Art of Distributed

Part 1 - Rethinking in distributed computing models

Haytham ElFadeel - Hfadeel@acm.org

Abstract

Day after day distributed systems tend to appear in mainstream systems. In spite of maturity of distributed frameworks and tools, a lot of distributed technologies still ambiguous for a lot of people, especially if we talks about complicated or new stuff.

Lately I have been getting a lot of questions about distributed computing, especially distributed computing models, and MapReduce, such as: What is MapReduce? Can MapReduce fit in all situations? How can we compares it with other technologies such as Grid Computing? And what is the best solution to our situation? So I decided to write about distributed computing article into two parts. The first, about distributed computing models and the differences between them. The second part, I will discuss reliability of the distrusted system and distributed storage systems. So let's start…

Distributed Computing:

Distributed computing is form of parallel computing, but parallel computing is most commonly used to describe program parts running simultaneously on multiple processors in same computer. In distributed computing, a program is split up into parts that run simultaneously on multiple computers communicating over a network. Both types (parallel and distributed) require dividing a program into parts that can run simultaneously, but distributed programs often must deal with heterogeneous environments, such as: network links of varying latencies, and unpredictable failures in the network and the computers themselves.

Distributed Computing has many advantages, such as:

- **Pluralism**: Different subsystems of an open distributed system include heterogeneous, overlapping and possibly conflicting information. There is no central arbiter of truth in open distributed systems.
- **Performance, Scalability**: Divide the task or the data into many pieces and execute or process them simultaneously in many computers this will give you good

scalability. So every time you add new computer (node) into the system your system should be able to scale and finish the task faster.

Designing and implementing a distributed system is not easy. In any distributed system, there are many considerations to be made, such as:

- Why, and how you will implement the distributed system, and which model you will choose?
- What about fault tolerance? This is a very important consideration in any distributed system: if you think that everything is reliable, you are wrong, you should take care of fault tolerance, and how you will handle any errors that occur. For example: any 'sick' server shouldn't accept new tasks, and errors in tasks, or data processing should be isolated and handled.

Distributed Computing Models:

There are several models of Distributed Computing, such as:

1- Grid Computing.
2- Master/Worker.
3- MapReduce or 'Aggregation data processing'.

Every model offers its own idea/technique to solve a set of problems. In my opinion, taking an abstract look at distributed models we will find all distributed computing models can be categorized in two groups:

1- **Tasks-Oriented Distributed Computing**: Basically based set of tasks, and sub tasks that executed in parallel in multiple machines.
2- **Data-Oriented Distributed Computing**: Basically based on data that needs to be processed in parallel in multiple machines.

Grid Computing:

The Grid Computing is application of several computers working together on single problem simultaneously. Grid computing depends on software to divide and apportion pieces of a program among several computers, sometimes up to many thousands. Usually, Grid Computing is used to compute intensive batch processing jobs. A typical batch processing job takes a long-running task, and breaks it into smaller tasks, and enables the execution of those tasks in parallel to reduce the time it takes to execute the entire job (Compared with the time it would have taken to execute the tasks sequentially). This model is a good fit for executing relatively compute-intensive and state-

less jobs. A typical scenario for this would be a Monte Carlo simulation, such as the one used to perform risk analysis reports in the financial industry. This type of computing model is more task-oriented than data-oriented. Most compute-grid implementations have the following components:

1. Scheduler.
2. Job executor.
3. Compute agent.

The executor submits jobs, and the scheduler is responsible for taking the job, splitting it into a set of smaller tasks (this process requires specific application code) that are sent in parallel based on a certain policy to a set of compute nodes. The agents on each compute node execute those tasks. The results of those tasks are aggregated back to the scheduler.
The scheduler and agents are responsible for monitoring and ensuring the execution of tasks, and the health of the computing nodes. The scheduler could also support advanced execution policies, such as priority-based execution as well as advanced life-cycle management.

Use example: Heavy computing task, any computing-intensive task.

Master/Worker:

We can consider Master/Worker as a simplified version of Grid Computing, or parallel batch execution. In this model tasks are assumed to be evenly distributed across worker machines. In this case there is no need for an intermediate scheduler. Load-balancing is implicitly achieved through a polling mechanism. Each worker polls the tasks and executes them when it is ready. If a worker is busy, it simply won't process the task~~s~~, and if it is free it will poll the pending tasks and process them.

Consequently, Workers running on a more powerful machine will process more tasks over time. In this way, load balancing is implicit, supporting simple task distribution models. For this reason, master/worker implementations tend to be more useful for simple compute-grid applications. The fact that there is no need for an explicit scheduler makes Master/Worker more performant and better suited for cases where latency is an important factor. The Master/Workers computing model can be considered as tasks-oriented like Grid Computing.

Use example: Web Crawling System, small data processing system, and dispatcher.

