CMPSCI 377 Operating Systems **Fall 2012**

Lecture 21: November 27

Lecturer: Prashant Shenoy *TA: Sean Barker & Demetre Lavigne*

21.1 Network Topologies

21.1.1 Point-to-Point Topologies

The topology of a network is defined by how the links between nodes are created. In a **fully connected** network, all nodes are connected to all other nodes. This provides the benefit that any node can reach any other in only "one hop", and that the system is resilient to failures since one node crashing will not cause any other node to become disconnected from the rest. However, this can become expensive when there is a large number of nodes, and is not practical for WAN environments where there can be huge numbers of nodes spread across wide distances.

A **partially connected** network loosens these requirements by only using links between a subset of the nodes. As a result, a message sent between two nodes may need to be directed through several other nodes along the path, requiring some sort of routing algorithm for coordination and increasing transmission times. This also makes this type of network less resilient to node failures. However, since this is significantly less expensive than a fully connected network, it is often used for WANs.

Another network topology is a **tree structure**. In this case, a root node is created, and all others connected to its children or their descendants. This provides a simple structure that often matches the logical hierarchy of an enterprise, but it can lead to slow performance since messages may have to pass up through multiple nodes to find a common ancestor for their intended recipient. Trees are also not resilient to failure, since one node failing causes all of its descendants to become disconnected from the root.

A **star** topology makes all nodes be directly connected to a single centralized node. This uses much fewer links than a fully connected network, but still allows all nodes to communicate within two hops. However, the central node is a single point of failure and can lead to the full network becoming disconnected.

21.1.2 Ring & Bus Topologies

Another class of topologies is a ring network—these topologies are very simple, but their restrictive structure can produce more predictable behavior. In a simple **one directional ring**, a node can only send messages in one direction, leading to a maximum message delay of $n-1$ hops in a network with n nodes. A one direction ring network will get partitioned by a single node failure. Allowing messages to travel in both directions reduces the maximum delay to $n/2$ hops, but the system can only tolerate one failure by increasing the number of hops.

A ring network can be created with additional links in order to gradually increase the level of connectivity. A **double connected ring** is one where each node is connected to its immediate neighbours, as well as the nodes two hops away. This allows messages to hop through the network more quickly, giving a maximum delay of $n/4$ hops and increasing the resilience to failure. Further increasing the number of links provides better performance and reliability at greater expense. Note that a ring network with links to all nodes is simply a fully connected point-to-point network.

A **bus network** relies on special links that can be shared by multiple nodes. For example, multiple components within a single computer communicating over the shared system bus can be considered a form of distributed system. A **linear bus** simply has multiple nodes connected to one bus that supports multiple access. This is inexpensive, but nodes may need to coordinate to ensure that only a single node uses the bus at any one time. Ethernet LAN links use this structure. A **ring bus** is the same as a linear bus but it "loops" around.

21.2 Distributed File Systems

A distributed file system is one of the most common types of distributed system. It is very common for a large number of computers to want to be able to have shared access to some set of data or program files. A distributed file system provides this by giving the abstraction that a shared disk appears to be attached to each node in the system, even though in reality it is only physically attached to one or more server nodes. This is commonly used in educational computer labs where users want access to their files regardless of which specific computer they log in to.

21.2.1 Naming and Transparency

In order to be useful, users must be able to address (or **name**) the shared files they would like to access. Different distributed file systems support naming in different ways. Some systems provide **location transparency**, which states that the knowing the name of a file does not reveal its physical storage location. This can be desirable for security reasons if administrators do not want users to necessarily know exactly where different files are stored. This can also produce a simpler system because it means that users do not *need* to know the exact storage location, either. **Location independence** says that even if a file is moved between physical storage locations, its name will still remain valid. This can be beneficial since users do not need to repeatedly look up where a file is located even if it is moved between disk drives. In practice, many distributed file systems support location transparency, but few provide location independence.

21.2.2 Naming Strategies

An **absolute name** is one which provides a complete address to a file including both the server and path names. This has the advantage that it is trivial to find a file once the name is given since it contains complete information. This means that no additional state must be kept since each name is self contained, which can lead to greater scalability. However, users must know the complete name of the server and path to files they want to access and because of the naming convention there is a clear difference between local and remote files. This limits transparency since users must deal with local and remote files differently, and makes it harder to changes such as when a file moves between hosts. This can also make the system less resilient to failure since there is no abstraction layer which could re-map addresses (names) if one server failed but another contained the same data. This technique is used in operating systems such as Windows and AppleShare.

