CMPSCI 677 Operating Systems **Spring 2013**

Lecture 2: January 28

Lecturer: Prashant Shenoy *Scribe: Demetre Lavigne*

2.1 Distributed Operating Systems

A distributed operating system is a collection of cooperating systems that looks like a single centralized OS to the user. The cooperating systems have multiple independent CPUs. This is done by managing dispersed resources in such a way that their use is both *seamless* and *transparent* to the user. 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. 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 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 This is a system which presents a uniprocessor operating system to the user by hiding the complexities of multiple processors (or multiple cores). Each processor in this system may have various level of hardware cache.

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.

2.2 Architectural Styles

2.2.1 Layered Design

Functionality is broken upto multiple *layers* in a layered design. Each layer can only use the functionality exported by the previous (below) layer to build new functionality which is then available to the layer above. The only communication that a layer can have is with its two neighbouring layers. The network protocol stack is a classic example of this design in use. Some simple multi-tier web applications also use this design.

2.2.2 Object-based

Functionality is broken up into multiple *objects* similar to in Java. Each object exports an interface so that other objects can invoke the functionality of other objects. Object methods are generally invoked via *remote procedure calls* (RPCs) or *remote method invocations* (RMIs). This architectural style is popular in client-server applications.

2.2.3 Event-based

The functionality is again broken up into individual components and there is an event bus on which communication can occur. These systems use the publish-subscribe model. Components "publish" events which are then delivered via the event bus. This is similar to how interrupts work. We will look at this architectural style in more depth later in the course.

2.2.4 Shared Data-Space

Components in this type of architecture are decoupled in both space and time. Communication between components occurs by each component posting something to a shared space which other interested components can then later "pick up" and use. It is not known how much later, after being published, a data item will be picked up. A main difference between this an other architectures is that the recipients are not well-known. Given its similarity to the real world equivalent, this is sometimes referred to as the "*bulletin-board*" architecture.

2.2.5 Client-Server

Functionality is partitioned into two components: client and server. The clients make requests to the servers and servers provide some kind of service. This is the most common type of architectural style that is used for distributed systems. 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. For example, part of the processing might be done by the client and the rest by the server (with the client responsible for UI and the server responsible for data storage). Another example would be if the client only handled the user interface and the server was in charge of processing and data (this is typical for mobile phone applications). The slides include a good diagram of different possible combinations of responsibilities split between the client and the server.

2.3 Decentralized Architectures

A client-server architecture can be modified to remove any distinction between a client and a server to create 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 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 neighbours 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 neighbours who then do a lookup for the data item. If a neighbour cannot find the data item, then it will ask all of its neighbours (and so on).

Using flooding to perform queries will not scale well to many nodes. This is because the amount of processing and communication needed grows extremely fast. This caused the creation of *SuperPeers*. A super peer is a nodes with slightly more responsibilities than the common peer. Groups of nodes elect a "leader" who will answer queries for the entire group. This means that a super peer must keep track of what data items other nodes have. An example of this is Skype's super-peers who track what users are logged in and where they are.