Distributed Operating Systems Overview

Basic Principles of Distributed Systems

GOALS

1. **Connecting Users and Resources**: Make it easy for users to access remote resources, and to share them with other users in a controlled way.
2. **Transparency**: Hide the fact that its processes and resources are physically distributed across multiple computers, the system present itself to users as a single system.
3. **Openness**: The system offers services according to standard rules that describe the syntax and semantics of those services.
4. **Scalability**:

Transparency	Description
Access	Hide differences in data representation and how resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource
Persistence	Hide whether a (software) resource is in memory or disk

Services are generally specified by **interfaces** described in an **Interface Description Language (IDL)**. Interface descriptions only capture the syntax of a service, not semantics. A good specification is complete and neutral.

Interoperability: Extent to which two implementations of systems or components from different manufacturers can co-exist and work together.

Portability: Characterizes to what extent an application developed for a distributed system *A* can be executed, without modification, on a different distributed system *B* that implements the same interfaces as *A*.

Scalability can be measured along three different dimensions:

a. Scalable with respect to its size.
b. Geographically, users and resources may lay far apart.
c. Administratively, easy to manage even if it spans many independent administrative organizations.

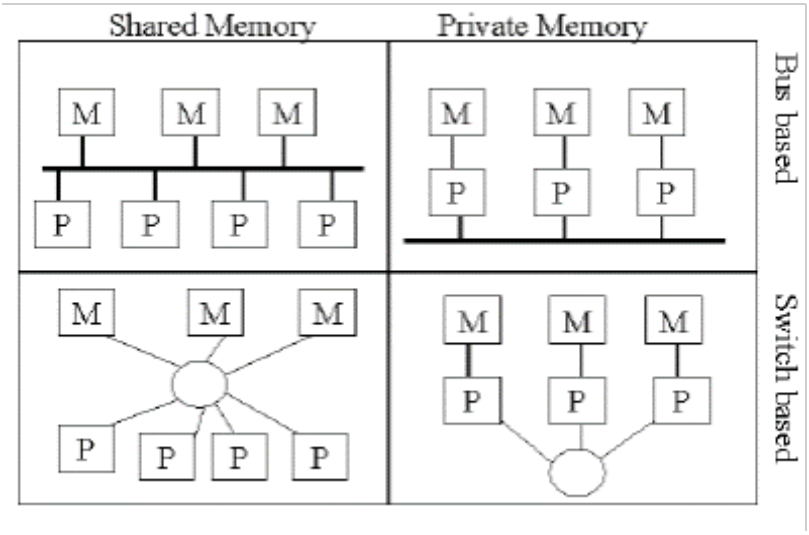
Characteristics of distributed algorithm:

a. No machine has complete information of the system state.
b. Machines make decision based only on local information.
c. Failure of one machine does not ruin the algorithm.
d. There is no implicit assumption that a global clock exists.

Scalability tricks:

a. **Distribution**: Split component into smaller parts and spreading those over the system.
b. **Replication**: Copy the component across the system. (Caching for example)
c. **A-synchronous**: Do not wait for a reply, but continue the process.

Architectures



Types of Distributed Systems

System	Description	Main goal
DOS	Tightly coupled OS for multiprocessors and homogeneous multicomputers	Hide and manage hardware resources
NOS	Loosely coupled OS for heterogeneous multicomputers (LAN and WAN)	Offer local services to remote clients
Middleware	Additional layer atop of NOS implementing general purpose services	Provide distribution transparency

I.Concurrent Processes/Threads

- Internal organization of processes
- Do processes support mutliple threads?
- Uses of threads in distributed systems:
  - may continue using the CPU when blocking I/O is performed
  - can build efficient server w/mutliple concurrent threads

Clients:

- responsible for UI: simple displays to complex interfaces handling compound docs.
- Client software aims to hide distribution transparency by hiding details of comm.
  w/servers, server locations, and possible replication.

- May also hide failures and help recover from them

Servers:

- iterative vs. concurrent,
- possible implement services,
- stateless vs. stateful
- Addressing services,
- Mechanisms for interruption

Object Servers:

- process with several objects in its address space
- many ways to invoke objects
  - Start new thread for each invocation request
  - thread per object
  - single thread for all objects
- object adapter allows allows server to handle different invocation policies
- single server can use several object adapter

Code Migration:

- increases performance -> when comm is expensive, can ship some computations from server to client
- increases flexibility -> client can download software needed to communicate with particular server
- problems with usage of local resources: must either migrate resources, establish new bindings at the target machine,
- code migration also demands consideration of heterogeneity -> best solutions is virtual machines

II.Client/Server and object models

- synopsis: **distributed systems need a more horizontal organization where clients and servers are physically distributed and replicated across several servers**
- server is a process implementing a specific service
- client requests the service offered by a server
- Communication between client and server can be handled by connectionless or connected protocol
  - Connectionless more efficient, better suited if underlying network reliable (LANs)
  - Connection-oriented lower performance – high setup/tear down cost, well suited for wide-area networks
- Application Layering: Could be divided into User-Interface, Processing, and data level
  - User-Interface: typically implemented by client
  - Processing level: contains core functionality
  - Data level: the programs that maintain the actual data on which the appl. operate. Also keep data consistent across different applications
- Architectures:
  - simplest type w/only two machines->client/server
  - multi-tiered (**vertical distribution**-> achieved by placing logically different components on different machines)
    a. terminal dependant part of UI on client – applications have remote control over data presentation
    b. all UI software on client side
    c. part of application resident on client. ex: form submission

    d. UI and all of app on client (only db on server). many client/server apps
    e. part of db also on client. ex: web cache
- modern architectures (**horizontal distribution**-> client or server may be split up into logically equivalent parts, with each part operating on a copy of the data set for load balancing).
  - ex: web server replication
  - clients can also be distributed. some collaborative apps w/no server (P2P).

