CMPSCI 677 Operating Systems **Spring 2022**

Lecture 24: April 27

Lecturer: Prashant Shenoy *Scribe:* ***Avani Jindal***

24.1 Overview

In this lecture, we go over a few slides from the previous lecture covering HDFS (Hadoop Distributed File System) and GFS (Google File System). Then, we move to Distributed Middleware.

24.2 Previous Lecture

24.2.1 Combining LFS with Software RAID

The combination of disk array RAID and log-structured FS gives the notion of a serverless file system called xFS where any data can be stored anywhere. Files are stored as a log. Since we are using network striping, we take these logs and construct large and contiguous segments. Using these segments, we construct a parity and we stripe the entire log and the parity group.

24.2.2 HDFS : Hadoop Distributed File System

This file system is designed for high throughput access to data because it is used in distributed data processing. General purpose file system on our machines are designed for interactive access whereas this is designed for batch applications as distributed data processing is batch oriented. It has a simple model called the WORM (Write Once Read Many) model. We have large datasets and it is unlikely that we will change something after their creation. We might add to it or delete it, but we do not modify it again and again. Here, we try to move the computations to where the data is in order to reduce data movement.

Architecture: HDFS runs on top of a standard file system like the Linux file system. It is basically a layer which takes the file systems of individual machines and constructs a distributed file system out of it. The yellow boxes represent the individual machines and disks on them. The blue node is a metadata server. Some machines serve as metadata servers and they keep track of the location of data. To access a file, we first go to the metadata server and it gives the blocks of where the files are stored. We will get a list of n machines where n is the replication factor. We can go to any one and get that block of file. The green blocks are the replicated blocks. Two principles that we have to keep in mind - separation of data from metadata and that chunks of data are large & replicated on more machines.

24.2.3 GFS : Google File System

It is also designed for large scalability. There is a coordinator and a master node and chunk servers. The master node is the metadata server. It uses a replication factor of 3. It accesses a file in the way similar to HDFS. xFS, HDFS and GFS all have built-in fault tolerance. Here, the files are actually replicated because

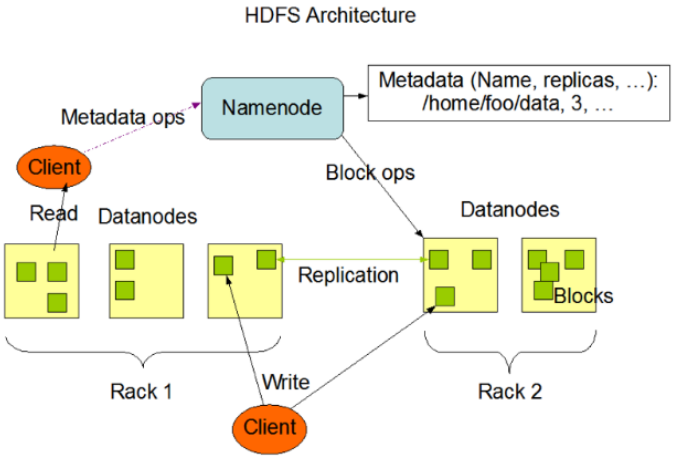

Figure 24.1: HDFS Architecture

we have large amounts of disk space and it is inexpensive to do so. Atomic writes means that when we update a file, we will create a version of it. The write will either go to all replicated files or none. The concepts used by HDFS and GFS are very similar.

24.2.4 Object Storage Systems

This is another way of storing large data files where the storage granularity is objects. The objects do not have a name like a file name, instead they have a handle. In most cases, the handle is actually a hash of the contents of the object. We have get and put operations instead of read and write operations respectively.

Question : What is meant by no block storage but object storage?
Answer : In block storage, the file system partitions the file into blocks and these blocks are stored on disk. In object storage, we can think of the object as being an entire block but there are varying sizes. We do not take an object and make it into smaller blocks and then spread those blocks out on a disk.

Question : What is the advantage of object storage over block storage?
Answer : The use case of object storage is archival storage and backups. We do not optimize object storage system for very heavy throughput applications, instead we optimize it for large amounts of data that we want to store but we are going to do infrequent reads and writes.

