CS162
Operating Systems and Systems Programming
Lecture 20

Reliability and Access Control / Distributed Systems

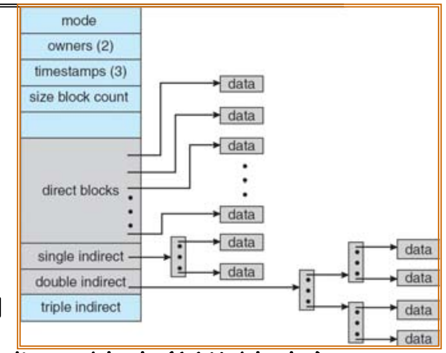
November 10, 2010

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

Review: Example of Multilevel Indexed Files

- Multilevel Indexed Files: (from UNIX 4.1 BSD)
  - Key idea: efficient for small files, but still allow big files
  - File Header format:
    - » First 10 ptrs to data blocks
    - » Block 11 points to "indirect block" containing 256 blocks
    - » Block 12 points to "doubly-indirect block" containing 256 indirect blocks for total of 64K blocks
    - » Block 13 points to a triply indirect block (16M blocks)



- UNIX 4.1 Pros and cons
  - Pros: Simple (more or less)
    Files can easily expand (up to a point)
    Small files particularly cheap and easy
  - Cons: Lots of seeks
    Very large files must read many indirect block (four I/Os per block!)

Review: UNIX BSD 4.2

- Inode Structure Same as BSD 4.1 (same file header and triply indirect blocks), except incorporated ideas from DEMOS:
  - Uses bitmap allocation in place of freelist
  - Attempt to allocate files contiguously
  - 10% reserved disk space
  - Skip-sector positioning
- BSD 4.2 Fast File System (FFS)
  - File Allocation and placement policies
    - » Put each new file at front of different range of blocks
    - » To expand a file, you first try successive blocks in bitmap, then choose new range of blocks
  - Inode for file stored in same "cylinder group" as parent directory of the file
  - Store files from same directory near each other
  - Note: I put up the original FFS paper as reading for last lecture (and on Handouts page).
- Later file systems
  - Clustering of files used together, automatic defrag of files, a number of additional optimizations

Goals for Today

- File Caching
- Durability
- Authorization
- Distributed Systems

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Where are inodes stored?

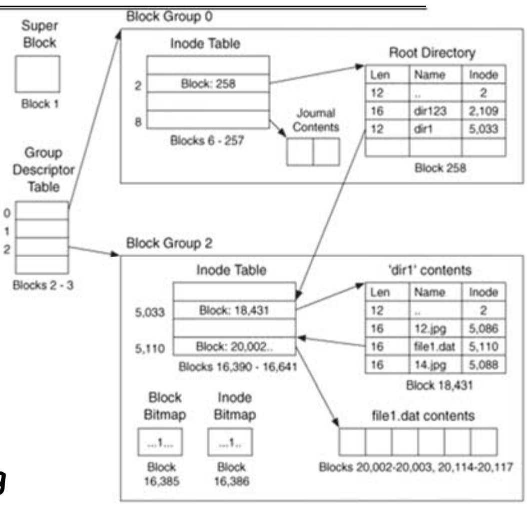
- In early UNIX and DOS/Windows' FAT file system, headers stored in special array in outermost cylinders
  - Header not stored near the data blocks. To read a small file, seek to get header, seek back to data.
  - Fixed size, set when disk is formatted. At formatting time, a fixed number of inodes were created (They were each given a unique number, called an "inumber")

Where are inodes stored?

