**What is a Distributed System?**

A distributed system consists of a collection of distinct processes which are spatially separated and which communicate with one another by exchanging messages

A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process

**Message based communication**

* whose format and meaning are specified by a communication protocol
* that is transported from its source to its destination by a communications network
* NOT TRUE assumptions
  + network is reliable
  + latency is zero
  + bandwidth is infinite
  + network is secure
  + topology does not change
  + only one administrator
  + transport cost is zero
  + network is homogenous
* properties:
  + connection-based vs connectionless
    - connection-based:
      * processes must setup the channel before exchanging data – analogous to the telephone network
    - connectionless:
      * processes need not set up the channel, can exchange data immediately – analogous to mail
  + reliable vs unreliable
    - reliable:
      * ensure that the data sent is delivered to the respective destination
        + under some assumptions
        + if not, the communicating processes are notified
    - unreliable:
      * it is up to the communication processes to detect the loos of messages and proceed as required by the application
    - duplication:
      * “generates” duplicates: the channel may deliver duplicated messages to the destination – it is up to the recipients to detect the duplicates
      * no duplicates: the channel ensures that it delivers each message to its recipients at most once
  + unsures order (or not)
    - order
      * ensures that the data is delivered to its recipients in the order in which it was sent
    - unordered
      * if it is important to preserve order, it is up to the applicationo to detect that the data is out of order and if necessary to reorder it
    - order and reliability are orthogonal
  + message-based vs stream-based
    - message
      * channel supports the transport of messages – sequences of bits processed atomically – analogous to mail
    - stream
      * channel does not support messages. Essentially, it works as a pipe for a sequence of bytes – analogous to Unix pipes
  + with or without flow control
    - flow control
      * prevents “fast” senders from overflowing with data “slow” recipients
        + it does not necessarily mean that the sender has more computing power than the receiver
  + number of ends of the channel
    - unicast (or point-to-point): only two end
    - broadcast: all nodes in the “network”
    - multicast: subset of nodes in the “network”
  + identification
    - “name” of the process itself
    - “name” of the channel endpoint (ex. phone number)

**Advantages of Distributed**

* sharing resources
* access to remote resources
* performance
  + can use multiple computers to solve the problem
* scalability
  + load
  + geographical
  + administrative
* fault tolerance
  + reliability
  + availability

**Scalability**

* distributed algorithms:
  + system global state is unkown (relativity)
    - can use only information locally available
  + correctness must be ensured even in the presence of faults
  + no single physical clock
* asynchronous communication
* replication and caches:
  + reduces communication latency
  + allow distributed processing
  + raises consistency problem

**Challenges**

* partial failures
  + some components may fail, while others continue to operate correctly
* IPC latency
  + IPC across the network has an unpredictable latency, which usually cannout be bounded
* No global time
* No shared physical memory and distinct address spaces
  + pointers are meaningful only in the context of the respective address space
* heterogeneity
  + has several facets
* lack of security and trust

**Internet Protocols**

* application: specific communication services
* transport: communication between 2 (or more) processes
* network: communication between 2 computers not directly connected with each other
* interface: communication between 2 computers directly connected
* UDP:
  + channels transport messages
    - basically, its API supports two operations: send() and receive()
    - each message is transmitted by invoking send() once
    - if delivered, the message will be delivered atomically, in a songle invocation of receive()
  + datagrams have a maximum size of 65535 bytes
    - application may have to split the data to send in datagrams before transmitting, and reassemble the data from fragments after receiving
  + connectionless:
    - pro: allows a process to start transmitting data immediately
    - con: requires the specification of the other channel endpoint on every invocation of send()
  + no reliability guarantees:
    - datagrams may be lost or even duplicated
    - if the application cannot tolerate the loss, or the duplication, of datagrams, it will have to detect and recover from such an event
  + no flow control:
    - receiver may be flooded with requests and run out of resources (ex. buffers) to receive other messages
  + supports multicast
    - by invoking send() once, it is possible to send a copy of a given message to several processes
* TCP:
  + stream channels
    - similar to pipes on Unix-like system, except:
      * can be used for communication between processes in different computers
      * bidirectional channels (ex. it is possible to send data in both directions simultaneously)
  + although we can use send() and receive() to exhcange data:
    - TCP does not ensure the “separation” between bytes sent by invoking two send() calls
    - write() and read() match better TCP semantics
    - actually, the Java API uses the many “stream” classes to exchange data via TCP
  + connection-oriented, communication has 3 phases:
    - connection set up
    - data exchange
    - connection tear down
  + ensures reliability (both loos and duplication of data):
    - prevents data loss because of insufficient resources
    - nevertheless, TCP may be vulnerable to denial-of-service attack
  + channels have only two endpoints, supporting communication between only two processes
  + channels on the same computer may have the same port number:
    - a TCP channel is identified by the pairs (IP address, TCP port) of its to endpoints
    - this allow the concurrent service of several clients in client-server applications

