Chapter 49: Sun's NEtwork File System (NFS)

- One of the first uses of distributed client/server computing was in the realm of distributed file systems
    - number of client machines and one server (or a few) -> server stores the data on its disks, and clients request data through well-formed protocol messages

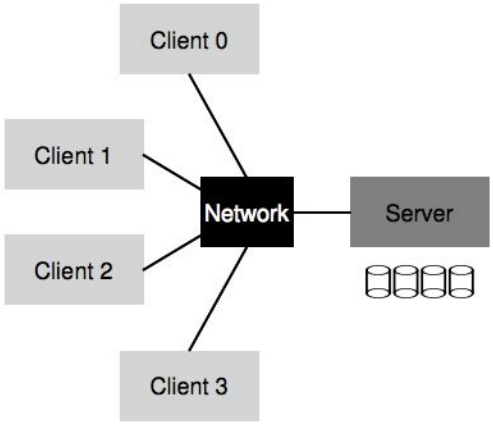


Figure 49.1: **A Generic Client/Server System**

-
- Why don't we just let the clients use the disks instead of the server?
    - The above setup allows for easy **sharing of data across clients**
        - If you access a file on different machines, you'll have the same view of the file system on different machines
    - **Centralized Administration**
        - Backing up files can be done from the few server machines instead of from the multitude of clients
    - **Security**
        - Having all servers in a locked machine room prevents certain types of problems from existing

49.1: A Basic Distributed File System

- A client application issues system calls to the client-side file system (i.e. open(), read(), write, close(), mkdir(), etc) in order to access files and directories stored on the server
    - Role of the client-side system is to execute the actions needed to service those system calls
    - read() from the client-side may send a message to the server-side file system to read a particular block -> sends message back with the requested data
    - Client side file system will them copy the data into the user buffer supplied to the read() system call, completing the request
    - Subsequent call to read of the same block may be cached in client memory or on the client's disk

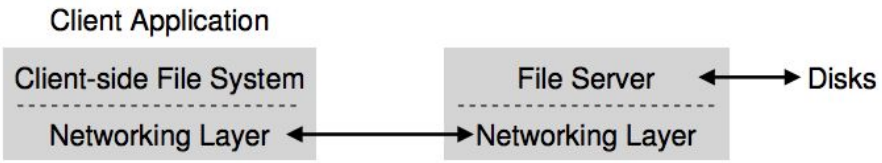


Figure 49.2: **Distributed File System Architecture**

-

49.2: On to NFS

- Sun developed an open protocol which simply specified the exact message formats that clients and servers would use to communicate
- Different groups could develop their own NFS servers and thus compete in an NFS marketplace while preserving interoperability

49.3: Focus: Simple and Fast Server Crash Recovery

- Any minute that the server is down/unavailable, all the client machines will become unhappy and unproductive

49.4: Key to Fast Crash Recovery: Statelessness

- The server, by design, does not keep track of anything about what is happening at each client. The server does not know which clients are caching which blocks or which files are currently open at each client
    - Rather, the protocol is designed to deliver in each protocol request, all the information that is needed in order to complete the request
- Example: open() system call
    - Given a pathname, open() returns a file descriptor -> can be used on subsequent read and write calls to access various file blocks
    - When we ask the server to read a few blocks from a certain fd, then the file descriptor is a piece of **shared state between the client and the server**
        - shared state complicates crash recovery -> what happens if the server crashes during a read?
        - To handle this, the client and server would need to engage in some sort of recovery protocol, where the client would make sure to keep enough info in its memory
    - Can also be difficult on the client side as well

49.5: The NFSv2 Protocol

- How do we define a stateless file protocol?
    - stateful calls such as open() can't be a part of the discussion, but we still want the client application to be able to make those calls
- **File handle:** Used to uniquely describe the file or directory a particular operation is going to operate upon -> many protocol requests include a file handle

    - File handle has three important components: volume identifier, inode number, and generation number
    - **Volume Identifier:** informs the server which file system the request refers to
    - **Inode number:** tells the server which file within that partition the request is accessing
    - **Generation number:** needed when reusing an inode number -> incrementing it whenever an inode number is reused, the server ensures the client with an old file handle can't accidentally access the newly-allocated file
- Important components of NFS protocol
    - LOOKUP: used to obtain a file handle, which is then subsequently used to access file data
        - passes a directory file handle and name of the file to lookup
    - Attributes are just the metadata that the file system tracks about each file, including fields such as file creation time, modification time, access time, size, ownership permissions, etc.