- Later versions of UNIX moved the header information to be closer to the data blocks
  - Often, inode for file stored in same "cylinder group" as parent directory of the file (makes an ls of that directory run fast).
  - Pros:
    - » UNIX BSD 4.2 puts a portion of the file header array on each cylinder. For small directories, can fit all data, file headers, etc in same cylinder⇒no seeks!
    - » File headers much smaller than whole block (a few hundred bytes), so multiple headers fetched from disk at same time
    - » Reliability: whatever happens to the disk, you can find many of the files (even if directories disconnected)
  - Part of the Fast File System (FFS)
    - » General optimization to avoid seeks

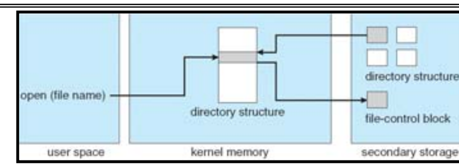
Linux Example: Ext2/3 Disk Layout

- Disk divided into block groups
  - Provides locality
  - Each group has two block-sized bitmaps (free blocks/inodes)
  - Block sizes settable at format time: 1K, 2K, 4K, 8K…
- Actual Inode structure similar to 4.2BSD
  - with 12 direct pointers
- Ext3: Ext2 w/Journaling
  - Several degrees of protection with more or less cost

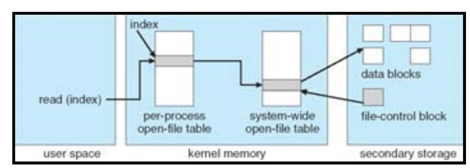


- Example: create a *file1.dat* under */dir/* in Ext3

In-Memory File System Structures



- Open system call:
  - Resolves file name, finds file control block (inode)
  - Makes entries in per-process and system-wide tables
  - Returns index (called "file handle") in open-file table



- Read/write system calls:
  - Use file handle to locate inode
  - Perform appropriate reads or writes

File System Caching

- Key Idea: Exploit locality by caching data in memory
  - Name translations: Mapping from paths→inodes
  - Disk blocks: Mapping from block address→disk content
- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain "dirty" blocks (blocks not yet on disk)
- Replacement policy? LRU
  - Can afford overhead of timestamps for each disk block
  - Advantages:
    - » Works very well for name translation
    - » Works well in general as long as memory is big enough to accommodate a host's working set of files.
  - Disadvantages:
    - » Fails when some application scans through file system, thereby flushing the cache with data used only once
    - » Example: `find . -exec grep foo {} \;`
- Other Replacement Policies?
  - Some systems allow applications to request other policies
  - Example, 'Use Once':
    - » File system can discard blocks as soon as they are used

File System Caching (con't)

- Cache Size: How much memory should the OS allocate to the buffer cache vs virtual memory?
  - Too much memory to the file system cache ⇒ won't be able to run many applications at once
  - Too little memory to file system cache ⇒ many applications may run slowly (disk caching not effective)
  - Solution: adjust boundary dynamically so that the disk access rates for paging and file access are balanced
- Read Ahead Prefetching: fetch sequential blocks early
  - Key Idea: exploit fact that most common file access is sequential by prefetching subsequent disk blocks ahead of current read request (if they are not already in memory)
  - Elevator algorithm can efficiently interleave groups of prefetches from concurrent applications
  - How much to prefetch?
    - » Too many imposes delays on requests by other applications
    - » Too few causes many seeks (and rotational delays) among concurrent file requests

File System Caching (con't)

- Delayed Writes: Writes to files not immediately sent out to disk
  - Instead, `write()` copies data from user space buffer to kernel buffer (in cache)
    - » Enabled by presence of buffer cache: can leave written file blocks in cache for a while
    - » If some other application tries to read data before written to disk, file system will read from cache
  - Flushed to disk periodically (e.g. in UNIX, every 30 sec)
  - Advantages:
    - » Disk scheduler can efficiently order lots of requests
    - » Disk allocation algorithm can be run with correct size value for a file
    - » Some files need never get written to disk! (e..g temporary scratch files written /tmp often don't exist for 30 sec)
  - Disadvantages
    - » What if system crashes before file has been written out?
    - » Worse yet, what if system crashes before a directory file has been written out? (lose pointer to inode!)