**TCP vs UDP**

* why not always TCP?
  + it provides “more” than UDP
* can you pay the cost?
  + connection must be set up before data exchange
  + recovery of a lost segment affects those that follow it
    - TCP ensures in-order delivery
* some application cannot (ex. internet telephony)
  + it is very sensitive to delays
  + but can tolerate some loss
  + TCP provides a service (recovery of lost data) that it (internet telephony) does not need, at a cost that may be too high

**Streaming applications**

* a multimedia application in which contents may be played before it is completely received
  + the application needs not download the full contents before starting to play it
* Classes:
  + streaming stored audio and video (ex. youtube)
  + streaming live audio and video (internet radio, television)
    - contents is generated asa it is being sent
  + real-time interactive audio and video (ex. skype, zoom)
    - two-way communication
* Requirements:
  + bandwidth: do not tolerate large variations
  + packet delay: and also its jitter (ex. its variation) are particularly critical
  + packet loss: ratio is not so stringent

**Internet Protocols and Streaming application**

* but the internet is designed on the best-effort principle
  + it does not provide any guarantees, especially regarding packet-loss ration, bandwidth, packet delay and its jitter
* “tricks people play”
  + bandwidth: use compression
    - encoding standards support compression
  + delay and its jitter
    - for non-RT application we can use buffering
    - for RT applications we can reduce the jitter by engineering a delay
  + packet loss:
    - streaming apps can use plain TCP, as long as the buffer are large enough
    - interactive RT apps use forward-error-correction (FEC)
      * encoding standards often support FEC
* rely on the end-to-end argument

**End-to-end argument**

* a design principle for layered systems and states:
  + if you have to implement a funciton, end-to-end does not implement it on the lower layers unless there is a compelling performance enhancement
* why is this relevant for distributed systems?
  + distributed applications are often layered
  + on the Internet, you can choose between TCP and UDP
* this is a design principle, not a physics law

**Dave Andersen’s Algorithm**

* do you need everything TCP provides?
  + if yes, choose it
* if not: can you pay the cost?
  + if yes, use it
* if not:
  + use UDP
  + implement what you need on top of UDP

**Remote Procedure Call (RPC)**

* typically implemented on top of transport layer
* Message-based programming with send()/receive() primitives is not convenient
  + depends on the communication protocol used (TCP vs UDP)
  + required the specification of an application protocol
  + akin to I/O
* function/procedure call in a remote computer
  + is a familiar paradigm
  + eases transparentcy
  + is particularly suited for client-server applications

**RPC Stub Routines**

* ensure programming transparency
  + client invokes the client stub – a local function
  + remove function is invoked by the server stub – a local function
* the stub routines communicate with one another by exchanging messages
* client stub:
  + request:
    - assembles message: parameter marshalling
    - sends message, via write()/sendto() to server
    - blocks waiting for response, via read()/recvfrom()
      * not in the case of asynchronous RPC
  + response:
    - receives responses
    - extracts the results (unmarshalling)
    - returns to client
      * assuming synchronous RPC
* server stub:
  + request:
    - receives message with request, via read()/recvfrom()
    - parses message to determine arguments (unmarshalling)
    - calls function
  + response:
    - assembles message with the return value of the function
    - sends message, via write()/sendto()
    - blocks waiting for a new request

**RPC Dispatching**

* often, RPC services offer more than one remote procedure
* the identification of the procedure is performed by the dispatcher
  + this leads to a hierarquical name space (service, procedure)

**Transparency:**

Platform Heterogeneity

* Problems: at least two
  + different architectures use different formats
    - 1’s-complement vs 2’s-complement
    - ASCII vs UTF
  + languages or compilers may use different representations for composite/compound datag-structures
* Solution: mainly two
  + standardize format on the wires
    - pro: needs only two conversions in each platform
    - con: may not be efficient
  + receiver-makes-right

Addresses as Arguments

* Issue: the meaning of an address is specific to a process
* Solution: use call-by-copy/restore for parameter passing
  + pro: works in most cases
  + con: complex
    - same address may be passed in different arguments
  + con: inneficient
    - for complex data structures (ex. trees)

Transparency in the Presence of Faults

* problem: what if something breaks?
  + the client cannot locate the server
    - RPC can return an error (like in the case of a system call)
  + the request-messae is lost
    - retransmit it, after a timeout
  + the response-message is lost
    - must use request identifiers (sequences of nums)
    - must save most recent responses for replay, if the request is not idempotent
  + server crashes
    - was the request processed before the crash
  + client crashes
    - need to prevent orphan computations (ex. on behalf of a dead process)
* issue: a client cannot distinguish between loss of a request, loss of a response or a server crash
  + the absence of a response may be caused by a slow network/server