49.6: From Protocol to Distributed File System

- Client-side file system tracks open files, and generally translates application requests into the relevant set of protocol messages. Server simply responds to protocol messages, each of which contains all information needed to complete the request

Client	Server
fd = open("/foo", ...);	
Send LOOKUP (rootdir FH, "foo")	
	Receive LOOKUP request look for "foo" in root dir return foo's FH + attributes
Receive LOOKUP reply allocate file desc in open file table store foo's FH in table store current file position (0) return file descriptor to application	
read(fd, buffer, MAX);	
Index into open file table with fd get NFS file handle (FH) use current file position as offset	
Send READ (FH, offset=0, count=MAX)	
	Receive READ request use FH to get volume/inode num read inode from disk (or cache) compute block location (using offset) read data from disk (or cache) return data to client
Receive READ reply update file position (+bytes read) set current file position = MAX return data/error code to app	
read(fd, buffer, MAX);	
Same except offset=MAX and set current file position = 2*MAX	
read(fd, buffer, MAX);	
Same except offset=2*MAX and set current file position = 3*MAX	
close(fd);	
Just need to clean up local structures Free descriptor "fd" in open file table (No need to talk to server)	

Figure 49.5: **Reading A File: Client-side And File Server Actions**

-

- Client tracks all relevant state for the file access, including the mapping of the integer file descriptor to an NFS file handle, as well as the current file pointer
    - allows the client to turn each read request into a properly formatted read protocol message which tells the server exactly which bytes from which file to read
    - Upon a successful read, the client updates the current file position -> subsequent reads are issued with the same file handle but a different offset
- Second, notice where server interactions occur
    - When the file is opened for the first time, the client-side file system sends a LOOKUP request message. -> if a long pathname has to be traversed, then it will send more LOOKUP messages
- Third, notice how each server request has the all information needed to complete the request in its entirety
    - ensures that the server does not need state to be able to respond to the request
- Tip: Idempotency: much easier to handle failure of the operation when an operation can be issued more than once -> we can simply retry

49.7: Handling Server Failure with Idempotent Operations

- When a client sends a message to the server, it sometimes does not receive a reply
    - messages may be dropped by the network, etc.
    - server could have crashed, and is not responding to messages at the moment
- Client handles all of these failures in a single, uniform, and elegant way -> simply retries the request
    - After sending the request, the client sets a timer to go off after a specified time period. If a reply is received before the timer goes off, the timer is cancelled
    - If the timer goes off before any response is received, the client assumes the request has not been processed and resends it
- NFS requests are idempotent
    - An operation is called **idempotent** when the effect of performing the operation multiple times is equivalent to the effect of performing the operation a single time
        - ex: if you store a value to memory three times, it is the same as doing so once, and therefore is an idempotent operation
        - Generally, any operation that just reads data is obviously idempotent -> updates data is generally not
    - LOOKUP and READ requests are trivially idempotent
        - Interestingly, WRITE requests are also idempotent -> if WRITE fails, we can simply just try again
        - Because write requests contain the data, the count, and the exact offset of the data to write to, it can be idempotent