Question : Did object storage system evolve with cloud storage?
Answer : Object storage systems are newer than file storage. Cloud was essentially the first popularly available object storage system.

Now, we will switch to Distributed Middleware.

24.3 Distributed Objects

In case of remote objects, code on a client machine wants to invoke an object's method on a server machine. A common way to achieve this, is as follows. Clients have a stub called proxy with an interface matching the remote object. An invocation of a proxy's method is passed across the network to the 'skeleton' on the

server. That skeleton invokes the method on the remote object and returns the marshalled response. This can be recognized from earlier in the course as an RMI or RPC call.

Distributed objects are similar, but the distributed objects are themselves partitioned or replicated across different machines. Distributed objects use RPC. Middleware systems have been developed to support distributed objects.

24.4 Enterprise Java Beans

Enterprise Java is used to write multi-tier applications where the app server is actually written in Java. It also gives some additional functionality like the concept of a bean. A bean is a special type of an object. As a middleware, we will essentially have our objects written as a bean of some sort. It also provides other services like RMI, JNDI, JDBC(used to connect to Databases), JMS(Java Messaging Service). EJBs support more functionality that makes it easier to write web applications.

EJB are fundamentally object oriented, with two components, the interface and the implementation.

24.4.1 Four Types of EJBs

- **Stateless session beans** - They are essentially objects which do not have any state at all, they might just expose code.
- **Stateful session beans** - By default, the memory state is transient and if we kill the application, the object is gone. We can automatically persist the state of the object using these.
- **Entity beans** - They look more like standard Java objects.
- **Message-driven beans** - They are designed for messaging and the messages can persist.

24.5 CORBA : Common Object Request Broker Architecture

At the core of CORBA is the Object Request Broker (ORB) [also called messaging bus] is a intermediate communication channel that allows communication between objects.

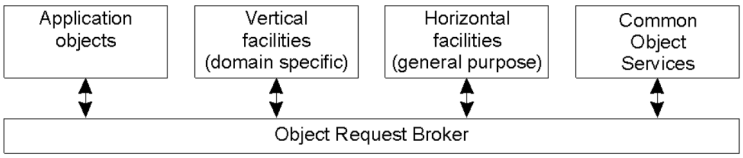

Figure 24.2: CORBA

The four boxes are the four components and they communicate using RPCs handled by the ORB. Many functionalities are already provided in CORBA as a service. They provide a dozen different services from concurrency to licensing in complex distributed systems. The advantage is that they can help reduce the code needed to develop complex distributed systems. However, in trying to provide every service you'd ever need, CORBA became very heavy weight. It does a lot of overhead to write even a small application. By becoming very heavy-weight, it became very difficult to learn and simple distributed applications would require deploying such a heavy weight system. Even though it did not become a commercial success, some

stripped down versions of CORBA actually got used. Example - messaging service in Linux desktop manager called gnome.

The stub in the object model of CORBA is called ORB. It uses Interface Definition Language (IDL) to use an interface and compiler to generate code (like protobufs). Proxy is used to specify the objects and services. Object adapter provides portability between languages. Thus CORBA, is language independent. CORBA provided even more flexibility having the option of invoking RPCs as any type including synchronous, one-way, or deferred synchronous. CORBA was one of the first distributed middleware systems. Modern middleware systems take many ideas from it.

24.5.1 Event and Notification Services

This is done using an event channel. It allows us to implement an application using the publisher-subscriber model. Publishers post events to the event channel, and consumers/subscribers ask for events that they subscribe to from the event channel. Publisher subscriber works with any combination of push and pull. In CORBA, it is a push-push model where data is pushed from publisher to event channel. The event channel will see the list of consumers subscribed and whenever there is a match, it will push to the consumer. In pull-pull model, event channel polls data from the publisher and similarly, consumer pulls data from the event channel.

24.5.2 Messaging

To implement all kinds of RPCs, CORBA has callback model. It will have a callback interface where we can register a callback function. Whenever a reply to the async RPC comes back, we are notified and we can get our reply. We can also use the polling method where we keep polling periodically to see if the reply has come back.

