**Disclaimer:** This study guide attempts to touch upon the most important topics that may be covered on the exam but does not claim to necessarily cover everything that one needs to know for the exam.

**Distributed transactions**

A key facet of a transaction is that it has to be **atomic** – all results have to be made permanent (commit) and appear as an indivisible action. If a transaction cannot complete, it must **abort**, reverting the state of the system to that before it ran. If several transactions run concurrently, the end result must be the same as if they ran in **some** (any) serial order.

A **write-ahead log** (or transaction log) is crucial for **rollback** (reverting to the previous state when aborting a transaction). It is also crucial for maintaining state in a stable place in case the system should die; it allows the system to recover from where it was in the transaction.

The **two-phase commit protocol** uses a coordinator send a request (“*can you commit?*”) to every member of the group (reliably; retransmitting as often as needed until all replies are received). Phase 1 is complete when every member of the group responds. If the coordinator gets even a single **abort** response, it will have to tell all group members to abort the entire transaction. Otherwise, it can tell everybody to commit it. In phase 2, the coordinator sends the commit or abort order and waits for a response from everyone. The write-ahead log is crucial here. For example, if a machine sent the coordinator a “commit” response for phase 1 and then died, it must be able to reboot and reconstruct the transaction state from the log; it cannot change its mind.

**Distributed deadlocks**

**Deadlock** is the condition when there is a circular dependency on resources allocation (for example, A needs a resource from B but B needs a resource that A is holding. A will not release the resource until it gets the resource it needs from B). Four conditions must be satisfied for a deadlock to be present: mutual exclusion, hold-and-wait, non-preemption, and circular wait.

Distributed systems can deadlock when the same set of conditions apply. Unfortunately it can be more difficult to detect deadlocks in this environment. A straightforward solution to deadlock detection is to use a central server to keep track of the resource utilization graph. Whenever a process requests or releases a resource, it sends a message to the server. If the server detects a cycle in the graph then a deadlock has been detected. This is **centralized deadlock detection**. One problem with this system is when request and release messages arrive out of order. In some cases, the server may detect a deadlock because a request message was received before a release message. This is known as a **false deadlock**.
An example of a distributed algorithm to detect deadlocks is the Chandy-Misra-Haas algorithm. Before a process requests a resource, it sends a probe message to the resource holder. The resource holder forwards the probe message to all of the processes holding its resources, and so on. If the originating process receives this probe message it knows that requesting the resource will result in a cycle, and hence a deadlock condition.

A technique for deadlock prevention is to force a certain order of resource allocation. The wait-die algorithm allows a cycle where older processes can wait for resources held by younger processes. However, if a younger process wants a resource held by an older one then the young process would kill itself (in a transactional system it would restart later and hope that the conditions that caused deadlock will no longer exist. The wound-wait algorithm also imposes resource utilization ordered by transaction time (process start) to prevent cycles (hence, deadlocks). If an older process wants a resource held by a younger process then the older process kills the younger process.

**Logical clocks**

Lamport clocks allow one to assign sequence numbers (“timestamps”) to messages and other events so that all cooperating processes can agree on the order of events. There is no assumption of a central time source and no concept of total ordering (when events took place). The central concept with logical clocks is the happened-before relation: $a \rightarrow b$ represents that event $a$ occurred before event $b$. This order is imposed upon consecutive events at a process and also upon a message being sent before it is received at another process. Beyond that, we can use the transitive property of the relationship to determine causality: if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. If there is no causal relationship between two events (e.g., they occur on different processes that do not exchange messages), the events are concurrent.

Lamport’s algorithm states that every event is timestamped (assigned a sequence number) and each message carries a timestamp of the sender’s clock (sequence number). A message comprises two events: the event of sending the message and the event of receiving the message. The clock is a process-wide timer (e.g., a global variable) and is always incremented before each event. When a message arrives, if the receiver’s clock is less than or equal to the message’s timestamp, the clock is set to the message timestamp + 1. This ensures that the timestamp marking the event of a received message will always be greater than the timestamp of that sent message.

One problem with Lamport timestamps is that multiple events on different processes may all be tagged with the same timestamp. We can force each timestamp to be unique by suffixing it with a globally unique process number. While these new timestamps will not relate to real time ordering, they will be unique numbers that can be used for consistent comparisons of timestamps (e.g., if we need to make a decision on who gets to access a resource based on a timestamp).