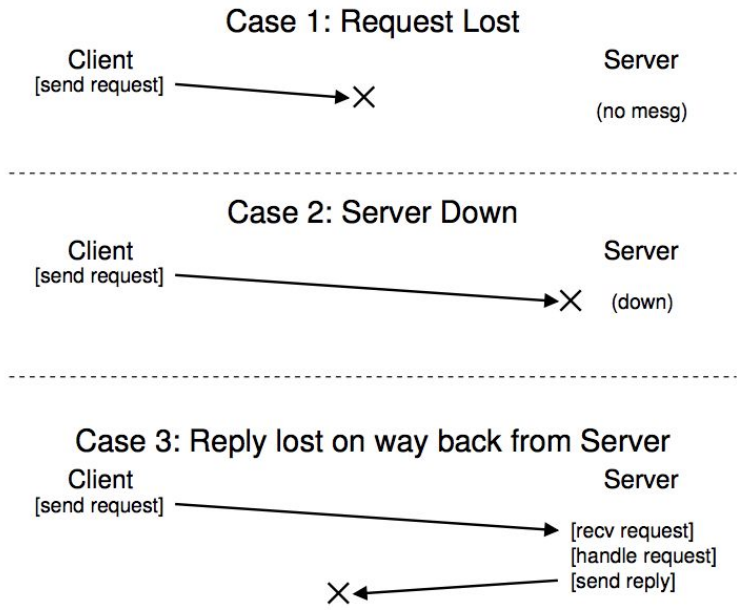


Figure 49.6: **The Three Types of Loss**

-
- Client can handle all timeouts in a unified way -> if a WRITE request was simply lost (CASE 1), the client will retry it
    - In all cases, we can just simply retry it
- Aside: Most operations are hard to make idempotent
    - Ex: when you try to make a directory that already exists, you are informed that mkdir fails
    - if the file system receives a mkdir protocol message and executes it successfully but the reply is lost, the client may repeat it and encounter the failure when in fact the operation at first succeeded and only failed on the rety

49.8: Improving Performance: Client-Side Caching

- Distributed File systems are good for a number of reasons but sending all read and write requests across the network can cause performance issues -> **How can we improve the performance of a distributed file system?**
- The NFS client-side file system caches file data and metadata that it has read from the server in client memory
    - first access is expensive but subsequent ones are not as expensive because they are cached in client memory
- Cache also serves as a temporary buffer for writes
    - When a client application first writes to a file, the client buffers the data in client memory before writing the data out to server
    - Decouples application write() **latency** from actual write performance

        - the application's call to write() succeeds immediately -> just puts the data in the client-side file system's cache)

49.9: The Cache Consistency Problem

- Adding caching into any sort of system with multiple client caches introduces a problem called the **cache consistency problem**
- ex: two clients and a single server
    - client C1 reads a file F, and keeps a copy of the file in its local cache
    - A different client C2, overwrites the file F, thus changing the contents -> F(v2)
    - Finally, there is a third client C3 that has not yet accessed the file
- Two subproblems
    - Update Visibility: When do updates from one client become visible to the others?
        - client C2 may buffer writes in its cache for a time before propagating them to the server
        - In this case, while F(v2) sits in C2's memory, any access of F from another client will fetch the old version of the file
        - Thus, by buffering writes at the client, other clients may get stale versions of the file/old versions
    - Stale Cache
        - C2 has finally finished its writes to the file server, and thus the new version of the file is located in memory for that particular machine
        - However, other clients will still get the old copies of the file in their own caches
- How to solve cache consistency problems: NFSv2
    - **flush-on-close (a.k.a close-to-open):** When a file is written to and subsequently closed by a client application, the client flushes all updates (i.e dirty pages in the cache) to the server
        - this ensures that a subsequent read/another node/client will see the latest file version
    - To address the stale cache problem, NFSv2 clients first check to see whether a file has changed before using its cached contents
        - when opening a file, the client-side file system will issue a GETATTR request to the server to fetch the file's attributes
        - The attributes include information as to when the file was last modified on the server
        - If the time of modification was more recent than the one on the server, it invalidates the one on the server and removes it from the client cache
- New problem arose when they tried to solve the stale-cache problem
    - The NFS server was flooded with GETATTR requests
    - to remedy this situation, SUN implemented an **attribute cache** to each client
        - A client would still validate a file before accessing it, but most often would just look in the attribute cache to fetch the attributes

49.10: Assessing NFS Cache Consistency

- Flush on close behavior was added to make sense, but introduced a certain performance problem
- Specifically, if a temporary or short-lived file was created on a client and then soon deleted, it would still be forced to the server
- A more ideal implementation might keep such short-lived files in memory until they are deleted
- The addition of an attribute cache into NFS made it very hard to understand or reason about exactly what version of a file one was getting
    - sometimes you would get the latest version, while sometimes you wouldn't