24.6 DCOM : Distributed Component Object Model

DCOM is Microsoft's middleware which has now evolved into .NET. COM is a simple RMI based framework running local to a machine that allowed communication within a machine. It is mainly used for communication between Microsoft applications. Object Linking and Embedding (OLE) was added to allow Microsoft office applications to communicate with one another via embedding and document linking. The ActiveX layer facilitates exposing these services as web applications by allowing us to embed things in web documents. Microsoft picked up this whole thing and made it distributed called DCOM. .NET has a language independent runtime, but ActiveX only works with Internet Explorer.

The architecture of DCOM is fundamentally the same as the distributed objects. The stub is essentially the COM layer. At its core, it is an RMI based system on objects. We also have a SCM (Service Control Manager) which keeps track of what all objects are in the system and where are they running.

The objects in DCOM can also be made persistent even though they are transient by default. It is done using the notion of a Moniker. Moniker is the name of a persistent object that allows us to reconstruct that object after we shut down the server application.

24.7 Distributed Coordination Middleware

In this case, we have a very loose coupling between how communication works. The idea is that we want to separate our computations from coordination. Distributed applications can be classified based on whats happening in time and space dimension. Applications can either be coupled or decoupled in space and time.

1. ⟨ coupled in space and time ⟩ Direct
2. ⟨ coupled in space but not time ⟩ Mailbox - receiver is known but receiver state does not matter.
3. ⟨ coupled in time but not space ⟩ Meeting Oriented - unknown who will show up to meeting.
4. ⟨ decoupled in space and time ⟩ Generative Communication - components can communicate with another without knowing who might read it or when it would be read. It uses a pub-sub model.

24.7.1 Jini Case Study

Jini is a Java based middleware that uses distributed coordination. It facilitates service discovery. We do not know what entities are present in the system so this notion of discovery allows us to discover what services are available and so on. These are also called zero-configuration services because we do not need to pre configure anything as services as discovered on the fly. It uses a event notification system that is pub-sub based. Jini uses a bulletin board architecture. Services advertise on the bulletin board and machines can access the services it requires through the board. It is decoupled in time and space.

In Jini, bulletin board is called JavaSpace or tuple space. Each tuple is a Java object. We have reads and writes into a shared database.

Question : Where is this JavaSpace running?
Answer : JavaSpace is our middleware service which has to run on some server or some set of servers. So essentially, our middleware is running somewhere else and we have to read and write from that.

Jini utilizes a pub-sub architecture with both pull based as well as notification based discovery.

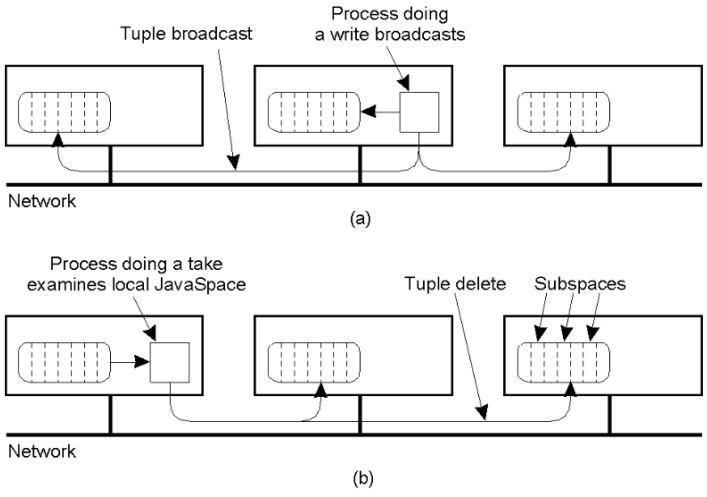

Figure 24.3: Jini Processes

Here, JavaSpace is distributed across multiple machines. The boxes are JavaSpaces with tuples in them. JavaSpaces can either be fully replicated or distributed. In case of replicated JavaSpaces, writes need to be broadcasted to all replicas whereas reads are local. On the other hand, in distributed bulletin board, each board has a subset of nodes, while writes are local and reads need to be done on each and every board.

Question : Whatever we are posting in the JavaSpace, is it a Java object and how is it posted (copied/sent)?
Answer : Tuples are not Java objects, they are data objects like a key and a value. So essentially, we are publishing data or events instead of objects which is different from sending/ receiving objects.