**RPC Semantics in the presence of faults**

* question: what cna a client expect when there is a fault?
* answer: depends on the semantics in the presence of faults provided by the RPC system
  + at-least-once
    - RPC returns a value: RPC has executed at least once, but could have executed several times
    - RPC raises exception: RPC may have executed several times or none at all
    - if after several tries does not get a reply
      * prevents client form blocking forever
    - client stub: keeps resending request until
      * either gets a reply
      * or get “tired of waiting” and gives up
    - server stub: executes request and sends reply
  + at-most-once
    - RPC returns a value: RPC has executed exactly once
    - RPC raises exception: RPC executed either once or not at all
    - client stub: send request once
      * return reply or raise exception, upon timeout or broken connection (if using TCP)
    - server stub: executes request and sends reply
    - issue: it is not robust against communication faults, if using UDP
    - implementation with UDP:
      * client stub needs to retransmit request, if it does not receive the reply
      * the RPC middleware must ensure that the procedure is not executed more than once
        + RPC requests include an id, managed by the RPC middleware
        + server stub keeps a table with replies
  + exactly-once
    - not always possible to ensure this semantics, especially if there are external actions that cannot be undone
    - problem: in the case of external actions (ex. file printing), it is virtually impossible to ensure exactly-once semantics
    - server policy (one of two):
      * send an ACK after printing
      * send an ACK before printing
    - client policy (one of four):
      * never resend the request
      * always resend the request
      * resend the request when it receives an ACK
      * resend the request when it does not receive an ACK

**At-least-once vs At-most-once**

* lost response (at-least-once):
  + client stub may send request more tha nonce
    - if procedure is not idempotent:
      * RPC signature/prototype must include a request id as argument, managed by application
      * server/application must keep table with responses previously sent
* client crash:
  + client may or not know it owns the lock, both before crash and after reboot
    - server (ex. application) may have to handle this
      * using leases, rather than locks
    - again, the RPC semantics are irrelevant
* server crash:
  + at-most-once:
    - client does not know if server granted it the lock
      * depends on when the server crashed
    - client, not RPC, may ask the server (or just retry)
      * server needs to remember state across reboots
      * ex. store locks state on disk
  + at-least-once:
    - client does not know if server granted it the lock, like for at-most-once
      * server needs to remember state across reboots
    - server may also run the procedure several times
      * client stub may send several requests before giving up
    - if requests are not idempotent:
      * RPC prototype must include request id
      * server needs to remember previous requests across reboots (if requests are not idempotent)
* Conclusion of at-least-once vs at-most-once:
  + simple services/applications:
    - if requests are idempotent
      * at-least-once
    - when requests are not idempotent
      * at-most-once
    - issue: some requests may be idempotent whereas otherss are not
      * need to pick one semantics
  + more sophisticated services/applications:
    - no clear advantages: the service/application itself may have to take special measures

**Thread and memory consistency**

* Memory consistency errors occur when different threads have inconsistent views of what should be the same data
  + on “thread.start” new threads have visibility on data from preceding thread
  + after “thread.join” thread gets visibility on data from ended thread
* writing under a lock and reading after acquiring the same lock also ensures visibility

**Barrier**

A synchronization aid that allows a set of threads to all wait for each other to reach a commom barrier point

**Hierarquical Locking**

Lock order of acquisition is relevant

In some cases we can have a mutable structure that holds cells with internal locks. Typically the big structure lock should only be held for a short time.

**Concurrency**

* there are several reasons for using concurrency
  + performance (more on servers)
  + usability (more on clients) – basically performance
* the goal:
  + overlap I/O with processing
  + take advantage of multiple cores
* Client-side
  + a web page may be composed of several objects
  + a browser can render some objects, while it fetches others over the Web
* server-side
  + may serve several requests simultaneously
* CPU-bound vs I/O bound
  + regardless, a process execution alternates between time intervals of computation, by the CPU, and idle time intervals, waiting for I/O
  + cpu-bound:
    - longer intervals of computation
  + I/O bound
    - smaller intervals of computation but more frequent

**Iterative Web Server**

* has only one thread
* processes a request/connection at a time
* each step/stage has one operation that can block
  + stat() is required because of the HTTP header fields “size” and “last modified”
    - but open() may also block /ex. file name resolution or checking permissions may require disk access)
  + server cannot process other requests while blocked
* such a server can process only a few requests per time unit

**Multi-threaded Server**

* each thread processes a request
* when one thread blocks on I/O
  + another thread may be schedules to run in its place
* a common pattern is:
  + one dispatcher: thread, which accepts a connection request
  + several worker: threads, each of which process all the requests sent in the scope of a single connection