49.11: Implications on Server-Side Write Buffering

- When data is read from disk, NFS servers will keep it in memory, and subsequent reads of said data will not go to disk, a potential small boost in performance
- NFS servers absolutely may not return success on a WRITE protocol request until the write has been forced to stable storage (e.g. to disk or some other persistent storage device)
    - To avoid this problem, NFS servers must commit each write to stable/persistent storage before informing the client of success -> enables the client to detect server failure during a write, and thus retry until it finally succeeds

Lease-Based Serialization

Challenges of Distributed Locking

- Among Deutsch's Seven Fallacies of distributed computing were
    - zero latency
    - reliable delivery
    - stable topology
    - consistent management
- In locking distributed systems
    - In a single node, a compare-and-swap mutex operation might take tens on nanoseconds. Obtaining a lock through exchange will likely take at least tens of milliseconds
    - In a single node, a mutex operation is guaranteed to complete, while in a distributed system, requests/responses could be lost
    - When a single node crashes, all of its applications go down with it and all locks will be reacquired on restart. Node holding the lock can crash without releasing it, causing all other clients to hang
    - If a process dies, the OS knows it, and has the possibility of automatically releasing all locks held by the process. If the OS dies, all lock-holding process will die, and nobody will have to cope with the fact that the OS no longer knows who holds what locks. However, in a distributed system, there is no meta-OS to observe the failure and perform the cleanup

Addressing these Challenges

- Easy to send all locking requests and messages to a central server who will implement then with local locks
- More complex failure cases and greater deadlock risks can be dealt with by replacing locks with **leases**
    - A lock grants the owner exclusive access to a resource until the owner releases it
    - A lease grants the owner exclusive access to a resource until the owner releases it or until the lease time expires
- Locks work on an honor system. An actor who does not yet have a lock will not enter the critical section that the lock protects
- Leases are often enforced. When a request is sent to a remote server to perform some operation, that request includes a copy of the requestor's lease. If the lease has expired, the responding resource manager will refuse to accept the request
- Does not matter why a lease may have expired
    - the release message was lost in transit
    - the owning process has crashed
    - the node on which the owner was running has crashed
    - the owning process is running slowly
    - the network connection to the owning node has failed
- Since leases are enforced, operations from the previous owner will no longer be accepted, and the leases can be granted to a new requestor -> automatically recover from any failure of the lease-holder.
- Two problems with above
    - An expired lease prevents the previous owner from performing any further operations on the protected resource. But if the tardy owner was part-way through a multi-update transaction, the object may be left in an inconsistent state -> updates should be done atomically -> if a lease expires before the operation is committed, it should fall back to its previous state
    - The choice of the lease period may involve some careful tuning. Too short -> an owner may have to renew it many times. Too long -> system may take a long time to recover from a failure
- Sending a network message for every entry into a short critical section would be disastrous. Using leases for long lived resourced (like DHCP allocated IP addresses) can be extremely economical. It comes down to
    - the number of operations that can be performed under a single lease grant
    - the ratio of the costs of obtaining the lease to the costs of the operations that will be performed under that lease

Evaluating Leases

- Mutual Exclusion
    - leases are at least as good as locks -> plus potential enforcement
- Fairness

    - Depends on the policies implemented by the remote lock manager, who could easily implement either queued or prioritized requests
- Performance
    - Remote operations are by their very nature, **expensive,** but we aren't going to network leases for local objects. **If leases are rare and cover large numbers of operations, they can be a very efficient mechanism**
- Progress
    - The good news is that automatic preemption makes leases immune to deadlocks. If a lease-holder dies, other would-be leases must wait for the lease period to expire
- Robustness
    - Clearly more robust than single-system mechanisms
- Problems with Leases
    - correct recovery from lock-server failures are extremely complex, while they are easily recovered from client failures
    - Raises the issue of the lease expiration duration in a distributed system without a universal time standard

Opportunistic Locks

- Contention is rare for most resources, and the locking code is only there to ensure correct behavior in unlikely situations -> seems unfair to have to pay the high cost of remote lock requests when dealing with an unlikely problem
- A requestor can ask for a long term lease, enabling that node to handle all future locking as a purely local operation
    - if another node requests access to the resource, the lock manager will notify the op-lock owner that the lease has been revoked, and subsequent locking operations will have to be dealt with through the centralized lock manager

Consensus (Computer Science)

- A fundamental problem in distributed computing and multi-agent systems is to achieve overall system reliability in the presence of a number of faulty processes
- Examples of consensus: Whether to commit a transaction to a database, agreeing on the identity of a leader, state machine replication, and atomic broadcasts
- Real world applications include clock synchronization, PageRank, opinion formation, smart power grids, etc.