24.8 Big Data Applications

This is mostly covered in CS532 systems for data science. So we will only partially cover this.

Distributed data processing is different types of middleware that are designed for processing large amounts of data. The basic idea is that we will use multiple machines of a cluster and parallelize our application for data processing. Each machine will read and process some part of the data.

24.9 MapReduce Programming Model

MapReduce is a framework for processing parallelizable problems across large datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes are on the same local network and use similar hardware) or a grid (if the nodes are shared across geographically and administratively distributed systems, and use more heterogenous hardware). Processing can occur on data stored either in a filesystem (unstructured) or in a database (structured). MapReduce can take advantage of the locality of data, processing it near the place it is stored in order to minimize communication overhead.

MapReduce is a two-stage process to process a large dataset - Map phase and Reduce phase.

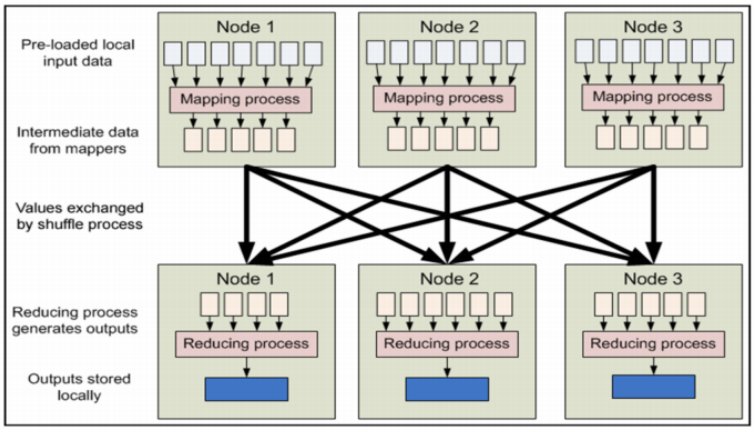

Figure 24.4: Map Reduce example

Map Step : Each worker node applies the "map()" function to the local data, and writes the output to a temporary storage. A master node ensures that only one copy of redundant input data is processed.

Shuffle Step : Worker nodes redistribute data based on the output keys (produced by the "map()" function), such that all data belonging to one key is located on the same worker node.

Reduce Step : Worker nodes now process each group of output data, per key, in parallel.

Example: Let's say we have huge dataset of number (or may be words in a document) which need to be sorted (or frequency of words). Suppose we have multiple machines. Then each machine could take a chunk of data and sort it. Later, all these sorted chunks chould be merge together similar to merge sort algorithm. However, this requires huge communication between the machines during merge phase. To overcome this problem, let's say we know the range of numbers. In such a case, we could follow the bucket sort paradigm where each machine sorts a specific range of numbers. This way, inter-machine communication can be reduced.

The datasets being processed here are actually stored on HDFS. Each node can read its local data from HDFS or it can also read remote data. Reading locally has lesser overheads and is cheaper.

24.9.1 Other Programming Models

Apache Tez An application framework which allows for a complex directed-acyclic-graph of tasks for processing data.

Microsoft Naiad Naiad is system for data-parallel dataflow computation which attempts to raise the levels of abstraction used by programmers from an imperative sequence of MapReduce-style statements, to involve higher level concepts of loops and streaming.

Spark Apache Spark is a fast and general engine for big data processing, with built-in modules for streaming, SQL, machine learning and graph processing. DAG with in memory resilient data sets.

Flink Apache Flink® is an open-source stream processing framework for distributed, high-performing, always-available, and accurate data streaming applications. DAG models extended to cyclic graphs.

