Lecture 20: November 20

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

20.1 Buffering & Caching

The nature of I/O requests is such that either the same or nearby blocks may be accessed multiple times in a short interval. To improve performance, I/O devices typically include a small on-board memory where they can temporarily store data before transferring it to or from the CPU. This allows a disk to buffer data to be read from, or written to, the device while the DMA controller transfers it to memory.

Using buffers both within the OS and on a device allows the system to improve I/O performance or consistency. This is particularly important when the speed of the device and the CPU are very different—the CPU may quickly write a disk block to a buffer, and then the device will more slowly spool it out to disk.

The use of caches also helps the OS improve performance by reducing the number of device operations that must actually be performed. For example, the OS can keep a memory cache holding recently used disk blocks. When a new disk read comes in, the OS checks whether the disk block is already stored in the cache, allowing it to be returned immediately. Storing blocks in a cache can significantly improve read performance, but it also complicates writes. In a **write-through** policy, when the OS makes a write it applies it both to the memory holding the block and to the disk itself immediately. This provides high reliability since all writes are known to have been made to disk. However, faster write performance can be obtained with a **write-back** policy which only writes the update to the memory at first, and queues the write to disk to be performed sometime later. A write-back policy allows many incremental changes to a file to be written to disk at the same time rather than with multiple smaller writes. This can lead to very fast writes, but results in a weaker reliability model since the programmer cannot be sure that a disk write has truly gone to disk.

20.2 Distributed Systems

A distributed system is one composed of multiple physically separated processors that are connected by communication links. A distributed system is typically considered **loosely coupled** because each of the processing nodes have their own memory and independent OS, and only periodically communicate in order to maintain synchronization. In this type of system, each processing node can run a different OS. In contrast, a **tightly coupled**, or **parallel**, system is one where multiple processors run under the control of a single operating system. The advantage of distributed systems is that they allow multiple systems to pool their resources. This kind of **resource sharing** can be as simple as having multiple PCs share a printer, or as complicated as a distributed set of machines all providing a shared storage system. Having the additional resources of a distributed system leads to **computation speed-up**. Ideally, having n processors should give you a factor of n speed-up, although in practice this is very difficult to achieve due to communication overheads and the need to cleanly decompose a problem into smaller sub-problems.

Distributed systems can also provide higher **reliability** by using replication to points of failure. This allows a distributed system to keep working even if one or more nodes crash. However, the system needs to be carefully configured in order to prevent a central server crashing affecting all other nodes in the system. Finally, distributed systems form the basis for **communication** systems such as e-mail and websites. This

allows software on machines all over the world to communicate and perform tasks. Easier communication also means that security is more of a concern because malicious users can have the same access as legitimate users do.

While each node in a distributed system typically runs its own operating system, the distributed system as a whole also must provide services similar to an OS. For example, the entire distributed system needs a way to manage resources, provide communication, ensure security and reliability within the system, and provide high performance even as the system scales to a large number of nodes. A distributed system also needs to provide a mechanism for timing, or synchronization, of each node's clock. There is also a concern about **transparency** meaning how much should end users know about the complexity of a distributed system that they are using. For example, a user of a web search engine doesn't really need to know how their query is broken up and processed by thousands of nodes.

20.3 Networks

Networks are responsible for providing efficient, correct, and robust message passing between two separate nodes. A **Local Area Network (LAN)** is used to connect nodes within a small area (e.g. within a single building), and is designed to provide high speed communication. The nodes in a LAN may communicate wirelessly, or they may all be connected via cables such as coaxial or fiber optics. The bandwidth provided by a LAN can range from 10Mbps-100Mb/s in a typical small LAN up to 1000Mb/s or even 10Gb/s in a well provisioned network. In contrast, a **Wide Area Network (WAN)** is designed to work over longer distances, and thus typically provide slower, less reliable links. WAN connections may be run over telephone lines, microwave links, or even satellite channels. The bandwidth provided in WAN links to a home or small business is much lower than in a LAN—a T1 link for a small business provides only 1.544Mb/s transfer speeds. LANs may also be connected through various WANs to create a vast network of networks (e.g. the Internet).

20.3.1 Communication Protocols

A communication protocol is a set of rules that are agreed upon by both nodes that want to send messages to each other. These rules define things such as the order that nodes should talk, the format that data should be sent, etc. The rules are typically implemented as part of a **protocol stack**—a set of software layers that convert application level requests into the format required by the network, transmit the data between nodes, and finally converts it back into messages that the destination application can understand.

