Deadlock Conditions: Mutual Exclusion, Hold-and-Wait, No Preemption, Circular wait **Mutual Exclusion:** Threads claim exclusive control of resources that they require **Hold and Wait:** Threads hold resources allocated to them while waiting for other resources **No Preemption:** Resources cannot be forcibly removed from threads that are holding them **Circular Wait:** There exists a circular chain of threads such that each thread holds one or more resources that are being requested by the next thread in the chain **Total Ordering:** Prevents circular wait; force threads to lock files in specific order **Deadlock Avoidance:** We only need to stop one of the four conditions to prevent deadlocks; we can modify the scheduler to not allow two threads to be scheduled concurrently **Detect and Recover:** Fallback strategy; detect deadlock and reboot system when it happens **File Systems:** Collection of files and directories that provide persistent storage **CPU:** Central Processing Unit of a computer; primary control of the machine **Memory:** Where instructions and data are stored within a computer **Bus:** Communication system that transfers data between components; groups of wires **Peripheral/Device:** Internal or external device that connects to a computer to do something **Device Controller:** Bridge between CPU and I/O devices; handles signals(both in and out) **Device Driver:** Software that operates a device; gives computer access to functionalities **Interrupt:** Signal emitted by a device/program that signals the OS to do something **DMA(Direct Memory Access):** How the OS interacts with a device; allows CPU to run a process during the copying process and I/O of a different process **Hard Disk Drive(HDD):** Main form of persistent data storage **Sectors:** Blocks within the address space to break down storage further **Track:** Each circle around the center of the drive, a surface has thousands of tracks **Rotational Delay:** Time for a sector to rotate around the disk head **Seek Time:** Time for the disk arm to change position to the correct track **Random Access:** The software requests addresses in random order, causing random latency **Sequential Access:** Software requests addresses in increasing order **Shortest Seek Time First:** Pick requests on the nearest track to complete first **File:** An array of bytes that has an identifier assigned to it called an inode number **Directory:** A list of pairs that also has an inode. Pairs have a name and an inode number **Root Directory:** The head of the directory tree; root directory has no parent directory **Absolute Path:** The path through the directory tree to get to a file; /bar/foo/bar.txt **File Descriptor:** A file's index in the computer's array of files(**Process Control Block**) **open():** Takes a file name and options, opens a file in the current directory **read(int fd, void *buf, int count):** Reads count bytes from file fd into the buffer buf **write(int fd, void *buf, int count):** Writes count bytes from buffer buf to the file fd **close(int fd):** Closes file fd, file descriptor no longer refers to a file and can be reused **lseek(int fd, int offset, int whence):** Changes the location of the next read/write **Hard Link:** Connects a filename to an inode; filename1 filename2 **Soft(Symbolic) Link:** Gives an alias to a name; -s filename1 filename2 **Metadata Region:** A few blocks on the disk reserved for the file system metadata **Data Region:** The rest of the blocks on the disk that are used to store data of the files **Contiguous Allocation:** Similar to memory allocation for data; causes external fragmentation **Inode:** Contains information about a particular file **Data Bitmap:** Stores which blocks on the disk are free/used within the data region **Inode Bitmap:** Stores which blocks are free/used in the metadata region **Direct Indexing:** A way to index where the inode of a file points to the data blocks of the file **Indirect Indexing:** An indirect pointer points to an inode with more pointers **Double/Triple Indirect Indexing:** Double has two levels of indirection, triple has three, etc **Caching:** Hold popularly used blocks to decrease number of times blocks are read from disk **Write Buffering:** Batch multiple updates into one smaller set of I/O operations **Berkeley Fast File System(FFS):** Keeps related stuff together to limit searching **Block Groups:** A grouping of blocks that has a superblock, inode blocks, and data blocks **Path Locality:** Consecutive file