III.Shared Memory interprocess communication (7.4,7.5)

- DSM systems differ from multiprocessor cache systems in that their memory modules are strictly local and physically seperated by communication links
- access latency higher than internal memory
- **efficient memory management algorithms and coherence protocols are the major design issues**
- Memory Management
  - blocks are basic units of data sharing and coincide with pages in each processors memory space
  - **Three access options for remote memory blocks:**
    - performed remotely at the remote node
    - remote block is migrated locally
    - remote block is replicated locally
  - factors that affect performance of memory management algorithms:
    - size of block – large block has better channel utilization, but longer transfer time
      - fault rate is a function of block size, program locality and read/write ratio for each process
    - block transfer communication overhead
    - read/write ratio- high read/write ratio gives good performance with read and full replication
    - program locality
    - number of interacting nodes and type of interaction – changes degree of sharing and the number of replicates that need to be managed. effect is higher for full-replication than read replication (updates have higher cost that invalidates)
  - **read-remote-write-remote (central server algorithm):**
    - can be implemented as a request sent to a centralized server – server responds with data for reads and acks for writes
    - multiple memory servers may perform same function, could be partitioned
    - no problem with memory coherence – server need only serialize request and response services
  - **read-migrate-write-migrate (migration algorithm)**
    - block is always migrated, data block in requested processor is marked invalid
    - reads and writes are always exclusive, owner processor of shared block changes dynamically
    - each process must update its virtual-page-to-physical-block mapping table -> broadcast new block location to all processors for every migration

        - better performance by exploiting localities, but suffers from thrashing
        - thrashing can be alleviated by replicating the shared block or reducing block size. could also limit migration (threshold of time or # of accesses)
        - coherence simple
    - **read-replicate-write-migrate (read replication algorithm)**
        - shared blocks classified as either shared-read-only or exclusive write
        - a read access to a remote block activates transfer, and all copies of the block are marked shared-read-only. subsequent reads can be concurrent
        - writes to a shared-read-only block use write-invalidate (invalidates all shared copies and then updates the local block). update block becomes exclusive copy
        - read-replication with write-invalidate – strong consistency (each read operation retrieves the value of the most current write)
        - good performance when reads dominate writes
        - must keep track of copies of shared block (invalidation message must be sent to processor holding copies) -> who is responsible for this data structure? : block owner may be static (original owner) or dynamic (current owner from recent write)
        - number of shared copies is a measure of read-misses between consecutive writes.
    - **read-replicate-write-replicate (full replication)**
        - all shared blocks fully replicated
        - write results must be timely and atomically propagated by the write-update protocol
        - dsms must use software for atomic broadcast protocols -> broadcast is a message and all messages must be delivered in same order to maintain sequential consistency
        - two-phase commit and sequencer can be used for full-replication
- Block owner and copy list
    - memory coherence managers must maintain some data structures for: 1-locating current block owner, 2-identifying replicas
    - when making a remote reference to a data object, each processor consults a page map table, which points to a remote processor and page number.
    - if the processor that receives the request is not the owner, it can forward the request, or tell the originating process who to contact next.
    - for replicated shared-read-only blocks the owner may be any processor that is holding a valid copy (processor with a read-miss can get a copy from any of them). one can be designated master copy.
    - for replicated shared-read-write blocks in the write update protocol, desirable to update the master copy first. will need a copy list for each shared block to keep track of replicas
    - the copy list could be distributed among all sharing nodes: each processor maintains a from (the processor the block was copied from) and to(processor to which the copy was sent) pointer for each block known to it.

    - after a write operation to a shared block, the processor broadcasts invalidation or update messages along the path of the pointers (like a spanning tree) -> difficult to know when broadcast is complete
    - copy list could be implemented like a distributed linear linked list (each processor keeps two pointers one to master copy node and the other to the next processor in the list)
        - write request is sent to node with master copy for distribution
        - the end node acks completion to requesting node
        - read request can be made to any owner of the block copy – requesting node is appended to the end of the list after replication
- Object based DSM
    - coherence unit is an object (data structure with prescribed methods or operations) or data structure
    - information sharing is explicit
    - advantages:
        - only declared shared objects that form the logical shared memory space need to be managed by the DSM kernels
        - synchronization accesses can be applied at the object level
        - false sharing eliminated
        - object-based DSM can be implemented w/o significant change to a system using a layer of library routines on top of message passing
    - shared objects should be migrated or replicated for efficient lcal and concurrent access
    - synchronization can be explicit (acquire(X) release(X)) or implicit w/other methods defined on the object -> but locking is required
- **DSM Implementation**
    - performance of a DSM system is directly related to performance of underlying hardware
    - recent communication mechanims are faster and support mapping of interconnect hardware into the address space of the proces participating in the DSM system -> caused DSM systems based on hardware, software, or both
    - DSM Characteristics:
        - **Implementation level**: hardware, software, hybrid
        - **architecture configuration**: describes the system and interconnect mechanism that the DSM is running on
        - **shared data organization**: either structured or unstructured – affected by implementation level.
            - structured data are objects or language types (compiler inserted primitives and programmer-inserted directives used as support) -> used by software or hybrid systems
            - unstructured data – used by hardware DSMs managed as a series of independent shared-memory locations
        - **coherence unit** – word, cache, block, page, data structure or object
        - **DSM algorithm** – SRSW, MRSW, MRMW
        - **management responsibility** – centralized or distributed (static or dynamic)
        - **consistency model**
        - **coherence control protocol**

IV.Message-passing interprocess communication

- in a message-passing system communicating processes pass composed messages to the system transport service, which provides connectivity for message transfer in the network
- interfaces to the transport service are explicit communication primitives: send(destination, msg); receive(source,msg)
- Four options for **addressing source and destination**
  - **process name** (global process identifier) – made unique by concatenating the network host address with the locally generated process id. -> implies that only one direct logical communication path exists between any pair of processes (symmetric). could be asymmetric if only sender needs to indicate recipient
  - **link –** introduced to allow multiple data paths between processes and direct communication. created and released on request. managed locally by the system kernel and are unidirectional communication channels. messages sent through a link are mapped to a network communication pathand delivered to the remote host.
  - **mailbox -** allows indirect communication. mailbox is shared among a group of processes -> provides multipoint and multipath communication.
    - requires proper synchronization
  - **port –** abstraction of a finite-size FIFO queue maintained by the kernel. messages can be appended or removed from the queue by send and received operations through a communication path. similar to links except bidirectional and buffered
- **synchronization and buffering:**
  - the synchronization of a message transfer occurs between the user process and system kernel, kernel and kernel, and source and destination processes.\
  - common default is a nonblocking send and a blocking receive-> assumes reliable message delivery
  - options for send primitive:
    - nonblocking (asynchronous) send
    - blocking send – sender process released after message transmitted to the network
    - reliable blocking send – sender process released after message has been received by receiver's kernel
    - explicity blocking send – sender process is released after message has been received by the receiver process
    - request and reply (explicit client/server) – sender process is released after message has been processed by the receiver
  - model assumes buffer space – can combine buffer at sender kernel, receiver kernel, and communication network into one buffer -> if buffer size is unbounded, then an asynchronous send is never blocked. if all components are bufferless, then sender and receiver must be synchronized to perform message transfer
- pipe and socket APIs
  - pipes implemented with a finite-size, FIFO byte-stream buffer maintained by the kernel. pipe is a unidirectional communication link.
  - pipe system call returns two descriptors: one for writing and one for reading
  - pipes used only for related processes