Problem Description (Consensus)

- requires agreement among a number of processes for a single data value
- Consensus protocols must be fault tolerant or resilient
- The consensus problem is a fundamental problem in control of multi-agent systems
- The output value of a consensus protocol must be the input value of some process
    - Another requirement is that a process may decide upon and output a value only once and this decision is irrevocable

- A consensus protocol tolerating halting failures must satisfy the following properties
    - **Termination:** Every correct process decides some value
    - **Validity:** If all processes propose the same value v, then all correct processes decide v
    - **Integrity:** Every correct process decides at most one value, and if it decides some value v, then **v must have been proposed by some process**
    - **Agreement:** Every correct process must agree on the same value
- A protocol that can correctly guarantee consensus amongst n processes of which at most 1 fail is said to be t-resilient
- Evaluating the performance of consensus protocols:
    - Running time and message complexity
    - Running time is given in big-o notation in the number of rounds of message exchange as a function of some input parameters
    - Message complexity refers to the amount of message traffic that is generated by the protocol
    - Other factors may include memory usage and the size of the messages

Models of Computation

- Two types of failures that a process may undergo, **a crash failure or a Byzantine failure**
- **Crash failure:** occurs when a process abruptly stops and does not resume
- **Byzantine failure:** failures in which absolutely no conditions are imposed
    - may occur as a result of the malicious actions of an adversary
    - a process that experiences this failure may send contradictory or conflicting data to other processes, or it may sleep and then resume activity after a lengthy delay
- For a consensus protocol that tolerates Byzantine failures is given by strengthening the Integrity constraint
- **Integrity (Revised)**
    - If a correct process decides v, then v must have been proposed by some correct process
- Varying models of computation may define a consensus problem
- **Binary consensus:** special case of consensus that restricts the input and hence output domain to a single binary digit (0,1)
- Consensus is impossible in a real-world fully asynchronous message passing distributed system (in a worst case scenario)
    - In an asynchronous model, some forms of failures can be handled by a synchronous consensus protocol.
    - Ex: the loss of a communication link may be modeled as a process which has suffered a Byzantine failure

Equivalency of Agreement Problems

- **Terminating Reliable Broadcast**

    - A collection of n processes, numbered 0 to n-1, communicate by sending messages to one another. Process 9 must transmit a value v to all processes such that
    - 1) If process 0 is correct, then every correct process receives v
    - 2) For any two correct processes, each process receives the same value
    - Also known as the general's problem
- **Consensus**
    - **Formal requirements for a protocol**
        - **Agreement:** All correct processes must agree on some value
    - **Weak Validity:** If all correct process receive the same input value, they they must all output that value
    - **Strong validity:** For each correct process, its output must be the input of some correct process
    - **Termination:** All processes must eventually decide on an output value
- **Weak Interactive Consistency**
    - For n processes in a partially synchronous system, each process chooses a private value
    - 1) If a correct process sends v, then all correct processes receive either v or nothing (integrity property)
    - 2) All messages sent in a round by a correct process are received in the same round by all correct processes (consistency property)

Solvability Results for Some Agreement Problem

- In a fully asynchronous system, there is no consensus solution that can tolerate one or more crash failures even when only requiring the non triviality property
    - known as the FLP impossibility proof
- FLP doesn't mean that consensus can never reached -> merely that under the model's assumptions, no algorithm can always reach consensus in bounded time -> highly unlikely to occur

Some Consensus Protocols

- Binary consensus protocol that tolerates Byzantine failures is the Phase King algorithm
    - solves consensus in a synchronous message passing model with n processes up to f failures, provided n > 4f
    - There are f+1 phases and 2 rounds per phases
    - Each process keeps track of preferred output -> first round of each phase each process broadcasts its own preferred value to other processes -> It then receives values from processes and determines which value is the majority value and count. In the second round of the phase, the process whose i**d matches the current phase number is designated the king of the phase**. King broadcasts the majority value it observed in the first round and serves as a tiebreaker. Each process then updates its preferred value