A second deficiency with Lamport timestamps is that, by looking at timestamps, one cannot determine whether there is a causal relationship between two events. For example, just because
event \(a\) has a timestamp of 5 and event \(b\) has a timestamp of 6, it does not imply that event \(a\) happened before event \(b\). A way to create timestamps that allow us to discern causal relationships is to use a \textbf{vector clock}. A vector clock is no longer a single value but rather a vector of numbers, each element corresponding to a process. Before affixing a vector timestamp to an event, a process increments the element of its local vector that corresponds to its position in the event (for example, process 0 increments element 0 of its vector; process 1 increments element 1 of its vector). When a process receives a message, it sets the vector of the event to one that contains the higher of two values when doing and element-by-element comparison of the original event’s vector and the vector received in the message. This becomes the new per-process vector from which future events on that process will be timestamped. For example, in an environment of four processors, P1 will "own" the second element of the vector. If one event on P1 is \((2, 4, 0, 1)\) then the next event will be \((2, 5, 0, 1)\). If the event after that is the receipt of a message with a timestamp of \((3, 2, 9, 8)\) then the timestamp will be created by setting an element-by-element maximum of \((2, 6, 0, 1)\) and \((3, 2, 9, 8)\), which is \((3, 6, 9, 8)\).

Two events are concurrent if one vector timestamp is neither greater than or equal nor less than or equal to the other element when doing an element-by-element comparison. For example, events \((2, 4, 6, 8)\) and \((3, 4, 7, 9)\) are not concurrent (i.e., they are causally related) because every element of the first vector is less than or equal to the corresponding element of the second vector. The vectors \((2, 4, 6, 8)\) and \((1, 5, 4, 9)\) represent concurrent events. Neither vector is less than or equal to the other. For instance, \(2 \geq 1\) (first element of the first vector is greater than the first element of the second vector) but \(4 \leq 5\) (second element of the first vector is less than the second element of the second vector).

**Clock synchronization**

No two clocks tick in perfect synchrony with each other. The difference between two clocks at any given instant is the clock \textbf{skew}. The \textit{rate} at which the clocks are drifting is the clock \textbf{drift}. A \textit{linear compensating function} adjusts the rate at which time is measured on a computer (e.g., number of ticks that make up a second).

**Cristian’s algorithm** sets the time on a client to the time returned by the server plus an offset that is one half of the transit time between the request and response messages: \(T_{client} = T_{server} + \frac{1}{2}(T_{received} - T_{sent})\). It also allows one to compute the maximum error of the new time stamp. The error is \(\pm (\text{round-trip time} - \text{best-case round-trip time})/2\). Errors are additive. If you incur an error of \(\pm 50\) msec and the server’s clock source has an error of \(\pm 80\) msec, your clock’s error is now \(\pm 130\) msec.

The **Berkeley algorithm** does not assume the presence of a server with an accurate time (i.e., one that keeps track of UTC time). Instead, one system is chosen to act as a coordinator. It requests the time from all systems in the group (including itself) and computes a \textbf{fault-tolerant average} (an arithmetic average, dismissing systems whose time values differ by more than a certain amount). It then sends each machine an offset by which to adjust its clock.
The **Network Time Protocol**, NTP, was created to allow a large set of machines to synchronize their clocks. A set of machines acts as time servers – this collection of machines is the **synchronization subnet**. The subnet is hierarchical, with the time server's **stratum** defined as the number of hops from a machine that synchronizes from a direct time source. Machines that are directly connected to a time source (e.g., a GPS receiver) are at stratum 0. Machines that synchronize from a system at stratum 0 are at stratum one, and so on. The **Simple Network Time Protocol**, SNTP, is a restricted form of NTP that does not support peer-to-peer synchronization of time servers (maintaining state on drift and synchronization time). It is essentially the same as Cristian's algorithm.

**Group communication**

Point-to-point communication is known as **unicast**. This is what we generally use to communicate a single client and server. There are other modes of communication. **Broadcast** is the sending of data to every node on the network. **Anycast** is point-to-point communication (as in unicast) but the receiver is the nearest one of receivers with specific capabilities (for example, IPv6 uses this to allow a host to update the routing table of the nearest host). Finally, there’s **group** communication, known as **multicast**. This is point-to-multipoint communication. A message gets sent to everyone in the group. One implementation of multicast is known as **netcast**. This simulates hardware multicast by invoking multiple unicasts, one to each recipient. There are two considerations in group communication (multicast): reliability and message ordering.

**Reliability**

An **atomic multicast** requires that a message must reach all group members (if one node cannot get the message, no others can process it). This multicast must survive machines going down. Because of this, it requires the most overhead in implementation, often employing persistent logs.