- for unrelated processes, need to uniquely identify a pipe since descriptors cannot be shared. -> replace the kernel pipe data structure with a special FIFO file (uniquely identified by path names)
- with named pipes, communicating processes need not exist concurrently, but must share a common file system.
- for interdomain process communication, need IPC API running on top of transport services (socket, or Transport Layer Interface)

o **group communication and multicast**
- two types of multicast application scenarios: client wants a service from any server that can perform it, or client needs to request a service from all members of a group of servers.
- first kind (any available server) – not necessary for all servers to reply as long as one does. **system needs only to guarantee the delivery of the multicast message to the reachable, nonfaulty processes.**-> **best-effort multicast**
- second case (group of servers) - necessary to **ensure that all servers have received the message. should be received by all or none**-> **reliable multicast**
- failures of recipients during multicast may be detected with time-outs or acks. sender can then abort or continue, excluding failed members.
- failure of sender during multicast – recipients must choose new originator. recipients buffer messages until delivery is safe. failures handled by virtual synchrony
- multicast message orderings:
  - **FIFO** – multicast messages from a single source are delivered in the order that they were sent. messages can be assigned a sequence number by communication handler at each site (receiving).
  - **Causal** – causally related messages from multiple sources are delivered in their causal order. two messages are causally related if one message is generated after the receipt of the other. can use a vector of sequence numbers maintained by each member.
    - Sk represents number of messages received so far by group member k. when member i multicasts a new message m, it increments Si by 1 and attaches the vector S to m.
    - Members use three rules to accept/reject
      - accept m if Ti=Si+1 and Tk<=Sk for all k/=i (case 1->member j expecting the next message in sequence from member i, case 2->member j has delivered all of the multicast messages that member i had delivered when it multicast m)
      - delay m if Ti>Si+1 or there exists k/=i s/t Tk>Sk (in case 1 some previous multicast messages from member i are missing, in case 2 member i received more multicast messages from some other members of the group when it multicast m that member j did)
      - reject m if Ti<=Si (message is a duplicate)
  - **Total** – all messages multicast to a group are delivered to all memebers of the group in the same order. requires that a multicast must be completed and the multicast messages must

be ordered by the multicast completion time before delviery to the processes.
    - use two-phase total-order multicast:
        - originator broadcasts messages and collects acks with logical timestamps from all members.
        - after all acks received, originator send commitment message that carries the highest ack timestamp as the logical time for the commitment.
    - could simplify by using a global sequence number server

V.Remote procedure call and remote object invocation

- **RPC**:
    - aimed at achieving access transparency, but poor at passing references
    - service is implemented by a procedure which is executed at the server
    - client is offered the signature of the method
    - client side implementation (stub) wraps parameter values into a message and sends to server
    - server calls procedure and returns results in a message
    - Problems w/RPC: **disjoint address space, parameter passing, machine failures**
        - passing value parameters-> problems with machine encoding of ints, etc
        - passing reference parameters-> use copy/restore (client copies data, server writes and sends back, client overwrites original with server data)

    - Idea is that we have a procedure call that is remote, but transparent to the caller. The caller of the procedure is not aware of the location of the procedure. The passing of parameters is important for RPC, since each technique must make use of different technique of passing parameters around:
        - **Call by value**: A value parameter is copied onto the stack. To the called procedure the parameter is initialized as a local variable.
        - **Call by reference**: A reference parameter is a pointer to a variable in memory (i.e. it is an address), rather than the value of the variable.
        - **Call by copy/restore**: A variable is copied to the stack by the caller, as in call by value, and then copied back after the call, overwriting the caller's original value.
    - RPC achieves transparency by using stubs. A stub looks like a local procedure but actually takes care of the sending and receiving of the variables to the remote procedure. The following steps occur:
        - Client procedure calls client stub in normal way.
        - Client stub builds a message and calls the local OS.
        - The client's OS sends the message to the remote OS.
        - The remote OS gives the message to the server stub.
        - The server stub unpacks the parameters and calls the server.
        - The server does the work and returns the result into the stub.
        - The server packs it in a message and calls its local OS.
        - The server's OS sends the message to the client's OS.

        - The client's OS gives the message to the client stub.
        - The stub unpacks the result and returns it to the client.
    - **For parameter passing we need to come to an agreement of type representation,** otherwise difficulties with different systems (big/little endian for example). We use an Interface Definition Language (IDL) to specify the interface of a remote procedure.
    - **(un)Marshalling:** The (un)packaging of parameters in a message.

Flavors of Remote Procedure Call (RPC)

- **Doors:** A system for local IPC (inter process communication using RPC).A *door* is a generic name for a procedure in the address space of a server process that can be called by processes collocated with the server. It requires support from the local OS.
- **Asynchronous RPC:** The server immediately sends a reply back to the client, the moment the RPC request is received, after which it calls the requested procedure.
- **Deferred synchronous RPC:** Asynchronous RPC where the server calls back the client upon completion of procedure to hand over the results.
- **One way RPC:** Client doesn't even wait for an acknowledgement when making RPC call.

Performing an RPC

When a server goes down we can have some problems, therefore we can have the following semantics:

- **At most once:** No call is ever carried out more than once, even in the face of system failures.
- **Idempotent:** A call can be carried out multiple times with the same result.

- **ROI/RMI**
    - key feature is that objects encapusulate data (state) and operations on data (methods). methods made available through interfaces.
    - separation of interface and implementation allows to place **interface at one machine and object on another (distributed object)** -> state is not distributed
    - when a client binds to a dis. obj. an implementation of the objects interface (a **proxy**) is loaded into the clients address space.
        - proxy marshals method invocations into messages and unmarshals reply messages to return the result of the method invocation to the client
    - incoming invocation requests are passed to the server stub (skeleton) which unmarshals them to proper method invocations at the object's interface at the server
    - **compile-time objects:** instance of a class
        - allows tranparency for programmers
        - increases dependence on one programming language
    - **run-time objects:** leave implementation open. sometimes with an **object adapter** (a wrapper around implementation to give it the appearance of an object)
        - object defined in terms of the interfaces they implement