Administrivia

Aside: Command Queueing

- Mentioned that some disks do queueing
  - Ability for disk to take multiple requests
  - Do elevator algorithm automatically on disk
- First showed up in SCSI-2 timeframe
  - Released in 1990, but later retracted
  - Final release in 1994
    - » Note that "MSDOS" still under Windows-3.1
- Now prevalent in many drives
  - SATA-II: "NCQ" (Native Command Queueing)
- Modern Disk (Seagate):
  - 2 TB
  - 7200 RPM
  - 3Gbits/second SATA-II interface (serial)
  - 32 MB on-disk cache

Important "ilities"

- **Availability:** the probability that the system can accept and process requests
  - Often measured in "nines" of probability. So, a 99.9% probability is considered "3-nines of availability"
  - Key idea here is independence of failures
- **Durability:** the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone
- **Reliability:** the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply availability: means that the system is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems

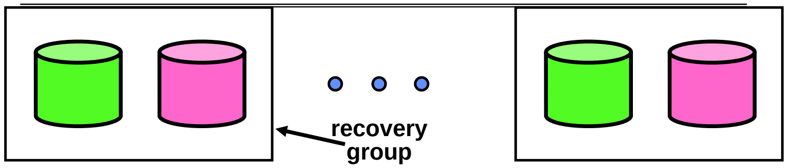
What about crashes? Log Structured and Journaled File Systems

- Better reliability through use of log
  - All changes are treated as *transactions*.
    - » A transaction either happens *completely* or *not at all*
  - A transaction is *committed* once it is written to the log
    - » Data forced to disk for reliability
    - » Process can be accelerated with NVRAM
  - Although File system may not be updated immediately, data preserved in the log
- Difference between "Log Structured" and "Journaled"
  - Log Structured Filesystem (LFS): data stays in log form
  - Journaled Filesystem: Log used for recovery
- For Journaled system:
  - Log used to asynchronously update filesystem
    - » Log entries removed after used
  - After crash:
    - » Remaining transactions in the log performed ("Redo")
- Examples of Journaled File Systems:
  - Ext3 (Linux), XFS (Unix), etc.

Other ways to make file system durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
  - Can allow recovery of data from small media defects
- Make sure writes survive in short term
  - Either abandon delayed writes or
  - use special, battery-backed RAM (called non-volatile RAM or **NVRAM**) for dirty blocks in buffer cache.
- Make sure that data survives in long term
  - Need to replicate! More than one copy of data!
  - Important element: **independence of failure**
    - » Could put copies on one disk, but if disk head fails…
    - » Could put copies on different disks, but if server fails…
    - » Could put copies on different servers, but if building is struck by lightning….
    - » Could put copies on servers in different continents…
- **RAID:** Redundant Arrays of Inexpensive Disks
  - Data stored on multiple disks (redundancy)
  - Either in software or hardware
    - » In hardware case, done by disk controller; file system may not even know that there is more than one disk in use

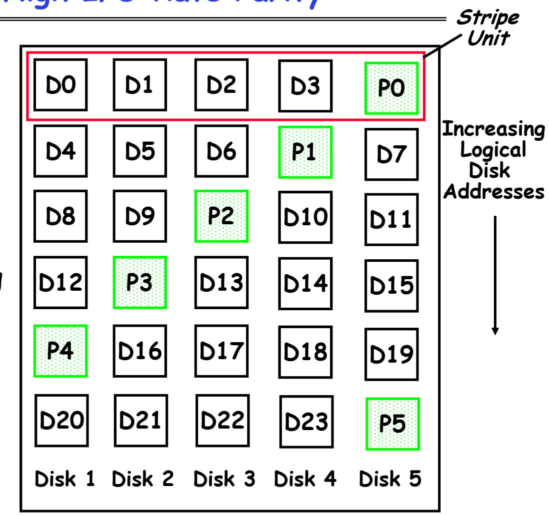
RAID 1: Disk Mirroring/Shadowing



- Each disk is fully duplicated onto its "shadow"
  - For high I/O rate, high availability environments
  - Most expensive solution: 100% capacity overhead
- Bandwidth sacrificed on write:
  - Logical write = two physical writes
  - Highest bandwidth when disk heads and rotation fully synchronized (hard to do exactly)
- Reads may be optimized
  - Can have two independent reads to same data
- Recovery:
  - Disk failure ⇒ replace disk and copy data to new disk
  - Hot Spare: idle disk already attached to system to be used for immediate replacement