- Google has implemented a distributed lock service library called Chubby -> maintains lock information in small files which are stored in a replicated database to achieve high availability in face of failure

In Shared-Memory Systems

- To solve the consensus problem in a shared-memory system, concurrent objects must be introduced.
    - concurrent objects are data structures which help concurrent processes communicate and reach an agreement
- Traditionally, people use critical sections to solve this problem, but critical sections present risks that may occur when a process dies inside the critical section
- **Wait-free implementation**
    - **Guarantees consensus in a finite number of steps**
Consensus number	Objects
1	Read/write registers
2	test&set, swap, fetch&add, queue, stack
...	...
2n-2	n-register assignment
...	...
∞	Memory-to-memory move and swap, augmented queue, compare&swap, fetch&cons, sticky byte
    - Consensus number means the maximum number of processes in the system which can reach consensus by a given object

Two-phase Commit Protocol

- In transaction processing, databases, and computer networking, the two-phase commit protocol (2PC) is a type of **atomic commitment protocol**
- Distributed algorithm that coordinates all the processes that participate in a distributed atomic transaction -> on whether to commit or abort the transaction
- In a “normal execution” of any single distributed transaction (i.e., when no failure occurs, which is typically the most frequent situation), the protocol consists of two phases
    - 1) **Commit-request phase:** coordinator processes attempts to prepare all the transaction’s participating processes to take the necessary steps for either committing or aborting the transaction to vote (commit or abort/ yes or no)
    - 2) **The commit phase:** Based on voting of the cohorts, the coordinator decides whether to commit or abort the transaction and notifies the result to all cohorts.

Assumptions

- The protocol works in the following manner
    - one node is a designated coordinator, which is the master sit, and the rest of the nodes in the network are cohorts

- The protocol assumes that there is a stable storage at each node with a write-ahead log, no node crashes forever, data in the write-ahead log is never lost or corrupted in the crash, and any two nodes can communicate with each other (assumptions)
- The protocol is initiated by the coordinator after the last step of the transaction has been reached
- The cohorts then respond with an agreement message or an abort message depending on whether the transaction has been properly processed or not

Basic Algorithm

- **Commit Request Phase (voting phase)**
  - 1) Coordinator sends a query to commit message to all cohorts and waits until it has received a reply from all cohorts
  - 2) The cohorts execute the transaction up to the point where they will be asked to commit. They each write an entry to their undo log and an entry to their redo log
  - 3) Each cohort replies with an agreement message (yes), if the cohorts actions succeed, or an abort message (no), if the cohort experiences a failure that will make it impossible to commit
- **Commit Phase (Completion phase)**
  - **Success**
    - If the coordinator received an agreement messages from **all** cohorts during this commit-request phase
    - 1) The coordinator sends a commit message to all the cohorts
    - 2) Each cohort sends an acknowledgement to the coordinator
    - 3) Each cohort sends an acknowledgement to the coordinator
    - 4) The coordinator completes the transaction when all acknowledgments have been received
  - **Failure**
    - If any cohort votes no during the commit-request phase (or the coordinator's timeout expires)
    - 1) The coordinator sends a rollback message to all the cohorts
    - 2) Each cohort undoes the transaction using the undo log, and releases the resources and locks held during the transaction
    - 3) Each cohort sends an acknowledgment in the coordinator
    - 4) The coordinator undoes the transaction when all acknowledgements have been received

Message flow [edit]

```
Coordinator                                          Cohort
                        QUERY TO COMMIT
                    ---------------------------------->
                        VOTE YES/NO                  prepare*/abort*
                    <----------------------------------
commit*/abort*          COMMIT/ROLLBACK
                    ---------------------------------->
                        ACKNOWLEDGMENT               commit*/abort*
                    <----------------------------------
end
```