An alternative approach is to use **mount points**. In this case, the client machine creates a set of "local names" which are used to refer to remote locations. These names are called mount points because the remote files or directories are connected and attached to the local file system. The operating system must maintain a table (such as /etc/fstab in Linux) to maintain the mapping of what server and path are mapped to each mount point. When the system boots up, it scans through the table and connects to each remote server. It then translates any file accesses to the mount point into network calls which are transferred to the remote file system. This provides location transparency described previously, because once the mapping is made, the local client does not need to know or care that the files it is using are actually located across the network. This allows the remote server to be changed without affecting the local file name mappings, although the system may need to be restarted to acknowledge the change. The main disadvantage of using mount points is that it can lead to confusion since two different local names may actually map to the same file on a remote system. This approach is commonly used in Linux and Sun operating systems.

A third option for naming is to use a **global name space**. In this case, all nodes within the system have an identical name space—the path and name of a file on one machine will be the same on every other machine, regardless of where the file is actually stored. This is typically implemented using a set of dedicated file servers that store all files for the system. When a client boots up, it contacts one of the file servers and receives the layout of the distributed file system. When a user accesses a file, the server sends a copy of it to the client machine where it is cached. As the client updates the files, the changes must be written back to the central file servers. The advantage of this approach is that naming is consistent across all clients. Also, the storage servers are able to seamlessly move files around because clients always contact the server to learn where files are located within the global name space. However, the fact that files are cached by clients can lead to challenges in keeping file content consistent across all nodes. Enforcing a global name space across all nodes also limits flexibility, and can lead to performance problems, particularly when the scale of the system grows. Examples of global name space systems include: CMU's Andrew, Berkeley's Sprite, and the web.

21.2.3 Remote File Access and Caching

When a client wants to modify a file contained in a distributed file system, it can perform all operations remotely, or use local caching. In the first case, the client sends the server an operation such as a write call, and the server performs the write and returns a result (often this is done using RPC). Alternatively, when a write is to be performed, the server can send the file (or part of the file) back to the client so that it can make the changes locally; this results in the client building up a local cache of files which it is working with.

If caching is used, then the distributed system faces additional challenges since it must figure out how to propagate changes back to the server after they have been made. This is especially complicated when multiple clients may be accessing the same file—the system must preserve consistency so that one clients actions do not conflict with those made by another.

A second challenge is deciding whether clients should keep its cache in memory or on its local disk. Using the local disk is significantly faster than making accesses over the network, but it is still much slower than keeping the file in memory. However, the disk is a persistent store, so it can provide greater reliability in the case that the node fails before writing its changes back to the file server. Of course, writing to disk also requires that the client *has* a disk, which is not always the case in "thin client" systems. Alternatively, the files can be cached in memory, but while this provides greater speed, it is less reliable and results in a more limited cache size.

21.2.4 Cache Update Policy

A cache update policy defines when writes made to a cache should be propagated back to the original disk. Using a **write through** policy provides high reliability since writes are immediately written to the server. However, the user will see reduced performance because all writes not only need to be made to the local disk, but transmitted over the network and acknowledged by the server. As a result, the cache provides no benefit for write requests (identical to doing all writes remotely), but it can still improve performance for reads.

In a **write back** policy, writes to the remote server are delayed until some event occurs such as the file being closed, the block being removed from the cache, or some set delay. This leads to better performance because writes can be queued up and done asynchronously as the local program continues executing. This can also reduce network traffic by exploiting the fact that a single block may be written and overwritten multiple times in a short window—with a write back policy, these changes can be aggregated into a single write which is sent to the server. Unfortunately, write back cannot provide the same reliability guarantees as write through since the node may crash or lose power before the write is sent back to the file server.

Cache Consistency is another important issue that determines how caches are maintained when multiple clients may be accessing the same file. In a **client-initiated consistency** system, the client is responsible for checking with the server to verify that each file in its cache is consistent (e.g. that no one else has modified the file since it was cached). Depending on the level of consistency required, the client could verify that its cache is consistent on every access, at a given interval, or only when the file is first opened. This is a relatively simple protocol to implement, but it requires the servers to trust that clients will indeed verify that their caches are correct—a single corrupt or malicious client could disrupt the complete system.

In **server-initiated consistency**, the server acts as a central authority over which clients have up to date or invalid caches. In this case, the server must track information about all clients that have cached a file so it knows which parts of which files are currently cached, as well as whether a given client is reading or writing to a file. Using this information, the server is able to detect when reading and writing clients might conflict with each other, and will send messages to clients to force them to invalidate their cache entries and request them again.