**Event-driver server**

* the server executes a loop, in which it:
  + waits for events (normaly I/O)
  + processes these events sequentially (ex. one after the other)
* blocking is avoided by using non-blocking I/O operations
* need to keep a FSM for each request
  + loop dispatches the event to the appropriate FSM
* kown as the state machine approach

**Thread-based vs Event-based debate**

* thread-based
  + ease of programming:
    - appears simple
      * structure of each thread similar to that of an iterative server
      * need only to ensure isolation in the access to shared data structures
    - could use only monitors (ex. synchronized method in Java)
      * not so easy: there are some implications in terms of modularity
      * possibility of deadlocks
      * performance may suffer
        + the larger the critical sections, less concurrency
        + but the main reason for concurrency is performance
  + performance
    - as number of threads increases, the system throughput increases, then levels-off and finally dives
    - in one experiment, the culprit is context switching
      * both the overhead itself
      * and its effect on the cache
    - other issues may arise:
      * thread creation/termination
      * memory management (ex. stack and heap)
      * synchronization to avoid race-conditions
* event-based
  + ease of programming
    - programmer need to:
      * break processing according to potentially blocking calls
      * manage the state explicitly (using state machines), rather than relying on the stack
    - the structure of the code is very different from that of the iterative server
    - no nasty errors like race conditions, which may be elusive
      * but only if we use a single thread
    - many complain about lack of support by debugging tools
    - also complain that it leads to pporly structured code
      * the author points out that the issue is preemption rather than multithreading
      * actually, the problem is lack of atomicity
        + with multiple cores, we can have race conditions, even if there is no preemption
  + performance
    - requires non-blocking (or asynchronoous) I/O operations
      * otherwise, may use multiple threads for emulation
    - allows user level scheduling
      * dispatcher may choose which even to handle next
    - as number of requests in a queue increases, throughput increases until it reaches a plateau
    - multiple threads needed to achieve paralellism in multicore/processor platforms

**Thread-based vs event-based: performance**

* the debate was somewhat “muddled” by implementations tht were less than optimal
* actually, at the technical level this is very similar to the debate about user-level vs kernel-level threads
* user-level threads are more efficient than kernel-level threads
  + function calls vs system calls
  + but performance suffers if OS does not provide non-blocking I/O
  + worse, there are some unavoiidable blocking (ex. page faults)
* need kernel-level threads in order to take advantage of multiple processors/cores

**Thread Implementation**

Threads can be implemented:

* directly by the OS (kernel-level threads)
  + the kernel supports processes with multiple threads
    - the kernel’s scheduler allocates cores to threads
  + the OS keeps a “threads table” with information on every thread
    - usually a process’ control block points to its own threads’ table
  + all thread management operations, such as thread creation, incur a system call
* by user-level code (ex. library) (user-level threads)
  + the kernell is not aware of the existence of threads at user-level
    - threads are implemented by a user-space library
    - the OS needs not support threads
  + implementation:
    - the threads’ library must provide functions for:
      * thread creation or destruction
      * thread synchronization
      * yield a core to other threads
    - the library is responsible for thread switching and keeps a thread’s table
    - the wrapper-functions of some system calls that may block, have to be modified
      * to prevent other threads from blocking
    - some issues:
      * how to make non-blocking system calls?
      * what about page-faults?
      * how to prevent a thread from never yielding a CPU/core

**User-level vs Kernel-level threads**

* pro: the OS needs not support threads
* pro: the kernel is not involved in most operations (ex. thread creating/destruction)
* con: page-fault by one thread will prevent the other threads from running (the single kernel-level thread is put in the WAIT state)
* con: cannot be used to exploit parallelism in multicore architectures

**Hybrid implementation**

* idea: multiplex user-level threads on kernel-level threads
* the kernel is not aware of the existence of user-level threads
* actually for better results:
  + the user-level scheduler should give hints to the kernel-level scheduler
  + the kernel-level scheduler should notify the user-level kernel about its decision
* the library maps user-level threads to kernel-level threads
  + the number of user-level threads may be much larger than that of kernel-level threads

**Thread-based vs event-based: conclusion**

* pure thread-based and event-based designs are the extremes in a design space
* threads are not as heavy as processes, but they still require resources
  + you many want to bound their number
* if you want more parallelism, you nede to use both:
  + threads: virtually all processors now-a-days are multicore
  + events: to limit the number of threads, and therefore their overhead
* there are many framworks supporting event-driven designs
  + java itself offer NIO (non-blocking I/O)
  + not sure about their performance
    - they are often build on top of stack of multiple layers
  + often they use a single-thread by default

**Server clusters**

* in order to support internet-wide services, we need to use server clusters/server farms
* a simple approach is to route the requests at the TCP level (image below)