accesses are likely to be near each other **Journaling:** Before making a change, store a note to the disk about what is going to happen **Transaction:** One thing that will happen to the system, should be on the note **Commit:** Write the transaction to the commit block **Checkpoint:** Write the metadata and data to their final locations in the system **Flash-Based Solid State Drive(SSD):** Made from flash memory, no moving parts **Cell:** Transistor creates this; stores a bit in a single-level cell or up to three bits in trip-level cell **Page:** Holds several cells, typically is around 4 KB **Block:** Consists of multiple pages, typically around 128 KB in size **Read Page:** Client provides a page number to read; doesn't depend on location of page **Erase Block:** Block must be set to all 1; tends to be slower than read **Program Page:** Writes the pages by setting 1's and 0's; time is between read and erase **Flash Translation Layer(FTL):** Where the flash controller virtualizes the flash **Log-Structured FTL:** Uses an in-memory table to map virtual pages to physical pages; Every write moves the page to a different physical location **Garbage Collection:** Removes old content that was left on the disk but isn't needed anymore **Wear Leveling:** Flash block has limited number of erases before it can't be used anymore **Virtual Machine Monitor/Hypervisor(VMM):** Creates illusion of being hardware to an OS **Supervisor Mode:** VMM reduces the privilege mode before calling guest OS trap handler **Information Gap:** If the OS has no useful work, it just spins in its scheduler loop **Para-Virtualization:** Guest OS has small modifications to operate more effectively **Confidentiality:** Only allow information maintained by a system to be accessed by authorized **Integrity:** Only allow a resource to be modified by authorized parties **Availability:** System can be accessed at requested times by authorized parties **Authenticity:** A computer system can verify the identity of a user **Intruder:** Those who attempt to breach the security of a system **Vulnerability:** Weakness in the security of a system **Threat:** Anything that leads to a loss or corruption of data/physical damage to hardware **Attack:** Attempt to break the security; can be either accidental or malicious **Malware:** Software designed to exploit, disable, or damage a system **Trojan Horse:** Program that looks legitimate but can take control of your computer **Spyware:** Program frequently installed with legit software, but displays ads/captures data **Ransomware:** Locks up data via encryption, demands payment to unlock it **Sniffing:** Intruder "sniffs" communication between users to get secret data and information **Masquerading:** Intruder communicates with the receiver to gain intel or information **Man-in-the-Middle:** Intruder communicates with both the sender and receiver to gain info **Denial of Service(DoS):** Overload the target computer to prevent it from doing work **Access Control:** Regulating what actions subjects can perform on general objects **Object Access Control:** Resources on which an action can be performed(files, tables, etc.) **Subject Access Control:** Human users representing users from which objects are protected **Access Mode:** Controllable actions of subjects on objects **Access Control Matrix:** Describes the access modes granted to subjects on objects **Access Control List:** Alternative to matrix; each column of matrix is stored in separate list **Capability:** Group of three things containing a subject, object, and access mode **Encryption:** Passing plaintext through an algorithm to get an unreadable group of characters **Decryption:** Passing the unreadable characters through an algorithm to get readable text **Plaintext:** Human readable text, most files are in plaintext **Ciphertext:** What comes out of an encryption algorithm; unreadable by humans **Symmetric Key System:** Same key is used to encrypt and decrypt; key must be secret **Asymmetric Key System:** Public-key encryption based on each user having two keys **Public Key:** Published key that is used to encrypt data **Private Key:** Key known only to an individual user that is used to decrypt data **Authentication:** Identify users and verify that they are who they say they are **Nonce:** A once in a lifetime number to use in one session, new number for every session **Defense in Depth:** Use multiple layers of defense; multiple independent methods used **Layered Architecture:** A way to design software