24.10 Hadoop Big Data Platform

Hadoop is an implementation of Map-Reduce framework. It is an open-source software framework for storing data and running applications on clusters of commodity hardware. It provides massive storage for any kind of data, enormous processing power and the ability to handle virtually limitless concurrent tasks or jobs. It has :

- **store managers** : where the datasets are stored. Eg: HDFS, HBASE, Kafka, etc (replication for fault tolerence.
- **processing framework** : Map-reduce, Spark, etc
- **resource managers** : allocates nodes and resources to jobs. Some of the concepts of distributed scheduling are also adopted since they need to serve multiple users. Example : Yarm, Mesos, etc

24.10.1 Ecosystem

Based on the requirements different types of frameworks can be used. For example, if user wants to process the data that have lot of graphs then Graph processing framework Giraph can be used. There are machine learning frameworks like MLLib, Oyyx, Tensorflow that are also designed to run to on Hadoop. If a user wants to input data to these distributed processing framework, he could use applications like hive to easily

write map-reduce codes. For real time data processing where data is generated continuously by some external source, framework like Spark Storm etc could be used.
We can see that it is not a single distributed application, it is a set of applications that work together.

24.11 Spark Platform

Apache Spark is a powerful open source processing engine built around speed, ease of use, and sophisticated analytics. It was originally developed at UC Berkeley in 2009. Spark was an important innovation over MapReduce. Although MapReduce uses parallelism, it is very heavy on I/O that can slow down the application. In Spark, we store intermediate data in memory of some server. What we decide to store and how to store is something we have to think about when writing a Spark application.

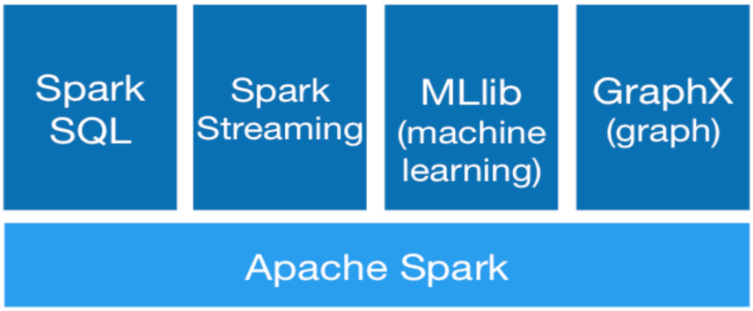

Figure 24.5: Spark Platform

Spark SQL : Spark SQL is a Spark module for structured data processing. It provides a programming abstraction called DataFrames and can also act as distributed SQL query engine. It enables unmodified Hadoop Hive queries to run up to 100x faster on existing deployments and data. It also provides powerful integration with the rest of the Spark ecosystem (e.g., integrating SQL query processing with machine learning).

Spark Streaming Many applications need the ability to process and analyze not only batch data, but also streams of new data in real-time. Running on top of Spark, Spark Streaming enables powerful interactive and analytical applications across both streaming and historical data, while inheriting Spark's ease of use and fault tolerance characteristics. It readily integrates with a wide variety of popular data sources, including HDFS, Flume, Kafka, and Twitter.

MLlib Built on top of Spark, MLlib is a scalable machine learning library that delivers both high-quality algorithms (e.g., multiple iterations to increase accuracy) and blazing speed (up to 100x faster than MapReduce). The library is usable in Java, Scala, and Python as part of Spark applications, so that you can include it in complete workflows.

GraphX GraphX is a graph computation engine built on top of Spark that enables users to interactively build, transform and reason about graph structured data at scale. It comes complete with a library of common algorithms.

Adavatages of Spark

- **Speed :** Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing.

- **Ease of Use:** Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it interactively from the Scala, Python and R shells.
- **Generality :** Combine SQL, streaming, and complex analytics.Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming. You can combine these libraries seamlessly in the same application.
- **Runs Everywhere:** Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

24.11.1 Spark Architecture

Distributed memory is just like memory but we access data across various machines. All of the memories of the servers can be accessed if we store data in the form of an RDD. The idea is that we will first read data from disk, say HDFS. Then we will do some partial processing, then transformed dataset will be stored in RDD and so on. Since data is in memory, processing will be much faster.

RDD is designed to support in-memory data storage, distributed across a cluster in a manner that is demonstrably both fault-tolerant and efficient. Fault-tolerance is achieved, in part, by tracking the lineage of transformations applied to coarse-grained sets of data. Efficiency is achieved through parallelization of processing across multiple nodes in the cluster, and minimization of data replication between those nodes. Server failures can be handled by recomputation.