A **reliable multicast** is a best-effort multicast. The sender sends a message and waits for an acknowledgement. If it doesn’t receive the acknowledgement in time, it will retransmit the message. Eventually, after a longer interval of time, it will give up and assume the receiver is dead.

An **unreliable multicast** doesn’t wait for acknowledgements and will generally use whatever underlying multicast mechanism is provided.

**Message ordering**

The issue in message ordering is that multiple nodes can be sending messages to the entire group at the same time. Will each node receive all the messages in the exact same order? Will each node receive all the messages in the order they were sent?
Global time ordering requires that all messages arrive in the exact order they were sent: if node A sent a message 5 nanoseconds before node B did then all nodes should receive the message from node A first. This is impossible to implement (clocks can’t be that perfectly synchronized and the chance of a tie always arises; also, networks will not be able to guarantee this kind of ordering if routing is involved). A more practical approach is total ordering. This requires that all messages arrive in the same order at all nodes. This can be easily achieved by providing a mechanism for obtaining a totally sequenced message ID – e.g., from a sequence number server.

Causal ordering is the ordering you would get by attaching Lamport time stamps to messages. The ordering of unrelated (concurrent) messages is insignificant, but causally related messages will be in the proper order. Sync ordering requires a special type of message – a sync message. When this issued, any messages that were already sent have to be processed (and acknowledged to the senders). The sync assures that any messages sent before the sync will be processed before any messages after the sync (globally – for all nodes).

Finally, an unordered multicast doesn’t impose any message ordering among messages from other nodes. It may choose to impose FIFO (first in, first out) ordering on messages from a particular source (e.g. as TCP does).

**IP multicast**

An IP multicast address (also known as a class D address) contains a 28-bit multicast address. A host may join this address and receive messages addressed to that multicast ID. Within a LAN, an IP class D address is mapped onto an Ethernet multicast address by copying the least-significant 23 bits of the address onto an Ethernet multicast address. The Internet Group Management Protocol (IGMP) manages membership beyond a LAN. A node broadcasts a multicast join message to join a specific group. A multicast-aware router will pick this message up and sent the join message to other connected networks (this way a spanning tree is built, ultimately forwarding the request to the source). Periodically, each router sends a query message for each multicast group. If any node (or router) is still interested in the group, it must send a join message. If no join messages are received, then the router will stop responding to join messages and the LAN will no longer receive packets for that group.

**Mutual exclusion**

The centralized algorithm is a server that accepts REQUEST messages for a resource (e.g., critical section). If nobody is using the resource, it responds with a GRANT message. If somebody is using the resource, it doesn’t respond. When a client is done with a resource, it sends the server a RELEASE message. The server then sends a GRANT message to the next client in the queue (if there is one).

The token ring algorithm creates a logical communication ring among the nodes in the group. A message, called a token, is created for each resource. This token is passed from node to node along the ring. If a node receives a token and does not need to access that resource, it simply
passes the token to the next node. Otherwise, it will hold on to the token until it is done with the resource.

The **Ricart & Agrawala** algorithm was an early demonstration that a truly distributed algorithm is possible. A node that wants to enter a critical section sends a request to *all* other nodes in the group and waits for all of them to respond. If another node is currently using the critical section, that node delays its response until it is done. If node $A$ and node $B$ sent out requests concurrently (i.e., two systems want the same resource concurrently), each system compares the timestamp of that request with that of its own request. If node $A$ received a request from node $B$ that is older (a lower timestamp), then node $A$ will give node $B$ priority to access the resource by sending a response to node $B$. Otherwise, if node $A$ has the earlier timestamp, it will queue the request from $B$ and continue to wait for all acknowledgements, sending a response to $B$ (and any other nodes who wanted the resource) only when it is done using the resource. The key point is that nodes $A$ and $B$ will make the same comparison and exactly one will hold back on sending the response.

**Lamport**’s algorithm, like the Ricart & Agrawala algorithm, is also based on reliable multicast. This algorithm has a node send a timestamped request to all nodes in the group, including itself. Every message is acknowledged immediately. A process decides whether it can access the resource by checking whether its own request is the earliest one in the queue of all requests that it has received. If so, it accesses the resource and, when done, sends a *release* to all members (including itself). The receipt of a *release* message causes each process to remove the process from the ordered queue. If a process now finds itself as the earliest process in the queue, it knows that it is now its turn to access the resource.