that used defense in depth **Trust:** A trusted system is relied upon to an extent to enforce a specific security policy **Trusted Computing Base(TCB):** Everything in trusted system necessary to enforce policy **Trusted Platform Module(TPM):** Hardware that controls what can be done on the machine **Virtual Machine:** Present the users with only the resources they need; fake an OS **Sandbox:** Protected environment where a program can run and not endanger the system **Honeypot:** Fake environment that is intended to lure in an attacker **Protocol:** Defines the interfaces between layers in the same system/same layers **Link:** Physical connection between nodes **Packet:** Block of data being communicated **Host:** Computer/Device connected to the network **Switch:** Nodes with multiple links that forward data/packets from one link to another **Router:** Forwards data between networks **Client/Server:** Has two types of communication channels; request/reply and message stream **OSI Architecture:** Open Systems Interconnection, seven different layers **Physical Layer:** Handles transmission of raw bits over a communication link **Data Link Layer:** Collects streams of bits into a frame, delivers frames to hosts **Network Layer:** Handles routing among nodes within a packet-switched network **Transport Layer:** Implements a process-to-process channel, exchanges messages **Session Layer:** Provides a namespace; ties together different streams of one application **Presentation Layer:** Concerned about the format of data exchanged between users **Application Layer:** Standardizes the common type of exchanges **Internet Architecture:** Does not imply strict layering; Wide at the top/bottom, narrow in the middle; needs to be protocol specification and at least one representative implementation **Socket:** The point where a local application process attaches itself to a network **IP Address:** Identifies a single device within a network **Port:** Port numbers are used to identify individual processes; 16-bits unsigned **TCP:** Transmission Control Protocol; communication standard, allows message exchanges **UDP:** User Datagram Protocol; Used for communication throughout the internet **Bind:** Binds newly created socket to a specified address, address combines IP and port **Listen:** Defines how many connection can be pending on the specified socket **Accept:** Carries out the passive open; does not return until connection is established **Connect:** Does not return until TCP has established connection; invokes send and recv **Bandwidth:** Width of the frequency band **Latency:** Transmit + Propagation + Queue **Propagation:** Distance / Speed of light **Stateful Server:** Maintains information about the client's state; slow recovery, complex **Stateless Server:** A server that does not have a client state **NFS:** Stateless protocol, server remembers nothing from the previous requests **File Handle:** Has a volume identifier, inode number, and generation number **LOOKUP:** Obtain a file handle **READ:** Read from a file at specified location for x bytes **WRITE:** Write to file at specified location for x bytes **GETATTR:** Get file attributes **Generation Number:** Identifies the version of the inode **Retry on Failure:** Client simply retries request after a timeout **Idempotency:** Performing operations multiple times has the same effect of performing once **Cache Consistency:** Client can't get most recent version or reads out-of-date cache **Update Visibility:** Problem when the client gets the wrong version of a file from the server **Stale Cache:** When the client reads an out-of-date cache **Flush-on-Close:** Cache is always flushed after the application closes a file **Attribute Cache:** Local cache that updates contents only after a timeout **AFS:** Client side code is called Venus and server side code is called Vice **Scalability:** It is easily able to grow based on modularity **TestAuth:** Test whether a file has changed(used to validate cache entries) **Fetch:** Fetch the contents of the file **Store:** Store this file on the server **Whole-File Caching:** After open(), all contents are copied to local disk, read() and write() commands are performed only on the local copy, on close() file is flushed back to the server **Callback:** Server promises to inform client when file is modified, reduces server interactions **Heartbeat Protocol:** Client sends periodic message to server and expects a response | Deadlock | File Systems | VMs | Sec | Net | DS |