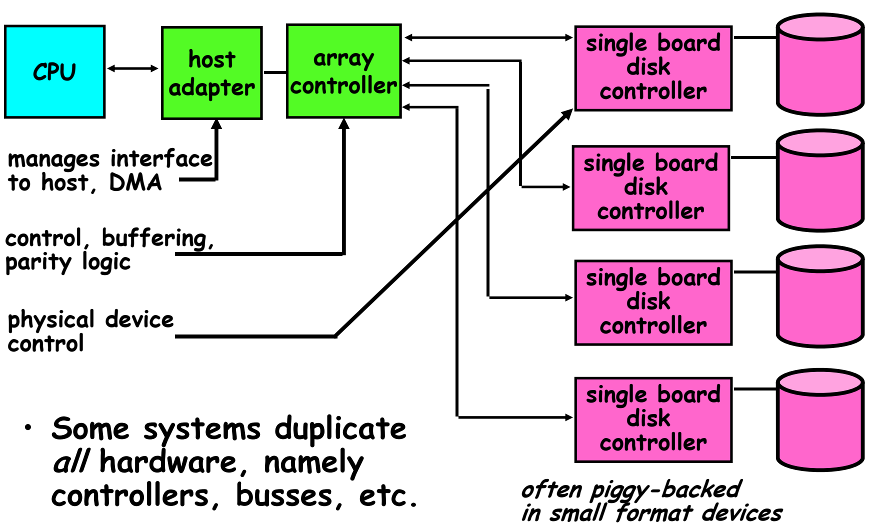
RAID 5+: High I/O Rate Parity

- Data stripped across multiple disks
  - Successive blocks stored on successive (non-parity) disks
  - Increased bandwidth over single disk
- Parity block (in green) constructed by XORing data bocks in stripe
  - P0=D0⊕D1⊕D2⊕D3
  - Can destroy any one disk and still reconstruct data
  - Suppose D3 fails, then can reconstruct: D3=D0⊕D1⊕D2⊕P0



- Later in term: talk about spreading information widely across internet for durability.

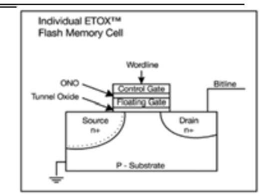
Hardware RAID: Subsystem Organization



- Some systems duplicate *all* hardware, namely controllers, busses, etc.

Solid State Disk (SSD)

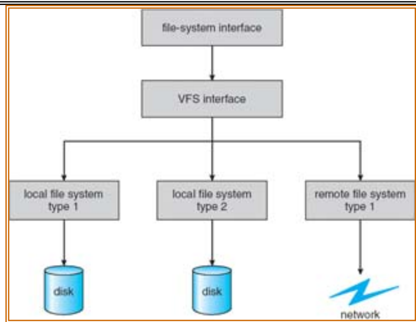
- Becoming Possible to store (relatively) large amounts of data
  - E.g. Intel SSD: 80GB – 160GB
  - NAND FLASH most common
    - » Written in blocks – similarity to DISK, without seek time
  - Non-volatile – just like disk, so can be disk replacement



Trapped Charge/No charge on floating gate
MLC: MultiLevel Cell

- Advantages over Disk
  - Lower power, greater reliability, lower noise (no moving parts)
  - 100X Faster reads than disk (no seek)
- Disadvantages
  - Cost (20-100X) per byte over disk
  - Relatively slow writes (but still faster than disk)
  - Write endurance: cells wear out if used too many times
    - » 10^5 to 10^6 writes
    - » Multi-Level Cells ⇒ Single-Level Cells ⇒ Failed Cells
    - » Use of "wear-leveling" to distribute writes over less-used blocks

Remote File Systems: Virtual File System (VFS)