        - interface implementation can be registered at an adapter which can make that inerface available for remote invocations
- **persistent objects:** continues to exist even if it is not currently contained in the address space of a server process. usually the current managing server can store the objects state on secondary storage and exit. other servers can then read the objects state into their address space and then handle requests
- **transient objects:** exists only as long as the server process that manages the object.
- **binding:** causes a proxy to be placed in the processes address space, implementing an interface containing the methods the process can invoke
    - **implicit binding:** the client is offered a simle mechanismt that allows it to directly invoke methods using only a reference to an object. client is transparently bound to the object at the moment the reference is resolved to the actual object.
    - **explicit binding:** the client should first call a special function to bind to the object before it can actually invoke its methods. generally returns a pointer to a proxy that then becomes locally available.
    - **object references:**
        - simple version would include the network addres of the machine where the actual object resides, along with an endpoint identifying the server that manages the object, plus an indication of which object.
            - drawbacks: if server crashes and gets a new endpoint after recovering, all object references are invalid. could use a daemon to match server Ids with endpoints. but even then. server could never move to another machine w/o invalidating all of the references to the objects it manages
        - better solution uses a location server that keeps track of the machine where an objects server is currently running. object references contain the network address of the location server, along with a systemwide identifier for the server
        - should also include identification of the protocol that is used to bind to an object and of those that are supported by the objects server
        - may include an **implementation handle** which refers to a complete implementation of a prxy that the client can dynamically load when binding to th e object
- static vs. dynamic RMI
    - **static invocation:** using predefined interface definitions (defined in interface definition language). interfaces of an object must be known when the client application is being developed
    - **dynamic invocation:** method invocation is composed at runtime.
- **parameter passing**
    - b/c most RMIs support systemwide object references, passing parameters less restricted.

- when invoking method w/object reference as a parameter, the reference is copied and passed as a value parameter, only when it refers to a remote object. (pass by reference)
- when referring to a local object (in the same address space as the client) the entire object is copied (pass by value)

- now allow systemwide object references to be passed as parameters
- better transparency for reference passing

VI.Distributed synchronization: mutual exclusion and leader election

1. Synchronization

Time is important in distributed systems, we need it so we can have total ordering on the occurrence of events.

Clock synchronization algorithms:

- **Timer:** Computer has a crystal which oscillates at well-defined frequency. Each crystal has a *counter* and *holding* register. Each oscillation decrements the counter by one. When counter hits zero an interrupt is generated (*clock tick)* and the counter register is filled with the value of the holding register.
- **Clock skew:** Difference in time between two clocks after running for a while

1. **Clock specs:** $1-\rho \le \frac{dC}{dt} \le 1+\rho$ the constant ρ refers to the *maximum drift rate*

Christian's algorithm: There is a *timeserver* with a WWV receiver. Periodically but no more than $\delta / 2\rho$ seconds each machine asks for the current time. 2 problems

1. Time change must be adjusted gradually.
2. Nonzero amount of time before reply. Estimate message propagation as $(\frac{T_1 - T_0}{2})$ or as $(\frac{T_1 - T_0 - I}{2})$. Where I is interrupt handling time, and T stands for message send and arrival times.

Berkeley algorithm: Timeserver is active, (ever so often) polling every machine for the time. Now timeserver calculates average time and sends it back to all machines.

Logical Clocks:

Lamport timestamps, achieve the total ordering by introducing happens before relation.

Happens before: *a* happens before *b* (a->b) whenever:

1. If *a* and *b* are events in the same process and *a* occurs before *b*, then a -> b
2. If *a* is the event of a message being sent by one process and *b* is the event of the message being received by another process, then a -> b

Using this we want to define that if a-> b then C(a) < C(b).

We can now create a global time (i.e. total ordering) as follows:

1. If *a* happens before *b* in the same process, C(a) < C(b) .
2. If *a* and *b* represent the sending and receiving of a message, respectively, C(a) < C(b). The receiver sets its time to +1 of the highest two times.

3. For all distinctive events *a* and *b*, C(a) /= C(b). (We achieve this by appending the process nr, behind the event time. For example an event at the same time in process 1,2 become 40.1 and 40.2)

You can built a *totally ordered multicast* easily now, (i.e. a multicast by which all messages are delivered in the same order to each receiver).

1. Each message is time stamped with current logical time of sender.
2. When it is multicast it sent to the sender, we also assume that messages don't get lost and arrive in order.
3. Upon receipt of a message it is placed in local queue, ordered according to time stamp. Receiver sends acknowledgement to all participants. (Note that acknowledgement message has a higher timestamp then received message!)
4. Process delivers message in the head of the queue to application when everyone acknowledges this message.

Vector timestamps

Lamport timestamps do not capture causality. I.e. C(a) < C(b) does not imply that *a* causally precedes *b*. Causality can be captured by making use of a:

Vector timestamp:

1. Vi[i] is the number of events that have occurred so far at Pi
2. If Vi[j] = k then Pi knows that *k* events have occurred at Pj

Now upon receipt of a message *m* from *i* to *j* we just update Vj[k] = max{Vi[k], vt[k]}, after this we update Vj[i] = Vj[i]+1. We can use this to implement causal multicast:

A message *r* is delivered only if the following conditions are met:

1. Vt(r)[j] = Vk[j]+1
2. For Every i/=j vt(r)[j] <= Vk[i]

Global state:

The global state of a distributed system consists of the local state of each process, together with the messages that are currently in transit, that is, that have been sent but not delivered.

Distributed snapshot: Reflects a state in which the distributed system might have been. It reflects a *consistent* global state. (for example cannot have received a message that has not been send yet). Can be graphically represented by a **cut**.

Snapshot:

If a process *Q* receives the marker requesting a snapshot for the first time, it considers the process that sent tat marker as its predecessor. When *Q* completes its part of the snapshot, it sends its predecessor a *DONE* message. By recursion, when the initiator receives a *DONE* from all its successors, it knows that the snapshot has been completely taken.