A diagram of a response

Description automatically generated with low confidence

* the crux is to balance the load on the different servers
  + round-robin is perhaps the simples approach
  + application-layer solutions are also possible

**Servers and session state**

Problem: the execution of the same task on every request form a client may unnecessarily tax the server

Solution: the server can keep some (session) state (ex. information about the status of ongoing interactions with clients)

* the size of each message are potentially smaller
* the processing demand of each message are potentially smaller

For example, in a distributed file system, the server may avoid opening and closing a file for each remote read/write operation

* the server may keep a cache of open files for each client

Depending on whether or not a server keeps session state, a server is called stateful or stateless

**Stateless file server**

* consider a simples file service that supports two operations:
  + read data (from file)
  + write data (to file)
* if the server is stateless it keeps no information, therefore each request must include at least:
  + operation
  + client credentials
  + file name
  + file offset
  + number of bytes to transfer
  + data (only in write requests)
* on each read request the server must:
  + check permissions for client
  + open the file (open())
  + set the file offset as requestes (lseek())
  + read the data from the file (read())
  + close the file (close())

**Statefull file server**

* server may keep information on a table about previous requests of each client
  + file name
  + client permissions
  + current offset
  + id of previous request
* server may support two additional operations:
  + open file, which returns a file handle
  + close file
* read/write requests need to include only:
  + operation
  + file handle (need to encrypt request)
  + number of bytes to transfer
  + data (only in write requests)
* on each read request the server must:
  + look up the file handle on the table, to get the file descriptor
  + read the data from the file (read())
* maybe add an “lseek” operation

**Stateful server failures**

* keeping state information raises some challenges upon failure of either clients or server:
  + of consistency
  + of resource management
* loss of state when a server crashes may lead to:
  + ignoring or rejecting client requests after recovery:
    - the client will have to start a new session
  + wrong interpretation of client requests sent before crash, if server recovery is too fast
    - TCP connection port reuse
* keeping state (on server) when the client crashes may lead to:
  + resource depletion (ex. if a client crashes before invoking close())
  + wrong interpretation of requests sent by other clients after the crash
    - if client id is reused (ex. IP address and port number)
* client crashes:
  + challenge: resources reserved for the client may remain allocated forever
    - state, in the case of stateful servers
    - application specific resources
      * what if these resources can be used only in excluse access mode
  + solution: leases (and timers):
    - a server leases a resource to a client for a finite time interval. Upon its experiation, the resource may be taken away, unless the client renews the lease

**Stateless server failures**

* message loss:
  + stateless servers are not immune to problems arising form failures:
    - message duplication may lead to handling the same request several times
      * operations must be idempotent, if the transport protocol does not ensure non-duplication of packets
      * even if the transport protocol ensures non-duplication of packets, we may still need idempotent operations
  + how can stateful servers handle duplicated requests?
    - careful about client identification
* client identification:
  + use the address of the access point (ex. of the channel endpoint)
    - ex. the client’s IP address and port
    - issue: may not be valid for more than one transport session
      * (ex. if a TCP connection breaks and a new one is setup in its place, the port number on the client’s side may be different)
  + use a transport-layer independent handle (ex. HTPP cookies)
* load balancing:
  + problem: what if th service keeps state in main memory and the service uses load balancing?
  + solution: need to ensure that the state is accessible to all servers 8ex. by saving it in a database)
    - the reference consider such an implementation as stateless
    - but what would be the difference from using shared emmory between servers?
      * indeed, some of the issues discussed above, assume that state is kept in volatile memory

**Servers, State and Protocols**

* obs. : statelessness is a protocol issue:
  + a server can be stateless only if each protocol message has all the information for its processing
  + likewise, a server can be stateful only if each protocol message has enough information to relate it to previous communications
* for example, Netscape had to add HTTP-header fields specifically for cookies
  + HTTP is essentially stateless
  + cookies are a device that allows a server to keep state about a client session (actually there are other types of cookies that may lead to abuse):
    - servers generate and send cookies to the clients
    - clients store the cookies received from servers
    - clients piggyback the cookies on HTTP request

**Failures:**

Challenges:

* components in a distributed application may fail, while others continue operating normally
* on the internet it is virtually impossible to distinguish network failures from host failures or even a slow host

Solution:

* highly application dependent
* distribution is harder than concurrency
  + in concurrent (local) systems: the programmer needs to consider all possible execution interleavings
  + in distributed systems: the programmer needs also to consider all possible failures
    - distributed systems are inherently concurrent

**Security**

Challenge:

* server execute with priviledges that their clients usually do not have

Solutions: servers must:

* authenticate clients (ex. “ensure” that a client is who it claims to be)
* control access to resources (ex. check whether the client has the necessary permission to execute the operation it requests