**Election algorithms**

The **bully algorithm** selects the largest process ID as the coordinator. If a process detects a dead coordinator, it sends an *ELECTION* message to all processes with a higher ID number and waits for any replies. If it gets none within a certain time, it announces itself as a coordinator. When a process receives an *ELECTION* message it immediately sends a response back to the requestor (so the requestor won’t become the coordinator) and holds an election to see if there are any higher-numbered processes that will respond.

The **ring algorithm** requires a logical communication ring among the nodes in the group. When a process detects a non-responding coordinator, it creates an *ELECTION* message with its own process ID and sends it to the next node on the ring. If the node is dead, it sends it to the following node (skipping dead nodes until it finds one that can receive the message). When a node receives an *ELECTION* message, it adds its own process ID to the body and sends that message out to the neighboring node. When the election message circulates back to the original sender, the sender gets the list of active nodes from the body and chooses a coordinator from among them (e.g., highest ID).

---

1 The timestamp only has to be a unique Lamport timestamp (e.g., `<timestamp, process ID>`). This ensures that every process sends a unique number for a timestamp.
One problem with election algorithms is when a network gets segmented: one set of nodes is separated from another. In this case, each segment may elect its own coordinator and problems can arise. To combat this, a redundant network is needed (or some alternate communication mechanism to enquire about the state of nodes).

**Distributed shared memory**

Distributed shared memory (DSM) is a mechanism to allow processes on different machines to have the illusion of sharing memory (i.e., being able read data from memory that another process wrote to memory). Transparency is generally achieved through the system’s memory management unit (MMU). When a page fault occurs, the page fault handler invokes the DSM algorithm, which can fetch the page from another server, map it into the process’ address space and restart the instruction. A directory is the server, or set of servers that allows a node to find out where the needed page lives (and possibly where cached copies reside).

A well-defined memory consistency models is key to DSM is the algorithms must enforce the model. **Sequential consistency** is what we generally expect from a memory system. A read always returns the result of the last write for a given instruction stream. For multiple streams (concurrent processes), any interleaving is acceptable as long as accesses within each stream are in sequential order. To achieve sequential consistency, each write operation must invalidate or update all cached copies before the write completes. Since writes have to be visible in the same order by all processes, a read cannot take place until the write is acknowledged.

Presenting sequential consistency is an expensive proposition, since considerable network activity has to take place for each write operation. To enhance performance, weaker consistency models can be employed.

A **weak** (or **relaxed**) consistency model requires that memory be consistent only at specific synchronization events. For example, a **sync variable** is an operation that will force all local operations to be propagated out and remote operations on memory to be brought in. This way, memory is made consistent explicitly on a sync rather than for each operation. **Release consistency** allows us to break up the operations of sending out changes and receiving updates into two parts: an **acquire** phase and a **release** phase. An acquire indicates that the processor is about to perform operations on the memory and needs to get any updates from other processors. A release indicates that the processor is done with its operations on the memory. Any of the processor’s writes now have to be made visible to a processor doing a subsequent acquire. There are two variants of release consistency: **eager** and **lazy**. The eager protocol ensures that all copies of pages on other nodes are updated with the contents of the newly-modified pages when a processor performs a release operation. In fact, page invalidations may be sent during program execution and the release operation itself is a blocking on that ensures that all invalidations have been acknowledged. The lazy protocol does not bother to update or invalidate existing copies of the pages upon a release (on the assumption that those nodes with cached copies of the page might never even access the page again). Instead, page invalidations are propagated at acquire time.
Finally, **entry consistency** allows the *acquire* and *release* operations to take place on regions of memory (such as individual variables or data structures) rather than all of shared memory. To achieve entry consistency, one must employ smart compilers or control the consistency explicitly since the system’s MMU cannot protect these boundaries.

### Cryptography

A **symmetric encryption algorithm** uses the same secret key for encryption and decryption. A **public key algorithm** uses one key for encryption and another for decryption. One of these keys is kept private (known only to the creator) and is known as the **private key**. The corresponding key is generally made visible to others and is known as the **public key**. Anything encrypted with the private key can only be decrypted with the public key. This is the basis for digital signatures. Anything that is encrypted with a public key can be encrypted only with the corresponding private key. This is the basis for authentication and covert communication.

A **one-way function** is one that can be computed relatively easily in one direction but there is no direct way of computing the inverse function. One-way functions are crucial in a number of cryptographic algorithms, including digital signatures, Diffie-Hellman key exchange, and RSA public key cryptography. For Diffie-Hellman and RSA keys, they ensure that someone cannot generate the corresponding private key when presented with a public key. A particularly useful form of a one-way function is the **hash function**. This is a one-way function whose output is always a fixed number of bits for any input. The hash function is the basis for message authentication codes and digital signatures.