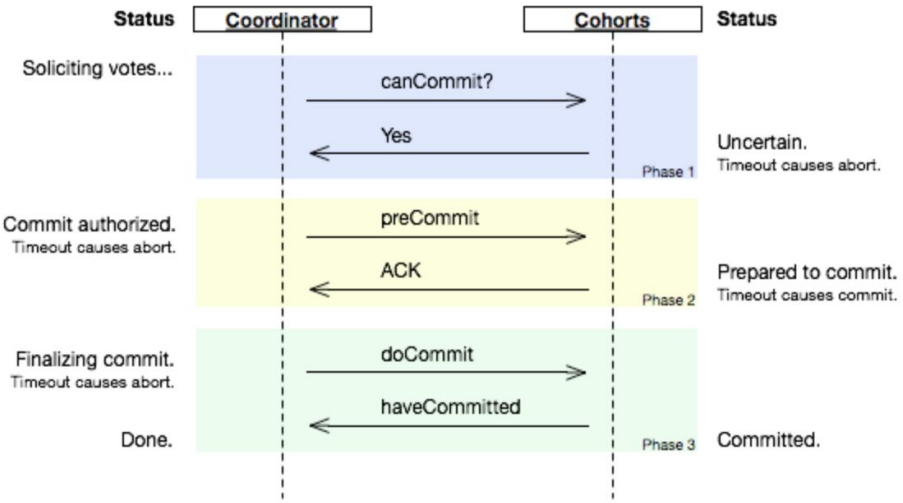
An * next to the record type means that the record is forced to stable storage.[4]

Disadvantages of two-phase commit protocol

- The greatest disadvantage of the two-phase commit protocol is **that it is a blocking protocol**
- If a coordinator fails permanently, some cohorts will never resolve their transactions: After a cohort has sent an agreement message to the coordinator, it will block until a commit or rollback is received

Three-phase Commit Protocol

- Also known as 3PC, the three-phase commit protocol is a distributed algorithm which lets all nodes in a distributed system agree to commit a transaction.
- Unlike the 2PC, 3PC is non-blocking and places an upper bound on the amount of time required before a transaction either commits or aborts
- Ensures that if a given transaction is attempting to commit via 3PC and holds some resource locks, it will release the locks after the timeout



Coordinator (3PC)

- 1) The coordinator receives a transaction request. if there is a failure at this point, the coordinator aborts the transaction (i.e. upon recovery, it will consider the transaction aborted) -> otherwise, the coordinator can send a canCommit? message to the cohorts and moves to the **waiting state**
- 2) If there is a failure, timeout, or if the coordinator receives a No message in the waiting state, the coordinator aborts the transaction and sends an abort message to all of the cohorts. Otherwise the coordinator will receive Yes messages from all cohorts within the time window, so it sends preCommit messages to all cohorts and moves to the prepared state
- If the coordinator succeeds in the prepared state, it will move to the commit state. However, if the coordinator times out while waiting for an acknowledgement from a cohort, it will abort the transaction. In the case where an acknowledgement is received from the majority of cohorts, the coordinator moves to the commit state as well

Cohort

- 1) The cohort receives a canCommit? message from the coordinator. If the cohort agrees it sends a Yes message to the coordinator and moves to the prepared state. Otherwise it sends a No message and aborts. If there is a failure, it moves to the abort state
- 2) In the prepared state, if the cohort receives an abort message from the coordinator, fails, or times out waiting for a commit, it aborts. If the cohort receives a preCommit message, it sends an ACK message back and awaits a final commit or abort
- 3) If, after a cohort member receivers a preCommit message, the coordinator fails or times out, the cohort member goes forward with the commit

Motivation

- Primary disadvantages of 2PC are covered in 3PC
    - in 2PC, you cannot properly recover from a failure of both the coordinator and a cohort member during the commit phase
- Eliminates problems by introducing the Prepared to commit state
    - If the coordinator fails before sending preCommit messages, the cohort will unanimously agree that the operation was aborted
    - The coordinator will not send out a doCommit message until all cohort messages have sent an ACK message that they are **Prepared to commit**
    - Eliminates the possibility that any cohort member actually completed the transaction before all cohort members were aware of the decision to do so, an ambiguity in the two-phase approach

Disadvantages of 3PC

- Main disadvantage: **cannot recover in the event the network is segmented in any manner**
- Assumes a fail-stop model, where processes fail by crashing and crashes can be accurately detected