A related requirement is data confidentiality

* need to encrypt data transmitted over the network

code migration (ex. downloaded form the network) raises even more issues

**Fault tolerance**

* definition: a system/component fails when it doesn not behave according to its specification
* definition: a system is fault-tolerant if it behaves correctly despite the failure of some of its components
  + obviously, no system tolares the failure of all its components
  + usually, a system tolerates only some kinds of failures, as long as they do not occur to frequently or they only occur on some of its components
* observation: fault tolerance is achieved by design. We need to include some redundancy in the system:
  + HW: processors, memory, I/O devices
  + Time: for executing additional tasks (ex. retransmission of a packet)
  + SW: to tamanage the redundant HW, or the repetition of a task or even n-version programming

**Triple Modular Redundancy (TMR)**

* HW-based FT-technique
* each node is triplicated and works in parallel
* the output of each module is connected to a voting element, also triplicated, whose output is the majority of its inputs
* the configuration can be applied to each stage of a chain
  + it masks the occurence of one failure in each stage~

**FT and Distributed Systems**

* obs: unless a distributed system is fault tolerant it will be less reliable than a non-distributed system
  + a distributed system comprises more components than a non-distributed system
* obs: the inherent HW redundancy in a distributed system makes it particularly suitable for making it fault tolerant
  + but, fault-tolerance does not emerge directly from distribution, it must be engineered

**Reliability and availability**

* reliability (R(t)) – the probability that a system has not failed until time t
  + particularly important for mission-oriented systems, such as spacecrafts, aircrafts or cars
  + it is often characterized by the mean time to failure (MTTF)
* availability: assumes that a system may be repaired after failing
  + limiting the probability that a system is working conrrectly:
    - particularly important for systems like utilities, services on the web, that tolerate the occurences of failures
* obs.: reliability and availability are somewhat orthogonal:
  + a system A may be more reliable than system B and still be less available
  + a system A may be more available than system B and still be less reliable

**Distributed system model**

* a set of sequential processes that execute the steps of a distributed algorithm
  + DS are inherently concurrent, with real parallelism
* processes communicate and synchronize by exchanging messages
  + the communication is not instantaneous, but suffers delays
* processes may have access to a local clock
  + but local clocks may drift wrt real time
* DS may have partial failure modes
  + some components may fail while others may continue to operate correctly

**Fundamental models**

* synchronism: characterizes the system according to the temporal behavior of its components:
  + processes
  + local clocks
  + communication channels
* failure: characterizes the system according to the types of failures its components may exhibit

**Models of synchronism**

* synchronous if:
  + there are known bounds on the time a process takes to execute a step
  + there are known bounds on the time drift of the local clocks
  + there are known bounds on message delays
* asynchronous: no assumptions are made regarding the temporal behavior of a distributed system
  + these 2 models are the extremes of a range of models of synchronism
* Dilemma:
  + it is relatively simple to solve problems in the synchronous model, but these systems are very hard to build

**Failure models**

* characterize a system in terms of the failures of its components (ex. the deviations from their specified behavior)
* crash: a component behaves correctly until some time instant, after which it does not respond to any input
* omission: a component does not respond to some of its inputs
  + loss of a message can be seen as an omission failure of the communication channel or of either processes at the channel ends
* timing/performance: a component does not respond on time (ex. it may repond too early or too late)
  + makes sense only on synchronous systems
* byzantine/arbitrary: a component behaves in a totally abritrary way (ex. a process may send a message as if it were another process)
* crash-recovery: in this model, we assume that a fault process may crash and recover a finite number of times
  + the practice is that if nothing is stated, then they do not
* failure and synchrony models
  + the byzantine model is similar to the asynchronous model in that:
    - neither model makes any assumption wrt the aspect of behavior it is supposed to describe
  + in the absence of faults, the synchronous and the asynchronous models are equivalent
    - they can solve the same set of problems

**Atomic commitment:**

* reminder: transction’s ACID properties
  + Atomicity: either all operations of a transaction are executed or none
  + Consistency: a transaction transforms a consistent sate into another consistent state
  + Isolation: the effects of a transaction are as if no other transactions executed concurrently
  + Durability: the effects of a transaction that commits are permanent
* informal:
  + problem: how to ensure that a set of operations executed in different processors?
    - either are all executed (all committed)
    - or none of them are executed (all aborted)
  + obs.: the origin of this problem is distributed databases (ex. distributed transactions)
    - the transaction comprises operations (sub-transactions) in different DBs
    - a transaction must be atomic (and also CID)
  + obs.: AC is useful:
    - not only when processes may fail
    - but also when the operations may not be performed because of some reason other than the failure of the process that execute them
* formal:
  + def: consider a set of n processes such that:
    - each process has to decide one of two values (commit/abort)
    - each process shall vote/propose one of these two values
    - the value decided by each process must ssatisfy the following assertions:
      * AC1: all processes that decide, must decide the same value
      * AC2: the decision process is final (cannot be changed)
      * AC3: if some process decides to commit, then all processes must have voted commit
      * AC4: if all processes voted commit and there are no failures, then all processes must decide commit
      * AC5: for any execution containing only failures that the algorithm is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes eventually reach a decision