Termination detection:

Take a snapshot with all channels empty. We do this by returning a *CONTINUE* or *DONE* message. We send a *DONE* message whenever:

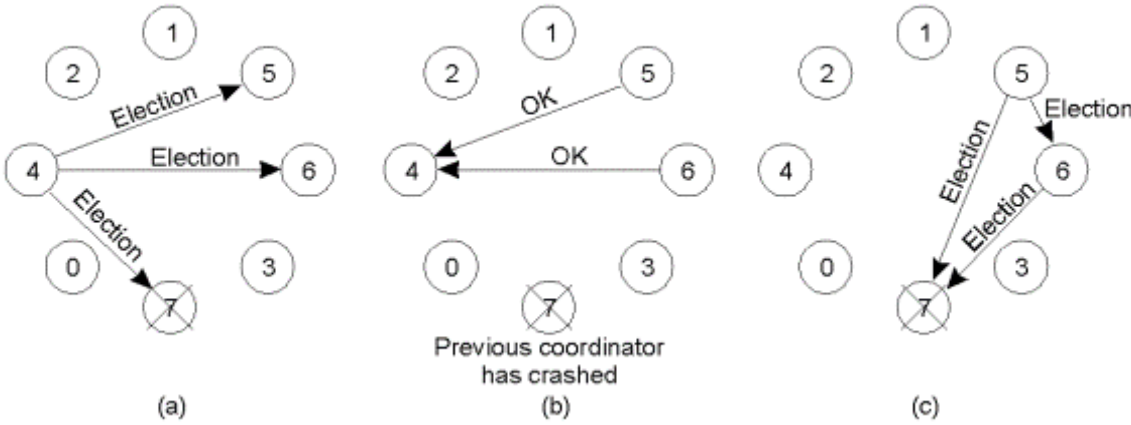
1. All of *Q's* successors have returned a *DONE*
2. *Q* has not received any message between the point it recorded its state, and the point it had received the marker along each of its incoming channels.

Otherwise *Q* sends a *CONTINUE* message.

2. Election Algorithms

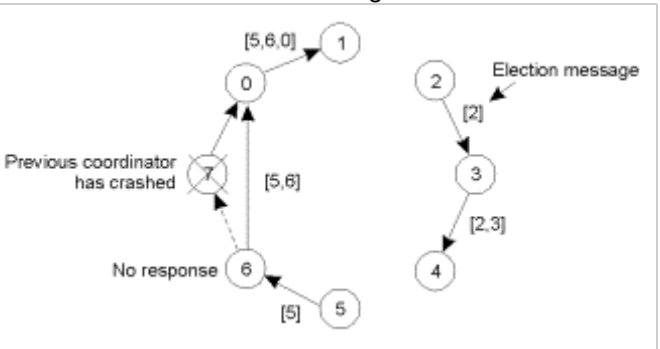
i. The Bully Algorithm

- When any process P notices that the coordinator is no longer responding it holds an election by:
    1. P sends an ELECTION message to all processes with higher numbers
    2. If no one responds, P wins and becomes the coordinator
    3. If someone responds, then it takes over and P stops.
- If a process ever get an ELECTION message from one of its lower-numbered colleagues it sends back an OK message and takes over
- If a temporarily down process comes back up, it holds an election
- COORDINATOR message sent by whoever wins election



ii. A Ring Algorithm

- No token used -- assume processes are logically/physically ordered so each process knows its successor
- Any process noticing a coordinator not responding sends an ELECTION message w/its own proceess number to its successor
- If successor not avai., then sender skips them and goes to next number until they find a running process
- At each step, the sender adds its own process number to the list (making itself a candidate)
- When sender receives an incoming message with its own process number, it changes the message type to COORDINATOR and circulates again - the coordinator is the list member with the highest number
- COORDINATOR message circulates once and then is removed



3. Mutual Exclusion

i. Centralized

- One process elected as coordinator
- Whenever a process wants to enter a critical region, it sends a request message to the coordinator

- Coordinator replies granting permission if free - otherwise could send a denied message, or just block
- Process sends release message to coordinator when leaving critical region

ii. Distributed

- **Contention-Based**
- **Token-Based**
  - **Ring Structure**
  - **Tree Structure**
  - **Broadcast Structure**
- **(Ricart and Agrawala)**
  - Requires that there be a total ordering of all events in the system (could use lamport for timestamps)
  - When a process wants to enter the critical region, it sends a message w/the name of the ciritcal region, its process number and the current time
  - Message sent to all processes (assumes message transmission reliable)
  - Actions of the message receiver:
    i. If not in the critical region and not wanting to enter it, send back an OK message
    ii. If already in the critical region, does not reply (queues request)
    iii. If wants to enter the critical region, but not yet done so, compares timestamp of incoming message with the one it sent out. If other message earlier, sends OK -- if not, queues message & sends nothing
  - Sending processor waits for Oks from all other processors. When it exits the cr, it sends Oks to everyone.
  - A crashed process will be interpreted as denying requests -> can fix by forcing a reply to all messages (OK or deny)
  - Either group communication needed, or each process must maintain membership list (method best for small groups with static membership)
  - Could change so that messages are sent to only a majority of the processes to reduce messages

iii. Token Ring

- Given a bus network, with no ordering of processes - > construct logical ring in software
- Token is generated for the ring, that is passed from node to node in point to point messages
- Upon receiving the token the process checks to see if it needs to enter a critical region -- may only access one on a given token -- otherwises passes the token
- Difficult to detect if token is ever lost, and if so must be regenerated
- Could require receipt of token ack -- that way, if process dies, token can be passed on (requires each process know ring configuration)

iv. Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

VII.Naming and location services (Ch 4)

1. **NAMING**

- access points are entities that provide access to other entities – the name of an access point is an address
- location independent name provides a name for an entity independent from its addresses
- properties of a true identifier: 1-refers to at most 1 entity, 2-each entity is referred to by at most one identifier, 3-an identifier always refers to the same entity
- name space organized as directed graph w/two node types.
  - leaf node-> represents named entity; no outgoing edges.
  - directory node-> multiple outgoing edges
  - if the first node in a path name is the root node, then the path is absolute, otherwise relative
- closure mechanism – selecting the initial node in a name space from which to start name resolution
- two approaches for aliases: 1-allow multiple absolute paths to refer to the same node in a naming graph . 2-represent an entity by a leaf node, which stores an absolute path name. (aka symbolic links)
- a mounted file system allows a directory node to store the identifier of a directory node from a different name space
- mounting a foreign name space requires: 1-name of an access protocol,2-name of the server,3-name of the mounting point in the foreign name space
- another way to merge name spaces is to add a new root node and to make the existing root nodes its children
  - this requires that existing names be changed
  - eventually have performance problems b/c root node of merged name space must maintain a mapping of identifiers of old root nodes to their new names
- Implementation of a Name Space: 3 layers
  - global layer: formed by highest level nodes(the root and its children). directory tables of these nodes rarely change. high availability critical. result of lookup operations remain valid for extended periods -> caching. performance requirements met with replication.
  - administrational layer: directory nodes that are managed within a single organization. lookup results should be returned w/in a few miliseconds, updates processed quicker than global layer. performance requirements met with high-perf. machines to run name servers.
  - managerial layer: nodes that typically change regularly. ie nodes for hosts in the local network, nodes representing shared files. nodes are maintained by sys admins as well as individual users

Name Resolution 2 methods

- iterative: client's name resolver give complete name to root server. root server resolves as much as possible, then passes back to client. name resolver then contacts secondary name server. name servers in global layer only support iterative. caching only with client's name resolver.
- recursive: instead of passing resolution result back to client, name server passes it on to the next name server. higher performance demand on name servers. caching better than iterative. reduced communication costs.