Gant Charts: Average Turn Around Time: (Process End Time - Arrival) + (...) / # of process
Average Response Time (Process Start Time - Arrival) + (...) / # of process

VPN: (Process VirAddress Read / Page Size) **Offset:** (Process VirAddress Read % Page Size)

PFN: Guest OS page table VPN # **MFN:** VMM Page Table PFN # **Machine Mem Address:** MFN * Page Size + Offset

Output: ABC BAC BC

Thread 1	Thread 2
sem_wait(S);	sem_post(S);
printf("A");	printf("B");
sem_post(S);	sem_wait(S);
	printf("C");

data bitmap	os.txt inode	data block 0
	1. read	
2. read 3. write		
		4. write
	5. write	

1. Read the inode of os.txt to get the length of the file
2. Because a new data block needs to be allocated for the file, read the data bitmap to find the location of a free data block
3. After finding a free data block, update the data bitmap by setting a bit to indicate the block has been taken
4. Write the data from char_buffer to the new data block
5. Update the inode to point to the new data block. Also, update other meta data such as the length of the file and time of last modification.

Guest OS Page Table

VPN	PFN	Valid
0	5	1
1	6	1
2	1	1
3	2	1
4	0	0
5	0	0
6	7	1

VMM Page Table

PFN	MFN	Valid
0	2	1
1	0	1
2	7	1
3	5	1
4	6	1
5	1	1
6	3	1

Some useful values

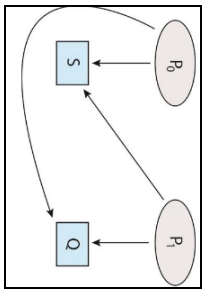
1KB = 1,024
2KB = 2,048
3KB = 3,072
4KB = 4,096
5KB = 5,120
6KB = 6,144
7KB = 7,168

VPN = 2050/1024 = 2 and offset = 2050 % 1024 = 2; PFN for VPN=2 is 1; MFN for PFN=1 is 0.

Machine's memory address: MFN*1024 + 2 = 2.

Consider a file system that uses Unix-style inodes. The system uses 8-KB blocks and 4-byte pointers. Multi-level indexing is used with an inode containing 8 direct pointers, 1 single indirect pointer, 1 double indirect pointer and 1 triple indirect pointer. What is the first file address that will be mapped to the double indirect pointers. You must show work, the answer can be left as an unsimplified arithmetic expression.

The 8 direct pointers point to addresses from 0 to 8*8K-1
The single indirect pointers point to addresses from 8*8K to 8*8K+(8K*8K/4)-1
The double indirect pointers to addresses starting at 8*8K+(8K*8K/4) = 64K + 8K*2048 = 16,842,752.



Explain how the distributed file system AFS uses callbacks. What would be an alternative approach? What are the advantages and disadvantages of callback over the alternative approach?

When a client fetches a file, the whole file is cached on the client. This means all future reads and write of the file are performed only from the cached version. Because the cached version can become out of date the client depends on the server notifying it of changes to the file by sending a notification callback message.
Callback has the advantage of fewer message from the client to the server because the client doesn't need to poll the server for file updates like in NFS.
Callback has the disadvantage that the server needs to be stateful, specifically it needs to remember all of the client that have files in their cache. This can be a problem if the server crashes and forgets its state.

FCFS:
Because FCFS doesn't use process preemption (different than resource preemption), it will not cause a hold and wait, therefore it will not cause a deadlock. For example, both processes in the chart below are to complete.

```
| P0 lock(s); lock(q); unlock(s); unlock(q); | P1 lock(q); lock(s); unlock(s); unlock(q); |
```

RR:
An RR scheduler uses process preemption, which can cause a deadlock, assuming there is no resource preemption. In the example below both P0 and P1 become deadlocked (enter a permanent wait state) at the statements highlighted.

```
| P0 lock(s); | P1 lock(q); | P0 lock(q); | P1 lock(s); |
```

1. Suppose that a computer can read or write a memory word in 10 nsec. Also suppose that when an interrupt occurs, all 32 CPU registers, plus the program counter are pushed onto the stack. What is the maximum number of interrupts pre second this machine can process.

To service an interrupt the CPU registers need to be written to memory and before the interrupt returns the registers need to be read back from memory. Assuming there is no caching 10 nsec is taken for each read or write of each word.

