needs the address of one other grid scheduler and the address of the RMI registry. Once it has registered itself with the RMI registry, it will query the other grid scheduler for a list of active grid schedulers and announce its presence to all of the grid schedulers. By using a simple announcement scheme, we can keep the state grid schedulers need to keep to a minimum, and thus also prevent the need for a complex consensus algorithm.

Logging

One of the requirements is ordered logging of jobs and requests. We achieve this by tagging each event with a Lamport logical clock timestamp to order events. When an event arrives in grid scheduler, it is added to a queue that is sorted by logical clock timestamps. Each event also gets a wall clock time timestamp whenever it enters this queue. The grid scheduler then periodically checks if the wall clock of the head event is sufficiently old, and log it. This way we can achieve almost realtime-logging while still preserving a correct order of events across the entire system.

Job Dispersal

A client wanting to run a job on the system will have to submit it to a resource manager. Resource managers keep a limited queue of jobs. If there is room is this queue, the job is simply added to the queue. If there is no room in the queue, the resource manager will hand off the job to a grid scheduler, which will at that point take over responsibility for the job. Whenever there are empty nodes in a cluster, the resource manager will schedule jobs from the queue.

When a job is handed off to a grid scheduler, this grid scheduler will have to determine whether it is going to schedule it on one of its own clusters, or whether it is going to hand it off to another grid scheduler which in turn will hand it off to another cluster. To make this decision, the grid scheduler asks all other grid schedulers what the capacity is of their resource manager with the highest capacity, and then hands of the job to that grid scheduler. Here we define capacity as $numberOfFreeNodes - jobsQueued$. This maximum capacity for each grid scheduler is cached for a limited duration to reduce the amount of messages in the system. Whenever a grid scheduler receives a hand off request from another grid scheduler, or when it determines that it is itself the grid scheduler with the resource manager with the highest capacity, it will hand off the job to the resource manager with the highest capacity under itself. In such a case the resource manager will queue the job, even if the soft-limit for the queue has been reached already.

Jobs always contain the RMI stub of the issuing client, so when jobs are handed off, no additional state needs to be kept, and no matter in which cluster it ends up completing, that resource manager can notify the client that the job completed.

Fault tolerance

To make our system fault tolerant, we have opted to replicate the resource managers. Practically this means each cluster has two resource managers, that will keep each others state consistent at all times. When either of them becomes unresponsive, the other will able to continue operating as normal. Of course this remaining replica should copy its state to a new replica as soon as possible, to ensure that the system as a whole remains fault tolerant. Our grid schedulers do not keep any state, so they do not need to be replicated.

It is important that the RMI registry server also gets replicated, since all hosts in the system need to communicate with this registry at least once in order to obtain their first RMI stub. Without a replica, this would be a single point of failure for the system (though it would only prevent new hosts from joining the system, existing hosts could continue operating properly).

Scalability

The grid schedulers keep track of other grid schedulers using an internal list and ensure that jobs get evenly distributed, which has the added benefit that at no point in time a node should get overloaded due to an imbalance in job distribution. Scalability is further facilitated by adding extra clusters to nodes, and adding extra grid schedulers to further handle extra workloads. The grid schedulers maintain a list of all other grid scheduler.

This system allows our system to scale to arbitrary amounts of nodes. However, at large numbers of grid schedulers, the size of the list might get restrictive. However, this factor should only come into play with significant amounts of grid schedulers.

Additional system features

Graphical User Interface

Our system also includes a GUI for easy visualization of the system. Each grid scheduler can be viewed as a separate pane, and inside of that pane, the number of resource managers assigned to that scheduler. The nodes are represented as squares, which will change color depending on their state. This allows for an easy visualization of the system.

We also developed a text-only interface, which is easier to utilize in scripts. All output adheres to a strict logging format, making the output easy to parse, allowing for efficient data mining on the data generated by the system.

Experimental Results

Experimental setup

Testing setup

As a test environment, we used one raspberry pi unit in combination with a single linux laptop. Together they would then function as a distributed system. We had to make sure that the Java versions are the same on the laptop and rasp pi for RMI. Once this was fixed we needed nothing other than the setup and our own code. To measure the system we used the standard Linux system monitoring tools (strace, top, syslog etc.) at the two nodes. Furthermore we looked at the output of the program it self to check it's behavior.

Front end

As discussed previously, we implemented two ways of visualizing the VGS, a GUI and a Text-based front-end. The GUI gives a better birds-eye view of what is happening in the system, which is useful to see if the jobs get distributed evenly over all nodes. The text interface was primarily used for performance testing, with the detailed output being easier to parse into usable metrics.