DNS

LOCATING

- **problems** using traditional naming for mobile entities:
  - 1-recording the address of the new machine is not a local update – violates assumption of efficient operations in managerial layer.
  - 2-turning into a symbolic link adds another step to lookup.
  - 3-original name is not allowed to change
- **broadcasting solution**: a message containing the identifier of the entity is broadcast to each machine and each machine checks for the entity. inefficient when network grows network bandwidth wasted, machines interrupted by requests they cannot answer.
- **multicasting solution**: restricted group of hosts receives request. can be used in point to point networks (the internet)
  - groups are identified by a multicast address.
  - network layer provides best effort to deliver message to all group members.
  - multicast address can be used as a general location service for multiple entities
  - multicast address can also be associated with a replicated entity and used to locate the nearest replica

forwarding pointers: when an entity moves from A to B, it leaves behind a reference to its new location at B.

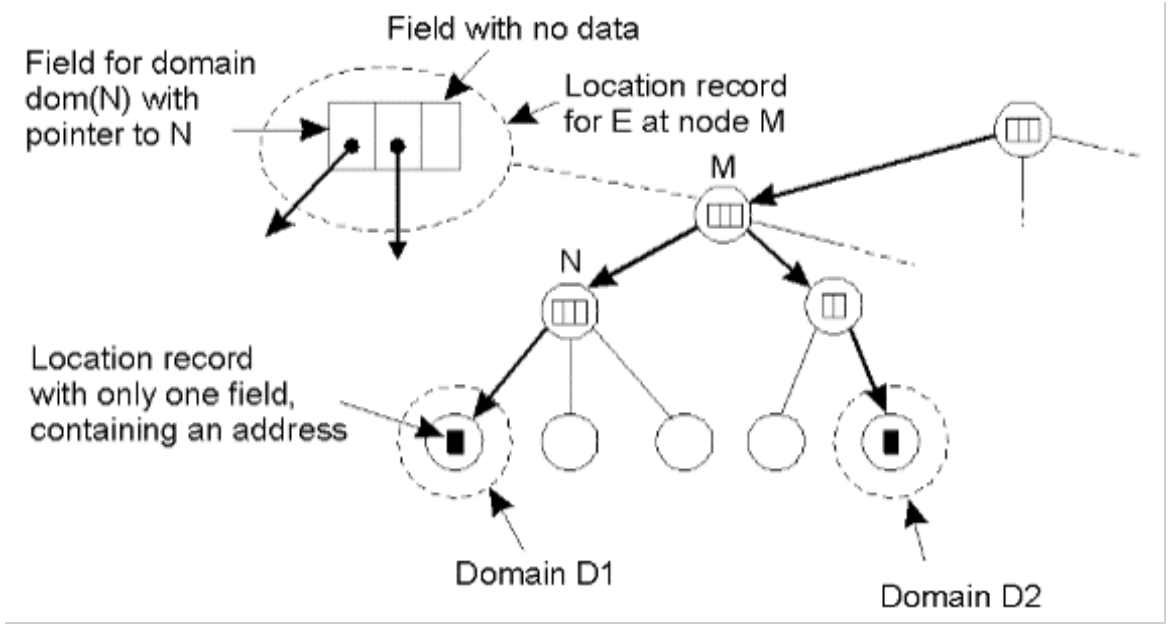
- chain can become so long that location is too expensive
- all intermediate hosts must maintain their pointers
- vulnerable to broken links->if a pointer is lost, the entity cant be located
- can short cut chain by sending response back to proxy initiating the invocation with objects current location (either send along the chain and update all pointers, or directly to that object)

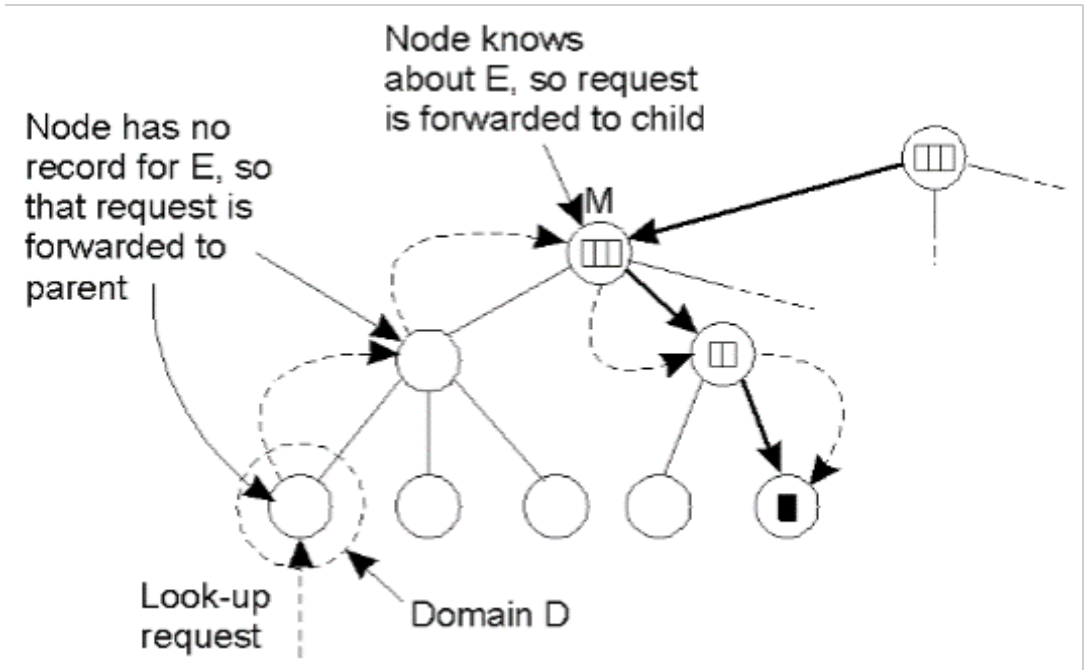
home based approach: each mobile host has a fixed IP. all communication to that IP address is directed to the hosts home agent. home agent located on the LAN corresponding to the network address contained in the mobile hosts IP address. when moving to another network, the mobile hosts requests a temp addr. registers the care-of address with home agent.