33 words * 10 nsec * 2 = 660 nsec

The maximum number of interrupts per second is: 1 / 660 nsec = 1.5 million

1. How many disk operations are needed to fetch the inode for the file `/var/log/boot.log`? Assume that the inode for the root directory is in memory, but nothing else along the path is in memory. Also assume that all directories fit in one disk block

Assuming the inode for the root directory is in memory, the reads will be:

1. **Root data block**
2. **Var inode block**
3. **Var data block**
4. **Log inode block**
5. **Log data block**
6. **boot.log inode block**

Therefore, it will be 6 read operations.

3. Consider a 320 GB SATA drive described in the previous figure. Suppose the workload is 10,000 reads to 10,000 sequential sectors on the outer-most tracks of the disk. How long will these 10,000 requests take (total) assuming the disk services requests in FIFO order?

Seek time and rotation time per request are the same as above.

Transfer time: 10,000 sectors * 512 bytes/sector * 1/(100 MB/s) = 51.2ms

So, 10,000 requests will take about 12ms + 5.56ms + 51.2ms = 68.76 ms (much faster that random access)

4. Linux provides a command-line utility rm that has the purpose of removing a file from a directory. When executing rm inside of the strace utility we can see that rm makes a system call to unlink(). Here is an example.

```
$ strace rm test.txt
...
unlinkat(AT_FDCWD, "test.txt", 0)      = 0
...
```

Explain why rm unlinks the file. How does unlinking result in the removal of the file? Why does Linux not have a system call such as delete() that rm could use to directly remove the file?

There may be more than one file name mapped to the inode.
If the file is deleted (inode is deleted), those other directory mappings would become invalid.
Unlinking removes the directory mapping.
Only when all files (references) to the inode are unlinked will the inode be deleted.

4. Some file systems allow disk storage to be allocated at different levels of granularity. For instance, a file system could allocate 4 KB of disk space as a single 4-KB block or as eight 512-byte blocks.

a. How could we take advantage of this flexibility to improve performance?

The advantage is less internal fragmentation will lead to less wasted space. Suppose a file only uses half of a block, then the other half would have been wasted space. With 512-byte subblocks the file will occupy exactly two sub-blocks and waste no space.

b. What modifications could be made to the bitmap free-space management scheme in order to support this feature?

There any many possible ideas. The most naive approach would be to make a bit map that is 4 times larger so that every bit represents one sub-block.

A more efficient approach might be to combine the ideas of a bitmap and free list. The bitmap would indicate if the entire block is free. When no free blocks remain a linked list indicates which sub-blocks are free.

2. Consider a 320 GB SATA drive described in the figure below. Suppose the workload is 10,000 reads to sectors randomly scattered across the disk. How long with these 10,000 requests take (total) assuming the disk services requests in FIFO order?

	Size
Form factor	2.5 inch
Capacity	320 GB
Spindel speed	5,400 RPM
Average seek time	12.0 ms
Maximum seek time	21 ms
Track-to-track seek time	2 ms
Transfer rate (surface to buffer)	850 Mbit/s (maximum)
Transfer rate (buffer to host)	3 Gbit/s
Buffer memory	8 MB

disk access time = seek time + rotation time + transfer time
Seek time:
- **Each request requires a seek from a random starting track to a random ending track.**
- **For seek time use the average of 12ms.**

Rotation time:
- **Once the disk head settles on the right track, it must wait for the desired sector to rotate under it.**
- **The average time it will take is one half of a rotation**
- **0.5 * 60s/5400RPM = 5.56ms**

Transfer time:
- **The disk's surface bandwidth is at least 100 MB/s, so transferring one sector (512 bytes) takes at most 5120ns (000512 ms).**

Total time:
- **Num requests * (seek + rotation + transfer)**
  = 10000 * (12ms + 5.56ms + 0.00512ms)
  = 10000 * 17.56512ms

5. Consider a file system that uses Unix-style inodes (index nodes) to represent files. The system uses 16-KB blocks and 8-byte pointers. Multi-level indexing is used with an inode containing 12 direct pointers, 1 single indirect pointer, 1 double indirect pointer and 1 triple indirect pointer.

a. What is the maximum size of a file that can be stored in the file system? Show calculations.