**Two-phase commit: a solution for AC**

* assumptions: processes may fail by crashing and recover
  + each process ahs storage whose content survives a crash
* the protocol has two kinds of processors:
  + coordinator
    - there is only one of these, at any time instant
  + participant
    - every process that performs an operation is a participant
      * the coordinator may also be a participant, in which case it will have to perform both the coordinator-side and the participant-side of the protocol
* the protocol has two phases:
  + phase 1: upon request from application
    - coordinator: sends a VOTE-REQUEST to each participant and waits for their reply
    - participant: upon receiving a VOTE-REQUEST each process sends its vote, either VOTE-COMMIT/YES or VOTE-ABORT/no
  + phase 2: once the coordinator determines that it is time to decide
    - coordinator decides/sends:
      * GLOBAL-COMMIT if it received a VOTE-COMMIT/YES from all participants
      * GLOBAL-ABORT otherwie
    - participant: decides according to the message received from the coordinator
* timeout actions: actions taken by a process upon a timeout
  + coordinator: waits for messages only in the WAIT state
    - decides ABORT and send a GLOBAL-ABORT to participants
  + participants: waits for messages both in:
    - INIT: decide ABORT and move to corresponding state
    - READY: must execute a termination protocol to find out the outcome
    - must communicate with the other participants to find out the outcome
      * if some participant was voted VOTE-ABORT
      * if some participant knows the decision
      * otherwise
        + process must wait until it learns the coordinator’s decision

may continue probing both the coordinator and other participants

* recovery actions: actions taken by a process upon recovery form a crash
  + assumes that each process keeps in stable storage the state of the 2PC protocol
  + if the process has not decided yet then
    - if crashed while waiting for a message, take the corresponding timeout action
      * including the execution of a termination protocol, if a participant failed in the READY state
    - otherwise (coordinator in the INIT state) decide ABORT
  + to allow recovery, processes must write the state of the protocol as entries to a lo in stable storage
    - the write of the entry should be performed before or after sending the corresponding message?
    - in addition to the state of the protocol, the log may be used to store application data
  + the 2-phase commit protocol satisfied assertions AC1-AC5, even in the presence of:
    - non-byzantine node failures
    - communication faults, including partitions
  + the main problem with this protocol is that it may require participants to block (wait longer than a communication timeout)
    - this problem can be made less likely by using the three-phase commit protocol
* independent recovery and blocking:
  + impossibility of independent recovery: there is no AC protocol that always allows local recovery (ex. without communication with other processes)
  + non-blocking impossibility: there is no AC protocol that never blocks in the presence of either:
    - communication failures
    - failures on all other processes
  + 2 phase-commit: may block even when there is no failure on all sites
  + assuming:
    - communication is reliable
    - process failure can be reliably detected

**stable storage**

* problem:
  + many protocols like 2PC assume that the state of processes, or at least some part of it, survives the failure of the process
  + usually, this means to store the state on disk
* solution: stable storage
  + use two identical disks
  + writing a block required writing first in disk 1 and then in 2
  + upon reading a block, try disk 1 first, unless its checksum is not valid
* recovery after a crash:
  + if the checksum of dis 1 is vlid, and the two blocks are different, copy block from disk 1 to disk 2
  + if the checksum of disk 1 is not valid, use the block on disk 2, if its checksum is valid
  + if both checksums not valid, then data has been lost (ex. have a catastrophic failure)

**Leader election**

Why: many distributed algorithms rely on a process that plays a special role – coordinator/leader. Such algorithms usually are:

* simpler
* more efficient

What: upon completion of the algorithm all non-faulty nodes agree on who the coordinator is

* only one node is elected the coordinator
* all nodes know the identify of the coordinator

**Garcia-Molina’s algorithms:**

* the algorithms were proposed in the scope of system reorganization upon failure/recovery of system components. but elections are also useful:
  + at initialization
  + to add/remove nodes (to a less extent)
* we can ensure fault-tolerance by two approaches:
  + masking failures: (ex. by using algorithms that continue to work correctly, even if some system components fails)
    - this is the only approach if we need continuous operation
    - also likely to be the more appropriate, if failures are common
  + reorganizing the system: (ex. by halting the normal operation and take some time to reorganize the system)
    - likely to allow simpler algorithms
* we abstract the leader election problem from its context
  + the leads to simpler versions of the GM’s algorithms