- if host not on network when request arrives, request is tunnelled, and requestor is notified
- if fixed home location unavailable, cannot locate entity. high cost to contact if mobile host relocates permanently-> can register home agent at a traditional naming service

hierarchical approach: network divided into a collection of domains (single top level domain for whole network)

- each domain D has a directory node dir(D) that keeps track of entities in the domain.
- the directory node of the top domain (root node) knows about all entities
- each entity in a domain has a record in the directory node dir(D)

- in a leaf domain D, the directory node N has a record with entity E's current address
- the next directory node N' for higher-level domain D' that contains D will have a location record for E that has a pointer to N.
- the parent node of N' has an entry for E with a pointer to N'
- **replicated entities with multiple addresses** in D1 and D2: the directory node of the smallest domain with both D1 and D2 will have two pointers, one to each domain.
- 

- **lookup request**: a client wanting to locate E issues a request to its directory node D. request is forwarded to the next highest parent until one knows the location of E
- 

- **updates**:

        - 1-(top-down) node wishing to create a record forwards up chain until reaching a node that already has a record for E. then it creates an entry for E that points to the child node that forwarded the request. then the process goes down the chain.
        - 2-(bottom-up) node that initiates the creation request creates a record first, then forwards to parent. this allows immediate lookup operations in local domain even if parent unavailable.
    - deletes work bottom-up
    - caching only effective if D is the smallest domain that E regularly moves in -> can start lookup operation at dir(D)
        - caching could be improved if dir(D) stores E's actual address.
    - root node can become bottleneck (too many requests) -> partition root-node and other high-level directory nodes into subnodes
        - where to place subnodes?

VIII.Memory consistency and data coherency (7.2,7.3)

Memory Consistency (Tanenbaum Ch 6)

Replication:

- Two main reasons: reliability and performance -> also scalability

Consistency Models

- **Data Centric**
    - **Strict Consistency**: *any read on a data item x returns a value corresponding to the resilt of the most recent write on x.*
        - Assumes existance of global time (impossible in distributed system)
        - All writes instantaneously visible to all processes
    - **Linearizability**: *(same as sequential). Also if tsOP1(x) < tsOP2(y) then operation OP1(x) should precede OP2(y) in the sequence.*
        - Ordering is done according to a set of synchronized clocks
    - **Sequential Consistency**: *the result of any execution is the same as if the read and write operations by all processes on the data store were executed I some sequential order and the operations of each individual process appear in this sequence in the order specified by its program*
        - All processes must see the same interleaving of operations
        - Poor performance
    - **Causal Consistency** : *writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines'*
        - Distinguish between events that are causally related or not
        - Concurrent operations - not causally related
        - Requires keeping track of which processes have seen which writes
    - **FIFO Consistency** : *writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.*
        - Called PRAM (Pipelined RAM) with distributed shared memory. Writes from a single process can be pipelined
        - Easy to implement -> two or more writes from a signle source must arrive in order. All writes from different processes are concurrent
    - **Weak Consistency** :

        - *Access to synchronization variables assoc. with a data store, are sequentially consistent*
        - *No operation on a synch. Var. is allowed to be performed until all previous writes have completed everywhere*
        - *No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed*
            - Synchronization variable - has a single associatied operation which synchronizes all local copeies of the data store. When the data store is synchronized, all local writes by process P are propagated to other processes and writes by other processes are brought into Ps copy.
            - Enforces consistency on a group of operations, not individual reads and writes
            - Problem: data store does not know if synchronization is being performed because the process is either finished writing shared data, or is otherwise about to start reading data
- **Release Consistency** :
    1. *Before a read or write operation on shared data is performed, all pervious acquires done by the process must have completed successfully*
    2. *Before a release is allowed to be performed, all previous reads and writes done by the process must have been completed*
    3. *Access to synchronization variables are FIFO consistent*
        - Defines two kinds of synchronization variables: acquire and release
        - Programmer responsible for inserting code in the program to do the operations explicitly
        - Barrier - synchronization mechanism that prevents any process from starting phase N+1 of a program until all processes have finished phase n
        - Lazy release consistency : does not push values at the time of a release -- waits until an acquire by a given process to get the most recent values.
            - Solves the problem of ineffeciency with eager release consistency (loop example)

- **Entry Consistency** :
    1. *An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed w.r.t that process*
    2. *Before an exclusive mode access to a synchronization variable by a process is allowed to perform w.r.t that process, n other process may hold the synchronization variable, no even in nonexclusive mode*
    3. *After an exclusive mode access to a synch. Var. has been performed, any other process's next nonexclusive mode access to that synch. Var. may not be performed until it has performed w.r.t that variables owner*
        - Also requires the use of acquire and release
        - Requires each shared data item to be associated with some synchronization variable

- When an acquire is done on a synch.var. only those data guarded by that s.v. are made consistent
- Reduces the overhead of acquire and release operations
- Increases parallelism by allowing multiple critical sections involving disjoint shared data to execute simultaneously
- Downside: more complicated programming
- Each sv. Has an owner (its last accessing process) -- owner can access repeatedly w/o sending messages to the network
- Non-owner must ask current owner for access

- **Client Centric**
  - **Eventual Consistency**
    - Tolerate high degree of inconsistency -- if no updates for a long time, replicas will gradually become consistent
    - Write-write conflicts easier to resolve when assuming small group of processes
  - **Monotonic Reads**: *if a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent one*
    - Process will never see an older version of data later in time
  - **Monotonic Writes**: *a write operation by a process on a data item x is completed before any successive write operation on x by the same process*
    - The copy on which a successive operation is performed, reflects the effect of a previous write operation by the same process
  - **Read Your Writes**: the effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process
    - Write operation is always completed before a successive read operation by the same process
  - **Writes follow reads**: a write operation by a process on a data item x following a previous read operation on x by the same process, is guaranteed to take place on the same or a more recent value of x than was read.
    - Updates propagated as the result of previous read operation

Data (cache) Coherence (Chow Ch 7)

Two types of Coherence Protocols

- **Write Invalidate:**
  - When a write operation occurs, the holders of the shared block are told to invalidate their copies
  - This forces a cache miss the next time the block is references
  - **Read Hit:** the processor reads the data block and continues
  - **Read Miss:**
    - Occurs if the processor doesn't hold the block, or it is invalid
    - The directory is consulted to transfer the block from the memory of one of the copy holders
  - **Write Hit:**
    - To an exclusive cache block: allowed to proceed and processor continues