A block can hold 16KB / 8 bytes = 2,048 pointers
(12 + 2048 + 2048^2 + 2048^3) * 16KB

(12 + 2^11 + 2^22 + 2^33) * 2^14
8,594,130,956 * 16,384

(8,594,130,956 * 16,384) / 2^40 TB = 128 TB

b. For each of the following addresses describe how many blocks (including the inode and data block) need to be read to perform a read of the data at the address. Explain your Answers.

4,096
32,768
1,048,576
268,435,456

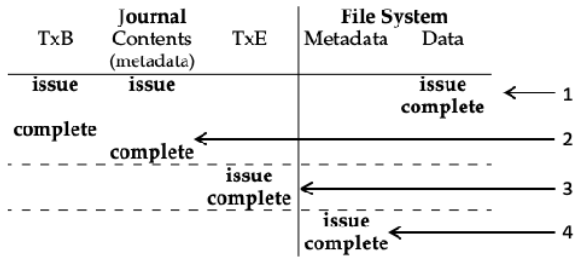
4,096 / 16,384 = block # 0
The block is less than 12, therefore it is directly pointed to by the inode, so 2 reads.

32,768 / 16,384 = block # 2
The block is less than 12, therefore it is directly pointed to by the inode, so 2 reads.

1,048,576 / 16,384 = block # 64
The block is >= 12 but less than 2060, therefore it is pointed to by a single indirect pointer, so 3 reads.

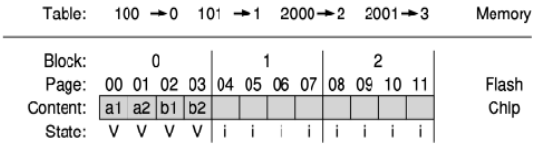
268,435,456 / 16,384 = block # 16,384
The block is >= 2,060 but less than 4,196,364, therefore it is pointed to by a double indirect pointer, so 4 reads.

1. The figure below shows a timeline (top to bottom) of when different actions required to write to a block to a file occur. Explain how the file system recovers from a failure at each of the 4 points indicated and say if any data is lost after the recovery.



1. No recovery as the data has not been logged to the journal or file system.
2. No recovery as the metadata has not been logged to the journal or file system.
3. No recovery as we don't know if the journal is complete (TxE has not been issued ye t).
4. Yes. We can replay the journal to recover the metadata in the file system. Data is alr eady in the file system since it is done before the issue of TxE.

a. Suppose the file system issues a write command to update the contents of virtual page 2000 with data "c1". Assume the current state of the SSD drive is shown in the figure below. Draw the most likely state of the SSD after the write has been completed.

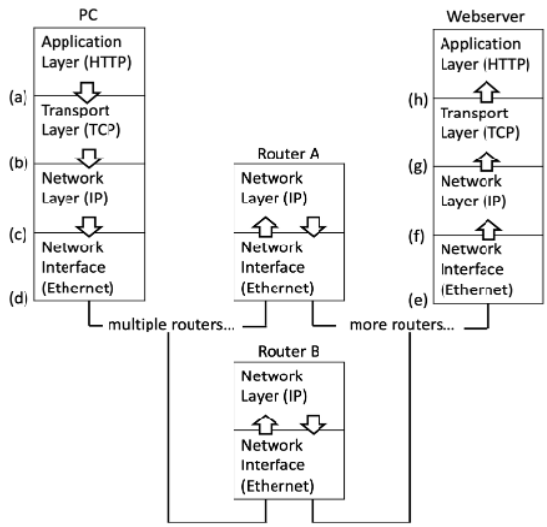


b. Now suppose, following the previous write, the file system deletes a file and removes virtual pages 100 and 101. Assume the FTL decides to garbage collect block 0. Draw the most likely state of the SSD after the garbage collection has been completed.

a.
table: 100->0, 101->1, 2000->4, 2001->3
| a1 | a2 | b1 | b2 | c1 |

b.
table: 2000->4, 2001->3
| e | e | e | e | c1 | b2 |

1. The diagram shows the scenario of a PC making a request of a webserver. The Internet is represented between the PC and the webserver. There are two possible paths for packets to take, one passes through router A and the other through router B. Assume that it is equally likely for packets to take either path. Use the example to answer the following questions.



a) Explain the difference between the transport layer and the network layer.