The International Standards Organization Open Systems Interconnect **(ISO/OSI)** model defines the standard network protocol stack used in modern operating systems. It uses a layered architecture with a set of layers that communicate only with those immediately above or below them. The layers, from highest to lowest are:

- Application Layer: contains the user processes which desire to send messages over the network, e.g. web browsers or IM clients.
- Presentation Layer: performs data conversion (e.g. big/little endian format) since the destination and source nodes may not be running the same architecture.
- Session Layer: includes libraries that implement the communication strategy such as Remote Procedure Calls (RPC) or simple message passing.
- Transport Layer: ensures reliable, end-to-end communication between two nodes. This layer is provided within the operating system, and the OS may support multiple different policies (e.g. TCP and UDP).

- Network Layer: performs routing and congestion control to ensure that messages can reach their destination and that resources are used efficiently within the network. Usually implemented within the OS.

- Data Link Control Layer: ensures reliable point-to-point communication of packets even over unreliable network paths. Can be implemented either within the network card hardware or in software.

- Physical Layer: controls the electrical or optical signalling across a "wire", e.g. the actual transmission of bits between two nodes. Implemented in the network card hardware.

20.3.2 TCP/IP Protocol Stack

The Transmission Control Protocol/Internet Protocol (TCP/IP) is the most common protocol stack used in network communication. In practice, many systems implement only the TCP/IP stack which contains four layers, rather than the full seven layer model proposed by ISO/OSI.

TCP/IP defines a suite of protocols to be used for the Transport Layer, that specifies how messages should be sent between hosts over the full end-to-end path. **TCP** is the primary transport protocol used on the Internet today because it provides a **reliable** transfer mechanism. TCP enforces the rules that any packet sent will eventually be received by the destination (assuming it does not become disconnected) and that packets will arrive in order. A second protocol, **UDP** can be used instead of TCP, but it is an **unreliable** protocol—no guarantee is given that packets sent will arrive or arrive in any particular order. While it may seem like TCP would always be preferable to UDP, TCP must give up some performance in order to make its guarantees about reliability. As a result, UDP is still often used for applications which can deal with some amount of packet loss but require low latency, e.g. audio streaming or video games. The TCP/IP stack also defines **IP**, a protocol which is used as a lower layer for transporting packets underneath of either TCP or UDP.

A **packet** is the minimal unit of transfer in a TCP or UDP network flow. When an application desires to send a message, the TCP/IP stack will split it up into multiple fixed size packets. Each packet acts as a self contained message that knows its source, destination, and ordering. This means that while a destination host may receive a set of packets out of order, it is able to rearrange them into the proper order. When packets are received at the destination, they are combined to rebuild the original message.

20.4 Client/Server Model

One of the most common models for building a distributed system is the Client/Server model. In these systems, there is generally a single node that acts as a **server**, running software which provides some sort of service such as a database, web server, or file server. In some cases, the server may be run on several nodes in order to improve performance or reliability. The server node is accessed by other nodes running **client** software. The clients connect to the server and send it requests to perform some action. The server runs the request on behalf of the client and returns a response. This is a simple structure that clearly divides which nodes in the distributed system are responsible for which tasks.

In order to request an action to be performed, the clients can use either message passing or remote procedure calls (RPC). In message passing, the clients sends a message which contains some kind of instructions of what task to perform, and the server returns a response which the client must parse. RPC is discussed in the following section.

20.5 Remote Procedure Calls

A Remote Procedure Call is a way for making an action performed by another node appear as if they were a simple function call. With RPC, the client software contains a *stub* method which is a special function which bundles up its arguments and transmits them to the server that will run the RPC call. The server then waits for these bundled requests to arrive, processes them, and then returns a bundle to the client with the result. Meanwhile, the client's stub method waits for the reply; when it is received, it unpacks the bundle and returns the results of the RPC as if it were an ordinary function call. In this way, the OS and the RPC library are able to completely mask all of the networking details from the application making the calls.

In order for RPCs to work, the server needs a way to advertise what functions it can run, and the clients need a way to find the servers. This process is called **naming**, and refers to both how the server registers its available functionality, and how the clients determine how to reach the servers to make calls. The servers define **signatures** of the functions they support, which define what input parameters they expect and what the return value type will be. Based on these signatures, a **stub compiler** builds a stub functions to be used by the clients and servers so they agree on how the method will be invoked.

The Java programming language supports RPC calls using the Remote Method Invocation (RMI) library. Many other languages support RPC through specialized libraries, and RPC is the most common model for communication between nodes in distributed systems. In fact, RPC can even be used within a single system in order to allow for simple inter-process communication.