MapReduce:

MapReduce is programming model, and software framework introduced by Google to support parallel computations over large data sets on clusters of computers. Now Ma-

pReduce is considered to be a new distributed computing model inspired by the ***map*** and ***reduce*** functions commonly used in functional programming. MapReduce is also called '**Aggregation Data processing**'

MapReduce is Data-Oriented, that process the data in two primary phases: Map, and Reduce. The philosophy behind MapReduce is: Unlike central data-sources, such as a database, you can't assume that all the data resides in one central place, and therefore, you can't just execute a query and expect to get the result as a synchronous operation. Instead, you need to execute the query on each data-source simultaneously, gather the results and perform a 2nd-level aggregation of all the results. To speed the time it takes to run this entire process, the query needs to be done in parallel on each data source. The process of mapping a request from the originator to the data source is called 'Map'; and the process of aggregating the results into a consolidated result is called 'Reduce'.

The MapReduce definition maybe gives you a wrong impression about the possible problems that could solved using this model. Actually over time, the term MapReduce has expanded in definition to describe a more general purpose computing model for executing parallel aggregation of distributed data-sources. Also I just read a research paper on using MapReduce in machine learning algorithms, so MapReduce is a general aggregation data processing model that you can fit into your situation depending on what you want to accomplish.

Today, there are several implementations of MapReduce, such as: Hadoop, Disco, Skynet, FileMap, and Greenplum. Hadoop is the most famous MapReduce implementation that is implemented in Java as an open source project. Hadoop implements the exact specification that defined by Google. As such, it was designed primarily to enable MapReduce operations on distributed file systems and was not really designed as a general purpose parallel processing mechanism.

Use example: Preparinge the index for the Web search engine (Reverse Web-link graph, count word occurrences, count of URL access frequency, etc), distributed data processing.

Other forms of MapReduce:

Over time, the term MapReduce has expanded in definition to describe a more general computing model for executing parallel aggregation of distributed data-sources, rather than referring to a specific type of implementation. GridGain, GigaSpaces, and Terracotta, all took a different approach than Hadoop in their MapReduce implementations. Rather than implementing the exact Google specification in Java, these three aimed to take advantage of the Java language and make the implementation of MapReduce simpler to the average programmers.

GigaSpaces started from the Tuple Space model and JavaSpaces. GigaSpaces was one of the first implementations of the Master/Worker model. At a later stage, it extended the JavaSpaces implementation to a full IMDG (In-Memory Data-Grid). In large scale compute grid applications, the GigaSpaces Data-Grid is often used in conjunction with other Compute-Grid implementations, either commercial or open source. This puts GigaSpaeces in a unique place, providing data-grid and data-aware compute grid capabilities using the same architecture.

How MapReduce differs from other distributed computing models?

While MapReduce represents one form of parallel processing for aggregating distributed data sets. Master/Worker and Grid computing represent as different forms of distributed computing, that is aimed towards Task-Orientation rather than Data-Orientation.

Although both MapReduce and Grid Computing provide a parallel computing model for solving large-scale problems, they are each optimized for addressing a different kind of problem. MapReduce was designed to address shortening the time it takes to process complex data-analytics scenarios. The results of the processing need to be returned in real-time, as the originator of the task normally blocks until its completion. Grid computing applications are aimed at speeding-up the time it takes to process complex computational tasks. The Job is executed as a background process that can often run for a few hours. Users don't typically wait for the results of these tasks, but are either notified or poll for the results. With MapReduce, the application tends to be Data-Oriented; therefore scalability is driven mostly by the ability to scale the data through partitioning. Executing the tasks close to the data becomes critical in this scenario. Grid Computing applications tend to be stateless, and normally operate on relatively small data-sets (compared with those of MapReduce). Consequently, data affinity is considered an optimization rather than a necessity.

When to use Grid Computing, Master/Worker, or MapReduce?

Generally speaking, the best solution depends on your situation (the problem to be solved) and your understanding of the appropriate model. However, here are some notes on how to decide which model fits the best:

- If your application is task-oriented and relatively stateless by nature you should consider Grid Computing.
- If your application is lightweight and task-oriented or you are looking for a real-time (or near-real-time) solution you should consider Master/Worker.

- If your situation is data-oriented and you need to aggregate data that resides on distributed sources or on a distributed file system then I recommend MapReduce.
- If you need to aggregate data that resides in other data sources, such as an in-memory data-grid (IMDG), you should consider GigaSpaces, or a combination of compute grid and data grid products.

In reality, most Task-Oriented applications are not purely stateless. To execute the job the compute tasks need to process data that is coming from either a database or a file system. In small scale applications, it is common practice to distribute the data with the job itself. In large scale compute-grid applications, however, passing the data with the job can be quite inefficient. In such cases, it is recommended to use a combination of Compute and Data Grid. In this case, the data is stored in a shared data-grid cluster and passed by reference to the compute task. So we see the need for a combination of Compute and Data Grids becoming more common today.