Both are part of the network stack, and the transport layer relies on the network layer to complete its tasks. The transport layer provides communication between processes (usually on different machines), the processes are identified by port numbers, and it is expected to be resilient to packet loss or out of order packets. The network layer provides communication between hosts (e.g., machines), the hosts are identified by IP addresses, and no attempt is made to resent, or reorder lost or out of order packets.

b) Why do the routers not have a transport layer?

Routers participate at the network layer and below, they look at each packet's IP address and forward the packet to the next appropriate node, they do not try to check for lost or out of order packets. Adding transport layer mechanisms at the router level would make routing slow, expensive, and inflexible.

c) For efficient sharing of the network infrastructure, TCP breaks long application layer messages into small packets (typically less than 1500 bytes). Explain why the packets arriving at the webserver can be out of order.

The hosts (PC and webserver) have no control over the route packets will take over the network, some packets might go to router A and others to router B. Communication over the network is asynchronous, some packets may be lost, some packets may take a longer/slower route, so packets from router A and router B could arrive at the webserver in any order.

d) The arrows in the diagram show an example TCP communication from the PC to the Webserver, the layer boundaries have been labeled (a) to (h). List all boundaries (a to h) where packets might **arrive** out of order.

On the PC, the application must provide the correctly ordered message to the transport layer, where there would be no reason for the transport layer, network layer or network interface to change the order of packets. Once the packets reach the network there is not guarantee over which route packets will take or how long a particular route will take. The transport layer (TCP) on the webserver maintains a receive window, it collects packets in the window and uses acknowledgements to have lost packets resent. Only a complete (contiguous) window of packets is forwarded to the application layer.

2. What are the three parts of a socket? Which layer of the 7-layer OSI model does each part deal with?

Protocol – transport layer
Port – transport layer
IP Address – network layer

What are the advantages and disadvantages of TCP and UDP? Give examples of applications where each might be preferred.

TCP is reliable as it guarantees the delivery of data to the destination router. A disadvantage of TCP is that it is slower and more expensive because it relies on establishing and maintaining a connection which requires extra packets being sent.

UPD is a connectionless protocol, packets are sent without any guarantee of delivery. The advantage is no overhead – no handshake packets sent to establish a connection, no acknowledgement packets sent for every data packet, etc. The disadvantage is that packets can be lost or delivered out of order, it is up the application to deal with that.

TCP is common for file transfer (e.g., FTP), email (e.g., SMTP), and web (e.g., HTTP). UDP is common for multimedia applications (e.g., VoIP) and client-server communication that doesn't need to send large messages (e.g., DNS).

4. Calculate the total time required to transfer a 1000-KB file in the following cases, assuming an RTT of 50 ms, a packet size of 1 KB data:

(a) The bandwidth is 1.5 Mbps, and data packets can be sent continuously.

Bandwidth = 1.5 Mbps = 0.1875 MB/s

Packet size = 1 KB = 1,024 B

Transmit time per packet = size/bandwidth = 1,024B / (0.1875B/s *1024*1024) = 5.200ms

Transmit time for 1,000 packets = 5,200ms

Propagation + queuing time = RTT / 2 = 25ms

Transfer time (latency) = 5,200ms + 25ms = 5,225ms

(b) The bandwidth is 1.5 Mbps, but after we finish sending each data packet we must wait one RTT (for data receive and acknowledgement) before sending the next.

