CMPSCI 677 Operating Systems **Spring 2015**

Lecture 2: January 26

Lecturer: Prashant Shenoy *Scribe: Amee Trivedi*

2.1 Overview

- Announcement
- Overflow slides
- Flavors of Distributed Systems

2.2 Announcement

- There will be weekly quiz put up on Moodle. The quiz will close on Sunday EOD.
- The class was introduced to Prateek, who is the TA for the course.

2.3 Distributed Operating Systems

The Uniprocessor operating system was the stepping stone to distributed system. It used a monolithic architecture where a single kernel acted as a single entity. Then came the micro-kernel architecture where the kernel was reduced to a set of essential services such as IPC and security and the other functionality were defined as user level processes that collborate with OS level components through message passing. The advantages of this approach was micro-kernel had a modular design and gave better security. If a module had bugs or would mal-function then it was easy to isolate bugs. However, the disadvantage was there was more message passing involved to get a task done resulting in performance penalty.

A pure distributed operating system manages resources seamlessly and transparently to the user. To the user it seems like a single centalized OS. The DOS does this by providing a set of transparencies such as Location, Access, Concurrency, Migration, Relocation, Replication and Failure. There are several types of distributed operating systems:

DOS This is the standard distributed operating system with a tightly-coupled operating system for multiple processors and (or) multiple homogeneous computers. These systems, as described above hides hardware resource management from the user.

NOS A network operating system is a loosely-coupled system for multiple heterogeneous computers communicating over a LAN and (or) WAN. In this system the transparency is less. The machines are not hidden and are openly visible. Login details are needed to remotely access them. They are simpler to design due to less transparency. This includes the common client-server architecture where users have to select machines with services they want to use. Each machine in this type of distributed system has a network OS services without much transparency (e.g. SSH).

Middleware Middleware takes a NOS and makes it look like DOS. This is a layer that is built upon a NOS that implements some generalized services. These systems, such as Sun's Grid Engine, keep track of the load on each machine and will schedule jobs automatically based upon that information.

Multiprocessor OS In this system there are multiple CPU's but the details are hidden from the user. To the user it looks like a single system. The operating system takes care of scheduling of applications on different processors.

Multicomputer OS Each machine has its own kernel and a layer above of distributed operating system services. This layer makes the multiple machines underneath transparent and the user just has to submit jobs and the system will determine how to schedule and run the jobs. An example of this type of system is a MOSIX cluster.

2.4 Architectural Styles

There are few main architecture styles of a distributed systems:

2.4.1 Layered Design

In this approach the distributed application is mapped into layers. Each layer can communicated with the immediate layer above it and the immediate layer below it, so, Layeri will communicate with Layer(i+1) and Layer(i-1). The classic example of layered design is the protocol stack. The network protocol stack is a classic example of this design in use. Some simple multi-tier web applications also use this design.

2.4.2 Object-based

In this architecture an application is divided into objects and the objects reside on different machines. These objects interact with each other through RPC. The object based style is a generalization of layered design with no restriction on the entity communication. This architectural style is popular in client-server applications.

2.4.3 Event-based

In this approach an application is broken into multiple components and they communicate via a common repository or event bus. The components are distributed and they generate events. It works on the publish-subscribe paradigm. Components "publish" events which are then delivered via the event bus. Example: Stock-prices

2.4.4 Shared Data-Space

Components in this type of architecture are decoupled in both space and time. It uses the Bulletin-board architecture. Communication between components occurs by each component posting something to a shared space which other interested components can then later "pick up" and use. A main difference between this an other architectures is that the recipients are not well-known.

2.4.5 Client-Server

This is the simplest type of architectural style that is used for distributed systems. Functionality is partitioned into two components: client and server. The clients make requests to the servers and servers provide some kind of service. Further, it could be synchronous method where the client requests for a service to server and then waits till the time the server sends a reply. Applications using this architecture can be layered: user-interface level, processing level, data level. What is the responsibility of the client versus the server is up to the application designer.

There is a wide spectrum of choices based on the application definition. The spectrum of choices consists of dividing the functionality of an application as client side features and server side features. The 3 main layers: User interface, Application and database can be divided to either be at client side or server side. The choice is made based on the type of application if it is web based or fat client.

2.5 Decentralized Architectures

When a client-server architecture is modified to remove any distinction between a client and a server and all machines are made equal, we get a peer-to-peer system. Peer-to-peer (P2P) is a type of decentralized architecture because, rather than having a more powerful server machine servicing requests, all machines are equal. Any peer is able to do whatever any other peer can do. These systems are often used to store and retrieve data items using a key-value pair. Decentralized architectures can be structured or unstructured. To illustrate what structured decentralized architectures are we give a couple of examples:

Chord This system automatically forms a logical ring structure by using a distributed hash table to locate data items. Each peer is given a node id and data items are hashed to make a key. This key is used to find the node which is responsible for that data item. For a key k, the smallest node with $id \geq k$ has the given data item. When a node is added, the data must be re-partitioned.

CAN A Content Addressable Network, or CAN, is a d-dimensional coordinate system. The key for a data item has multiple dimensions (e.g. filename, file type, etc). The nodes in the system split up the d-dimensional space and take responsibility for data items within a region. For two dimensions this looks like a standard 2d coordinate space and the regions are rectangles (which keeps queries simple). When a node joins or leaves the regions are redistributed into new rectangular regions (most likely preserving much of the regions that existed prior to the join or departure). A node joining is easy because it only requires an existing region to be split at a random point. Nodes leaving is more difficult because the resulting areas might not default to being rectangular.

Unstructured P2P Unstructured peer-to-peer systems form a topology based on randomized algorithms rather than by some predefined structure. When a node joins, it will randomly select a set of nodes to become neighbors with. There is no way to assign responsibility to each node unlike with structured P2P networks. Queries must be performed by "flooding" since there is no set responsibility. A flood query will have a node ask all of its neighbors who then do a lookup for the data item. If a neighbor cannot find the data item, then it will ask all of its neighbors (and so on). Further, to avoid a message from looping forever in a network cycle, a message table is maintained by each node. The table keeps track of messages that have been forwarded by a node so that it can check and not forward an already forwarded message. Additionally, while sending a query a "hop count" can be decided which states that for how many hops the message should be forwarded. The hop count is reduced every time the message is forwarded until it equals 0. The hop count helps in deciding the query distance and it will also aid in restricting query to a subset of nodes in network.