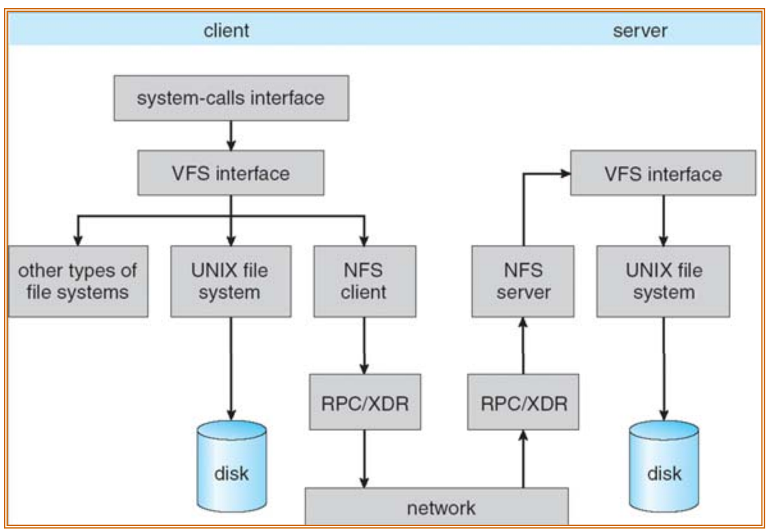


- **VFS:** Virtual abstraction similar to local file system
  - Instead of "inodes" has "vnodes"
  - Compatible with a variety of local and remote file systems
    - » provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - The API is to the VFS interface, rather than any specific type of file system

Network File System (NFS)

- Three Layers for NFS system
  - **UNIX file-system interface:** open, read, write, close calls + file descriptors
  - **VFS layer:** distinguishes local from remote files
    - » Calls the NFS protocol procedures for remote requests
  - **NFS service layer:** bottom layer of the architecture
    - » Implements the NFS protocol
- NFS Protocol: remote procedure calls (RPC) for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes/reading and writing files
- NFS servers are **stateless**; each request provides all arguments require for execution
- Modified data must be committed to the server's disk before results are returned to the client
  - lose some of the advantages of caching
  - Can lead to weird results: write file on one client, read on other, get old data

Schematic View of NFS Architecture



Authorization: Who Can Do What?

- How do we decide who is authorized to do actions in the system?
- **Access Control Matrix:** contains all permissions in the system
  - Resources across top
    - » Files, Devices, etc…
  - Domains in columns
    - » A domain might be a user or a group of users
    - » E.g. above: User D3 can read F2 or execute F3
  - In practice, table would be huge and sparse!

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Authorization: Two Implementation Choices

- Access Control Lists: store permissions with object
  - Still might be lots of users!
  - UNIX limits each file to: r,w,x for owner, group, world
  - More recent systems allow definition of groups of users and permissions for each group
  - ACLs allow easy changing of an object's permissions
    - » Example: add Users C, D, and F with rw permissions
- Capability List: each process tracks which objects has permission to touch
  - Popular in the past, idea out of favor today
  - Consider page table: Each process has list of pages it has access to, not each page has list of processes …
  - Capability lists allow easy changing of a domain's permissions
    - » Example: you are promoted to system administrator and should be given access to all system files

Authorization: Combination Approach

- Users have capabilities, called "groups" or "roles"
  - Everyone with particular group access is "equivalent" when accessing group resource
  - Like passport (which gives access to country of origin)
- Objects have ACLs
  - ACLs can refer to users or groups
  - Change object permissions object by modifying ACL
  - Change broad user permissions via changes in group membership
  - Possessors of proper credentials get access

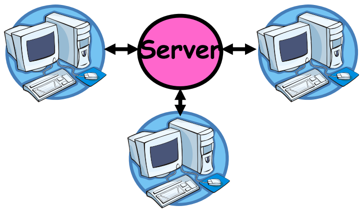
Authorization: How to Revoke?

- How does one revoke someone's access rights to a particular object?
  - Easy with ACLs: just remove entry from the list
  - Takes effect immediately since the ACL is checked on each object access
- Harder to do with capabilities since they aren't stored with the object being controlled:
  - Not so bad in a single machine: could keep all capability lists in a well-known place (e.g., the OS capability table).
  - Very hard in distributed system, where remote hosts may have crashed or may not cooperate (more in a future lecture)