(5.200ms + 50ms) * 1,000 = 55,500ms

1. Suppose a distributed system is using acknowledgement-based communication:
   a. How does the sender distinguish between the message being lost, the receiver being not available, or the acknowledgement being lost?

      The sender has no way to know.

   b. Explain, how the sender deals with a lost acknowledgement.

      After a timeout, it resends the original message. Messages have a sequence number attached so the receiver will know if it receives the same message twice.

   c. Explain, how the sender deals with an acknowledgement that is significantly delayed.

      From the perspective of the sender this is the same as a lost acknowledgement. If the acknowledgement does not arrive before the timeout, it resends the original message.

2. In most operating systems file access permissions are checked only when the file is opened and not with each read or write. Thus if you open a file successfully for both reading and writing, obtaining file descriptor *f*, and if you subsequently change the file's access permissions to read-only, you can still write to the file using *f*.

a. Explain why supporting this feature is difficult in distributed file systems whose servers do not hold open-file state, such as NFS.

A stateless server such as NFS does not remember what it told a client in a previous response. Because of this, for security, the NFS server much check file permission on every request.

b. How might you "approximate" this feature so that it's supported on NFS for what's probably the most common case: the file's owner is accessing the file?

If the files owner is always given both read and write permission at the server (which seems reasonable given that it is the owner), then additional permission restrictions can be supported at the client.

3. Suppose we wish to add to basic NFS a remote procedure for appending data to the end of a file.
a. Why does NFS not already have an APPEND procedure?
Append is not a idempotent operation, so it was omitted from the client-server protocol.
b. Explain why such a remote procedure is needed: why can't its effect be obtained through the use of the GETATTR procedure (which, among other things, returns the size of a file) and the WRITE procedure?
The approach of using GETATTR followed by WRITE is not an atomic operation. This could result in a race condition between client where they overwrite each other appends.
c. Assuming a well behaved network and a server that never crashes, will there be any problems if an append request or response is lost? Explain.
Yes, the client will assume the request failed and resend to the append. This can potentially result in multiple appends to the file

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)		read write	read	read write	read write	read	read	write		
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

How to create file /foo/bar?

1. Read "root inode" (i.e., inode for "/") to get location of "/".
2. Read data block of "/" (i.e., "root data") to get inode number of "/foo".
3. Read "foo inode" to get location of "/foo".
4. Read "/foo" (i.e., "foo data"); read inode bitmap to find an empty inode, denoted as x, and update the bitmap; add a pair ("bar", x) to "foo data".
5. Read "bar inode" (i.e., inode with number x) and initialize it.
6. Update last access time at "foo inode".

How to write to new file /foo/bar?

1. Read "bar inode".
2. Read data bitmap to find an empty data block x and update the bitmap.
3. Write to data block x (i.e., "bar data [0]").
4. Add block x to "bar inode".
5. Update the last access time at "bar inode"
6. Read "bar inode".
7. Read data bitmap to find an empty data block y and update the bitmap.
8. Write to data block y (i.e., "bar data [1]").
9. Add block y to "bar inode".
10. … …

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			read	read	read	read	read			
read()					read write			read		
read()					read write				read	
read()					read write					read

How to read from the file /foo/bar? First, fd=open("/foo/bar").

1. inode number for root directory is well known; read "root inode" and get the location for "root data".
2. Read "root data" to get inode number for "/foo".
3. Read "foo inode" to get location of "foo data".
4. Read "foo data" to get inode number for "/foo/bar".
5. Read "bar inode" and bring the metadata of /foo/bar to main memory.

How to read from the file /foo/bar? Second, read(fd,...).

1. Read "bar inode" to get the location of the first data block of /foo/bar, i.e., "bar data [0]".
2. Read "bar data [0]".
3. Update last access time at "bar inode".
4. Read "bar inode" to get the location of the second block of /foo/bar, i.e., "bar data[1]".
5. Read "bar data [1]".
6. Update last access time at "bar inode".