Inputs and jobs

For testing, some scripting was used to automatically generate inputs for the system. The automated test ran 5 grid schedulers 20 resource managers with 50 nodes per resource manager. Furthermore, multiple realistic captures of jobs from other grid workloads were used.[1] These inputs were originally in the Grid Workload Format (or .gwf) format, which we then converted to our own Time Inclined Model format (or .tim) with a conversion script. The script took the information we needed from the Grid Workload Format files and used these Time Inclined Model files to start up jobs from a single instance called a multi-client. This multi-client acts as many different clients processing the jobs that it reads from the Time Inclined Model input file. However, none of the realistic workloads was able to run completely. This was due to time constraints and not simulating time, instead running the captured jobs real time. We only later noticed that these traces included month-long jobs.

[1] http://gwa.ewi.tudelft.nl

Experiment analysis

Consistency

To check consistency we used a small batch of jobs as the system traffic. Enough to make a small load but also small enough that it was easy to follow what was happening. Then we checked in the logs whether or not the tasks arrived correctly at the correct locations. We can not be sure if it will work the same for larger input sets but we expect that behavior will be similar.

Scalability

With the resources available it was hard to test for scalability on a physical level. However it was feasible to test scalability in the system. To achieve this, the system can be started with more nodes, resource managers and grid schedulers. We were able to achieve successful execution with the required amount for each level of the grid.

Fault-tolerance

The system attained partial fault tolerance We tested this by randomly killing parts of the system during runtime and observing how it functions when these things are turned off and what kind of effects this has on the jobs running and the rest of the system. Although in the design the resource managers would be duplicated, due to time constraints this was not implemented. Thus whenever a resource manager fails, the jobs underneath are lost. Other than those jobs the system can continue as normal. The only point of failure is the registry.

Performance

Due to time constraints the project didn't get to accurate performance testing. Considering our time boxing schedule, we lost a little bit more time in development than expected and therefore had to cut the projects loses somewhere else. However we did consider how we would go about testing the system for performance.

The original plan, was to schedule different resource managers (with their nodes) on different DAS-nodes, however due to some issues with RMI, we eventually wanted to include some tests where we'd schedule the entire system on a single node of the DAS and then deploy this with different configurations.

Sadly our experiment failed to accurately time the actual job handling and got stuck trying to measure when certain events would be triggered instead.

Discussion

Remote registry

We developed most of our system on a local machine, simply spawning many processes to simulate hosts. Since we were using Java RMI, we assumed that we could simply run different parts of the system on different hosts without much effort. Unfortunately it turns out that this is not as straightforward as we thought. Apparently the Java RMI registry will not allow you to register hosts on a remote registry, only on registries that are run on the same machine. Our design relies heavily on the assumption that all hosts are able to register themselves with some remote registry, which means that we have not been able to test our system with each all components running on separate hosts, as we had intended. We have tried a number of workarounds, without success. There was not enough time for us to redesign our system.

We have learned a valuable lesson here, which is not to wait until the end of the project to run the a system in the environment in which you wish to run it. In the future we would opt for a more agile approach, with more extensive testing the development process.

Parallel to the remote registry not being able to register remote hosts, we were also confronted with the problem that the registry is one of the most critical components of the entire infrastructure, as each component in the system needs an active registry connection to be able to talk to other nodes. Our initial approach was to replicate the registry, but since we are still limited to the same host, the advantage of replicating the registry is largely nullified, as the registry and its copy will both go down if the underlying node goes down.

Another valuable lesson learned here is that not only does new technologies introduce their own set of requirements and limitations, but those limitations might have unforeseen interactions with system requirements that can severely impact those requirements, as we have seen with our requirement of Fault Tolerance being undercut by the RMI registry.

Conclusion

In this report we have looked at implementing a virtual grid scheduling system. We first analyzed prior work and analyzed the requirements of a virtual grid scheduling system. Afterwards, we designed a system that would optimize job dispersal, scalability and fault tolerance.

However, when we implemented the system we ran into unforeseen difficulties with our underlying technology, namely the Java RMI stack. While we initially chose this to abstract away networking to allow us to focus on the system design, it later turned out that due to severe restrictions on RMI, it undercut several of our requirements.

Thus, while we have designed a working distributed system in theory, practice did not yield the satisfactory results we hoped for.

Appendix: Timesheets

Groups hours is time spend mostly as a group. Man hours is time spend by an individual.

type	time
think time	8 group hours
development time	90 man hours
experiment time	5 group hours
analysis time	2 group hours
write time	35 man hours
wasted time	7 group hours
total time	$(8 + 5 + 2 + 7) * 6 + (90 + 35) = 257$ man hours

References

[1] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.