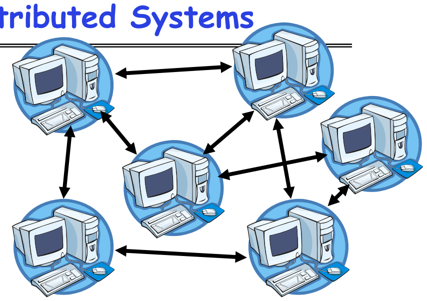
Revoking Capabilities

- Various approaches to revoking capabilities:
  - Put expiration dates on capabilities and force reacquisition
  - Put epoch numbers on capabilities and revoke all capabilities by bumping the epoch number (which gets checked on each access attempt)
  - Maintain back pointers to all capabilities that have been handed out (Tough if capabilities can be copied)
  - Maintain a revocation list that gets checked on every access attempt

Centralized vs Distributed Systems



Client/Server Model



Peer-to-Peer Model

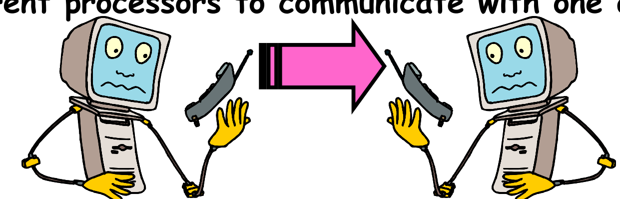
- **Centralized System:** System in which major functions are performed by a single physical computer
  - Originally, everything on single computer
  - Later: client/server model
- **Distributed System:** physically separate computers working together on some task
  - Early model: multiple servers working together
    - » Probably in the same room or building
    - » Often called a "cluster"
  - Later models: peer-to-peer/wide-spread collaboration

Distributed Systems: Motivation/Issues

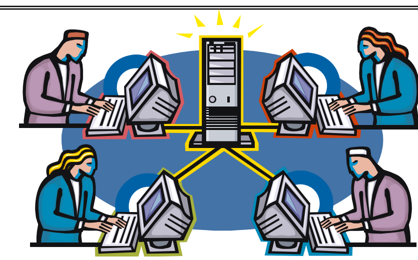
- Why do we want distributed systems?
  - Cheaper and easier to build lots of simple computers
  - Easier to add power incrementally
  - Users can have complete control over some components
  - Collaboration: Much easier for users to collaborate through network resources (such as network file systems)
- The *promise* of distributed systems:
  - Higher availability: one machine goes down, use another
  - Better durability: store data in multiple locations
  - More security: each piece easier to make secure
- Reality has been disappointing
  - Worse availability: depend on every machine being up
    - » Lamport: "a distributed system is one where I can't do work because some machine I've never heard of isn't working!"
  - Worse reliability: can lose data if any machine crashes
  - Worse security: anyone in world can break into system
- Coordination is more difficult
  - Must coordinate multiple copies of shared state information (using only a network)
  - What would be easy in a centralized system becomes a lot more difficult

Distributed Systems: Goals/Requirements

- **Transparency:** the ability of the system to mask its complexity behind a simple interface
- Possible transparencies:
  - **Location:** Can't tell where resources are located
  - **Migration:** Resources may move without the user knowing
  - **Replication:** Can't tell how many copies of resource exist
  - **Concurrency:** Can't tell how many users there are
  - **Parallelism:** System may speed up large jobs by spliting them into smaller pieces
  - **Fault Tolerance:** System may hide varoius things that go wrong in the system
- Transparency and collaboration require some way for different processors to communicate with one another



Networking Definitions



- **Network:** physical connection that allows two computers to communicate
- **Packet:** unit of transfer, sequence of bits carried over the network
  - Network carries packets from one CPU to another
  - Destination gets interrupt when packet arrives
- **Protocol:** agreement between two parties as to how information is to be transmitted

Conclusion

- Important system properties
  - Availability: how often is the resource available?
  - Durability: how well is data preserved against faults?
  - Reliability: how often is resource performing correctly?
- Use of Log to improve Reliability
  - Journaled file systems such as ext3
- RAID: Redundant Arrays of Inexpensive Disks
  - RAID1: mirroring, RAID5: Parity block
- Authorization
  - Controlling access to resources using
    - » Access Control Lists
    - » Capabilities
- Network: physical connection that allows two computers to communicate
  - Packet: unit of transfer, sequence of bits carried over the network