- Protocol requires at least three round trips to complete, needing a minimum of three round trip times (RTTs)

Authentication Servers

- In a large distributed systems, keeping track of information about all of the authorized users and what each is authorized to do may be a great deal of work
- Public Key Certificates as a means of authenticating ones' self to a server that was not previously aware of us or able to engage us in a password dialog. But …
    - may not be practical to require all actors to have registered public/private key pairs, or for all servers to use them as the basis for authentication
    - even if we have an authenticated identity for an actor, we still may need to be able to access and interpret a large access control database to determine whether or not they are allowed to perform particular actions
- Unreasonable to expect application level services to be able to decide who can do what

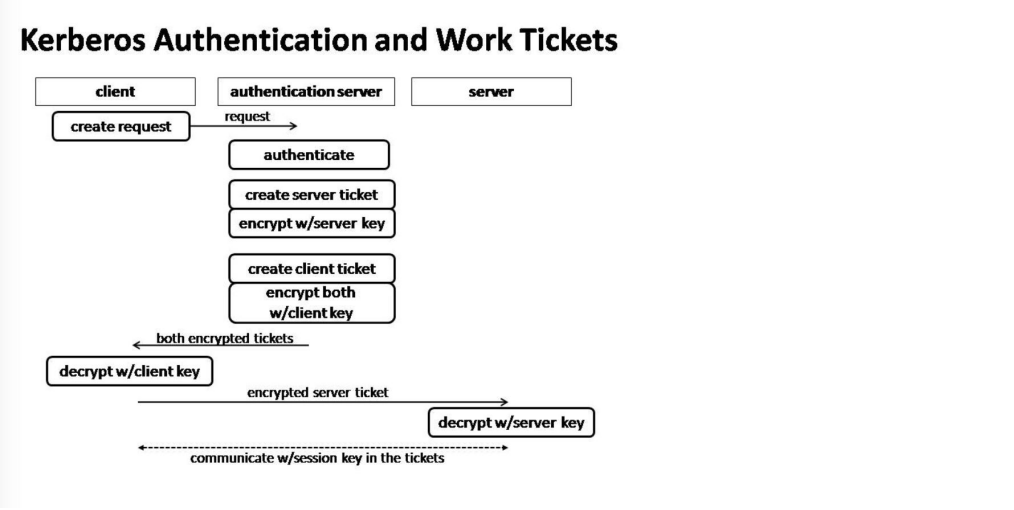
Authentication and Authorizåtion as a Service

- A public key certificate is one type of credential, which we trust because it was signed by someone we trust. We could use similar technique to get trusted **capabilities**
    - An actor would contact an authentication server, and describe the actions to be performed
    - the authentication server would both authenticate the actor and determine whether or not that actor was allowed to do the proposed operations
    - after a successful access check, the authentication server would create a **work ticket** describing the actor, the permitted operation, and any other relevant information, and then sign it with his private key, for example.
    - the actor would present the work ticket to the appropriate server along with the request
    - the server would not have to do any authentication or authorization checking beyond verifying the validity of the work ticket
- Relieves applications of all responsibility for authentication and authorization checking. Also completely isolates them from the particular mechanisms and policies by which identities are authenticated and privileges ascertained

Work Tickets

- Work tickets are unforgeable -> signature prevents random agents from creating work tickets
    - if they include expiration dates, they cannot be reused
    - They are, however, transferable and therefore stealable
- How can we determine that the ticket can only be used by the authorized agent/actor?
    - We could certainly include a public key of the authorized agent in the ticket, but this would get the server back into the authentication business

- Another common approach is to have the authentication server generate a session, and encrypts a copy for the client with the client's public key, and a copy for the server with the server's public key.



- Each work ticket contains a session key, generated by the authentication server
    - When the server receives the work ticket, it can verify the authentication server's signature and so trust that the client has been authenticated

Work Tickets without Public Key Encryption

- Can be done with symmetric key encryption as well (only shared with the authentication server)
- keys could be used
    - a) to authenticate a ticket as having come from the authentication server
    - b) to communicate information which can only be read by actors of the authorized agent