        - If shared-read-only block:
            - Send message to the directory that invalidates all cache copies according to its P-bits vector
            - When invalidates are completed, the processor writes data and sets the exclusive bit
            - Processor is sole owner until reads arrive from other processors
    - **Write Miss:** similar to write hit, except a block copy is transferred to the faulting processor by the directory server after the invalidation
    - Efficient when multiple writes between reads -> writes are exclusive, or only require invalidation at a few processors
    - Invalidations cause additional misses when a block is shared and frequently accessed by many processors
- **Write update:**
    - Propagate new block to all copy holders
    - Generates high traffic (updates sent even if unneeded)

Two classes for the implementation of memory coherence control:

- **Directory-Based**
    - Directory maintains the state of each memory block and is resonsible for the invalidations and udates
    - Can either be centralized or distributed
- **Snooping Cache**
    - Hardware implementation of fully distributed directory
    - With a common bus, each cache controller can monitor all memory accesses
    - Because all data and control are broadcast on the bus, each processor snoops to invalidate or update its own cache
    - Common bus could be a bottleneck. Solutions:
        - Coherence control traffic (updates, invalidates) seperated from regular data access in another bus
        - Processors clustered on segmented buses and coherence maintained intra-cluster and inter-cluster
        - Bus contention reduced w/special topologies or networks

Consistency and Replication

Eventual Consistency

- Offered by data stores where simultaneous updates are rare -- and when they do happen they can be easily resolved
- Most operations involve reading data
- In the absence of updates, all replicas converge toward identical copies of each other
- This model works only when clients always access the same replica

Replica Placement

- Permanent replicas: initial set of replicas that constitute a distributed data store
    1. Files of a single site are distributed across servers on a single LAN
    2. Web site is copied to limited number of servers (mirrors) spread out across the internet
- Server Initiated Replicas: exist to enhance performance, created at the initiative of the data store owner
    - **Push caches :** temporary replicas in regions with high # of requests
    - **Dynamic allocation** : each server tracks access counts per file, and where the acceses originate from. Replication and deletion threshold determine

when a given file will be replicated on another server or deleted. Between the thresholds the file can only be migrated.

- Client-Initiated replicas (aka client caches): client temporarily stores a copy of data that it has just requested. Used only to improve access times

Update Propagation (General Issues)

- What will be propagated:
    1. Update notification - invalidation. Little bandwidth used. Work best when many updates compared to reads (read:write is low)
    2. Transfer data - useful when read:write is high -> high probability the data will be read before the next update is high.
    3. Propagate the update operation (**active replication**) - assumes each replica is represented by a process that can keep its data up to date through operations. Minimal bandwidth costs. Requires more processing power, especially for complex operations
- Push or Pull:
    - **Push-based approach**:
        - updates propagated without replicas asking.
        - Often between permanent and server-initiated replicas.
        - Used when replicas need a high degree of consistency
        - Used when replicas are shared by many clients -> reads dominate writes (read:update high)
        - Server must track all clients -> high overhead
    - **Pull-based approach:**
        - Client or replica requests updates from server
        - Often used by client caches
        - Efficient when read:update low
        - Response time increased with each cache miss
        - Client must poll server
    - **Lease:**
        - Server promises to push updates to the client for a specified time. On expiration, client must poll for updates, or renew lease.
        - Could be age-based (depending on the last time the data was modified)
        - Could be renewal-frequency based (long-lasting lease to a client whose cache is often refreshed, shorter lease for client that asks for item occaisionally)
- Unicast or Multicast:
    - Multicast often cheaper - can be combined well with push approach

Epidemic Protocols

- Main advantage is scalability -> little synchronization between processes
- Infective server holds update and is willing to spread
- Susceptible server does not have update
- Removed server has update, but cant/wont spread it
- Anti-Entropy:
    1. P only pushes its own updates to Q
    2. P only pulls in new updates from Q
    3. P and Q send updates to each other
- Pushing updates bad w.epidemic protocols, b/c w/many infective servers the probability of finding a susceptible one is low (one server could remain susceptible for a long time).

- Rumor spreading: a newly updated server selects another at random and tries to push its update at random. If the server it selects already has the update it may lose interest in spreading with probability 1/k
- Deletion: record deletion as another update (spread death certificates)

Primary-Based Consistency Protocols

- Each data item x in the data store has an associated primary -> responsible for coordinating write operations

1. **Remote Write**
   - (**Pure**) all read and write operations carried out at a remote server
   - **Primary-Backup**

   Processes perform read operations on locally available copy, but forward write operatoins to a primary one
   Primary performs update on its copy, forwards updates to backups who then send acks to the primary
   Finally primary sends ack to the initial process
   Problem: client requesting the update may be blocked for a long time
   Implement sequential consistency
2. **Local-Write**
   a. **(Fully Migrating) No replicas -** when a process wants to operate on a data item, it must be transferred to the process
      - Difficult to track location of entity
   b. (Primary Backup) - primary copy migrates between processes
      - Allows multiple successive writes locally
      - Used with mobile computers in disconnected mode

Replicated-Write Consistency Protocols

1. **Active Replication**
   - Each replica has a process that carries out updates
   - Updates must be in the same order everywhere -> totally ordered multicast
   - Central sequencer used instead of lamport (b/c of scalability problems)
   - Each operation forwarded to the sequencer where it gets a sequence number and then is forwarded to replicas'
   - Solution to **replicated invocations** : replication aware communication layer for replicated objects to execute upon. Ensures only a single request from all the replicas of B and only a single reply from all the replicas of C.
2. **Quorum Based**
   - Require clients to request and acquire the permission of multiole servers before reading/writing
   - **Read quorum and write quorum conditions:** Nr + Nw > N, Nw > N/2. N replicas of file

Cache Coherence Protocols

- Many solutions based on support from hardware
- **Software issues:**
  1. When are inconsistencies detected? (**coherence detection strategy) :**
     - Static - compiler analyzes before exection, inserts instructions to avoid inconsistencies
     - Dynamic - runtime detection
       1. When a cached item is accessed, must verify consistent with server version

        2. Allow transaction to proceed during verification (abort later if inconsistent)
        3. Verify consistency when committing transaction
2. How are caches kept consistent? (**coherence enforcement):**
    - Send invalidation
    - Send update
3. Modification of cached data:
    - Read-only
    - Write through - improved performance, all operations local
    - Write back - more improved, mutliple writes before update propagation

IX.Security: cryptography, mutual authentication protocols, distributed access control

Created with Microsoft Office OneNote 2003
One place for all your notes