We recently encountered the problem where a complex simulation program employing Hazelcast in-memory computation in some cases produced so much data that it resulted in an out of memory error. Besides the obvious solution of increasing the number of cluster nodes we decided to take advantage of the powerful resource negotiator abilities of Apache YARN. This post will provide an overview of the two technologies, the challenges developers may face during the integration process, and last, but not least, our solution.

Hazelcast[[1]](#footnote-1) provides a great open source in-memory grid framework that our program uses to cache large amounts of data and to compute optimization solutions in memory. Hazelcast is flexible, provides fast response time and supports complex processing that our simulation requires (but there is also some evidence[[2]](#footnote-2) that the recently graduated Apache Ignite[[3]](#footnote-3) provides better results while not requiring you to pay for exceeding a certain number of nodes like Hazelcast does). It also makes it easy to setup[[4]](#footnote-4) and configure[[5]](#footnote-5) our Hazelcast cluster. Pretty much all you have to do is to download Hazelcast, modify an xml configuration file and launch a shell script.

I’m pretty sure you’re already familiar with the Apache Hadoop framework, but it’s important to reiterate the advantages (and disadvantages) of Apache YARN[[6]](#footnote-6). Hadoop-0.23 provided a major overhaul of the MapReduce framework in response to serious limitations in scalability, reliability, availability, programming model support and resource allocation of the previous versions, giving birth to MapReduce 2.0 or Yet Another Resource Negotiator (YARN). YARN conducts resource management and job scheduling/monitoring in separate daemons. It is highly scalable, available, reliable and serviceable. YARN supports multiple tenants to operate on your cluster simultaneously (multitenancy), moves the process of computation to where your data is (locality awareness), utilizes the physical resources of your cluster flexibly and amazingly, as well as operating securely while enabling great auditability. Furthermore, it supports programming model diversity, provides complete backward compatibility and also support programs that are not MapReduce applications. Probably the greatest disadvantage of YARN is that application development is difficult and requires multiple hours of the programmer’s time to gain an understanding of the framework. There are tools (like Cloudera Kitten[[7]](#footnote-7) and Apache Twill[[8]](#footnote-8)) to simplify the process, but the wide range of issues we encountered while using these tools persuaded us to stick with plain Java. We recommend reading the Great YARN book[[9]](#footnote-9) written by Arun C. Murthy, Vinod Kumar Vavilapalli, Doug Eadline, Joseph Niemiec and Jeff Markham.

We modified YARN Distributed Shell’s source code to setup our Hazelcast cluster and to manage its resources. As mentioned before, YARN supports applications that fall outside of the MapReduce category. In a nutshell, Distributed Shell launches shell scripts on the cluster nodes as well as making the necessary resources available to them. In our case, the shell script was the one that launces the Hazelcast server and the necessary resources were the Hazelcast server files. All of them were located in a zip file that had to be made available to all of our cluster nodes. This can be done using Distributed Shell ‘s Client class, which is responsible for launching the command-line interface, managing local resources and setting up the ApplicationMaster environment. We modified Client to upload our Hazelcast zip file to the YARN local file environment (which is pretty much a temporary HDFS folder). Distributed Shell’s ApplicationMaster decompiles the zip file and launches the final containers where the shell script needed to launch Hazelcast is executed. All of the containers successfully launch Hazelcast instances and they quickly discover each other to form a cluster (we use discovery by TCP to start our Hazelcast Cluster) and ask for resources from

**YARN**

<https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>

Mainly MR, but can distribute many other programs. DistributedShell:

* Pretty much like running a shell script on multiple machines
* Share the same local resources
* „ask” for resources

**Hazelcast**

Hazelcast provides an open source in-memory grid framework

<https://hazelcast.com/use-cases/imdg/>

Easy to install and to configure

**Script**

Java, modification of the DistributedShell scripts that come with Hadoop.

Alternatives like Kitten or Twill, but encountered problems, it proved to be simpler to rewrite the Java source. Low-level, some difficulty.

<https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/WritingYarnApplications.html>

http://www.amazon.com/Apache-Hadoop-YARN-Processing-Addison-Wesley/dp/0321934504

**Hazelcast vs. Ignite**

<http://www.gridgain.com/benchmarking-data-grids-apache-ignite-vs-hazelcast-part-ii/>

1. http://hazelcast.org/ [↑](#footnote-ref-1)
2. <http://gridgain.blogspot.hu/2015/04/benchmarking-data-grids-apache-ignite.html> [↑](#footnote-ref-2)
3. https://ignite.apache.org/ [↑](#footnote-ref-3)
4. http://docs.hazelcast.org/docs/3.3/manual/html/installation.html [↑](#footnote-ref-4)
5. http://docs.hazelcast.org/docs/3.3/manual/html/configuringhazelcast.html [↑](#footnote-ref-5)
6. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html [↑](#footnote-ref-6)
7. https://github.com/cloudera/kitten [↑](#footnote-ref-7)
8. http://twill.incubator.apache.org/ [↑](#footnote-ref-8)
9. http://www.amazon.com/Apache-Hadoop-YARN-Processing-Addison-Wesley/dp/0321934504 [↑](#footnote-ref-9)