The earliest form of cryptography was the **substitution cipher**. In this cipher, each character of plaintext is substituted with a character of ciphertext based on a substitution alphabet (a lookup table). The simplest of these is the **Caesar** cipher, where a plaintext character is replaced with a character that is *n* positions away in the alphabet. The key is the number *n*. Substitution ciphers are vulnerable to **frequency analysis attacks**, in which an analyst analyzes letter frequencies in ciphertext and substitutes characters with those that occur with the same frequency in natural language text (e.g., if “x” occurs 12% of the time, it’s likely to really be an “e” since “e” occurs in English text approximately 12% of the time).

#### Polyalphabetic substitution ciphers

Polyalphabetic ciphers were designed to increase resiliency against frequency analysis attacks. Instead of using a single plaintext to ciphertext mapping for the entire message, the substitution alphabet may change periodically. In the Alberti cipher (essentially a secret decoder ring), the substitution alphabet changes every *n* characters as the ring is rotated one position every *n* characters. In the Vigenère cipher, the next character of the key determines which Caesar cipher will be used for the next character of plaintext. A **rotor machine** is an electromechanical device that uses a number of rotors, each of which implements a randomly-structured substitution cipher. The rotors rotate with each character in the style of an odometer (after a complete rotation of one rotor, the next rotor advances one position). This allows for a huge
number of substitution alphabets to be employed before they start repeating (the rotors all reach their starting position).

**Transposition ciphers**

Instead of substituting one character of plaintext for a character of ciphertext, a transposition cipher scrambles the position of the plaintext characters. Decryption is the knowledge of how to unscramble them.

**One-time Pads**

The one-time pad is the only provably secure cipher. It requires a completely random key that is as long as the plaintext. Each character of plaintext is permuted by a character of ciphertext (e.g., add the characters modulo the size of the alphabet or exclusive-or binary data). The reason this cryptosystem is not particularly useful is because the key has to be as long as the message, so transporting the key securely becomes a problem. The challenge of sending a message securely is now replaced with the challenge of sending the key securely. Moreover, the recipient needs to have a key that is as long as all the messages that a sender might send and the sender’s and recipient’s position in the key (pad) must be synchronized at all times. Error recovery from unsynchronized keys is not possible. Because the key is as long as a message, any ciphertext can be converted to any plaintext, depending on the key that is used.

**Secure communication**

To communicate securely using a symmetric cipher, both parties need to have a shared secret key. Alice will encode a message to Bob using the key and Bob will use the key to decode the message. If Alice wants to communicate with Charles, she and Charles will also need a secret key. The fact that every pair of entities will need a secret key leads to a phenomenon known as key explosion. The biggest problem with symmetric cryptography is dealing with key distribution: how can Alice and Bob establish a key so they can communicate securely? The Diffie-Hellman exponential key exchange algorithm allows one to do this. Each party will generate a private key and a public key (these are not encryption keys; they’re just numbers). Alice can compute a common key using her private key and Bob’s public key. Bob can compute the same common key by using his private key and Alice’s public key.

Using public key cryptography, if Alice encrypts a message with Bob’s public key, Bob will be the only one who can decrypt it since doing so will require Bob’s private key. Likewise, Bob can encrypt messages with Alice’s public key, knowing that only Alice will be able to decrypt them with her private key.

**Session keys**

A session key is a random key that is generated for encryption during one communication session. It is useful because if the key is ever compromised, no lasting information is obtained: future communication sessions will use different keys. A hybrid cryptosystem uses public key
cryptography to send a session key securely. The originator generates a random session key and encrypts it with the recipient's public key. The recipient decrypts the message with the corresponding private key to extract the session key. After that, symmetric cryptography is used for communication, with messages encrypted with the session key. This has the advantages of higher performance (public key cryptography is a lot slower than symmetric cryptography), ease of communicating with multiple parties (just encrypt the session key with the public keys of each of the recipients), and allows the bulk of data to be encrypted with ever-changing session keys instead of the hardly-ever-changing public keys.

**Digital signatures**

Digital signatures employing symmetric cryptography must turn to a trusted third party. Messages are encrypted with the owner's key and sent to the third party who decrypts the contents and re-encrypts them for the recipient. The trusted third party avoids the problem of key explosion where every pair of communicating parties must have a secret key.

With public key cryptography, a digital signature is simply the act of encrypting a hash of a message with the creator's private key. Anyone who has the public key can decrypt the hash and thus validate it against the message. Other parties cannot recreate the signature since they do not have the private key even though they can create the hash.