**System Model/assumptions**

* all nodes cooperate and use the same algorithm
* all nodes has some stable/safe storage
* when a node failts, it immediately halts all processing
  + crashed nodes may recover
  + data on stable storage is not lost (ex. is as before the crash)
* the communication subsystem does not spontaneously generate messages
* there are no transmission errors (but messages may be lost)
* messages are delivered in the order in which they are sent
* communication system does not fail and has an upper bound on the time to deliver a message
* a node always responds to incoming messages with no delay
* observation: last 2 assumptions mean system is synchronous

**Specification of Leader Election**

* Assertion 1: at any time instant, for any two nodes, if they are both in NORMAL state, then they agree on the coordinator
* Assertion 2: if no failures occur during the election, the protocol will eventually transform a system in any state to a state where:
  + all non-faulty nodes has state = NORMAL
  + there is a non-faulty node i such that it is the coordinator

**Safety vs Liveness Properties**

* any specification can be expressed in terms of safety and liveness properties:
  + safety: states that something (bad) will not happen
    - proving such a property involves proving an invariant
    - once an execution violates a safety property, there is nothing that can be done to fix that
  + liveness: states that something (good) must happen
    - proving such a property involves a different technique
    - any “partial” execution can always be extended so that eventually something good happen
      * if that is not possible, then something bad must have happened (ex. some safety property must have been violated)

**Leader election vs Mutual exclusion**

* in an election fairness is not important
  + all we need is that one node becomes the leader
* an election protocol must deal properly with the failure of the leader
  + usually, mutual exclusion protocols assume that a process in a critical section does not fail
* all nodes need to learn who the coordinator is

**Bully election algorithm**

* idea: node wishing to become leader:
  + looks around to ensure stronger nodes are not up (phase 1)
  + if does not see any, it
    - imposes itself as leader (phase 2)
    - brags about it
* convention: the smaller a node’s id the storng it is
* phase 1:
  + a node wishing to become leader checks if stronger nodes are around sending them an ARE-U-THERE message
    - if present, a stronger node responds with a YES message and initiates a new election itself
      * by checking if stornger nodes are around
    - a candidate whose challenge is answered, backs off
      * but should start a timeout to detect a possible failure of the challenger
* phase 2:
  + node i begins phsae 2, if it does not receive any response to its probe withing 2T, it comprises two steps:
    - send a HALT message to weaker nodes
      * sets its state to ELECTION
      * cancels any election it has already started
    - T time units later, node i send a NEW-LEADER message to weaker nodes, and sets S(i).c to i and S(i).s to NORMAL
      * upon receiving that message, node k sets S(k).c to i and S(k).s to NORMAL
* comment: the HALT message (1st step) is required to ensure that Assertion 1 is not violated
* failure of the candidate during the second phase will trigger new election
* what about node failures? depends on the node:
  + leader: upon detection of failure of the current leader
    - a process initiates an election
  + candidate: upon detection of failure of of the candidate
    - a process initiates an election
  + other processes: it does not matter
    - GM’s algorithm starts a new election on such an event, because its focus is on reorganization

**The actual bully algorthm**

* RPC: rather than messages, with some interesting features
  + synchronous: caller is suspended until it receives the reply from remote process or there is a timeout
  + ONTIMEOUT clause
    - ARE-U-THERE, HALT: just send the next RPC (synchronous)
    - NEW-LEADER: restart election (reorganization)
  + immediate: assume special implementation for low latency
    - used for RPCs that use timeouts
* failure suspector: each node has a failure suspector that triggers a TIMEOUT event, if the node does not receive a message from the last known leader for some tim
  + upon a TIMEOUT event
    - if s == NORMAL the node calls ARE-U-THERE
      * starts a new electoin if ARE-U-THERE times out
    - otherwise: starts new election immediately
* if we drop last 2 assumptions, first 2 assumptions cannot be satisfied always:
  + assume node i is the coordinator, has not crashed but it does not respond to other nodes because it is too slow
  + from the point of view of other nodes, it has crashed, so to satisfy assertion 2, they must elect a new coordinator
  + but, if they elect a new coordinator, assertion 1 will be violated

**The invitation algorithm**

* idea: rather than imposing itself as a leader, a node wishing to become coordinator invites others to join in a group where it is the coordinator
  + initially, each node creates a singleton group
  + periodically, coordinators try to merge their group with other groups in order to form larger groups
* description:
  + failure detection: a node that is not a leader periodically checks if its leader is still alive
    - if not, created singleton group
  + group merging: a node that is a leader periodically probes all other nodes for leadership
    - if one or more nodes reply, node i initiates the merging protocol after a delay inversely proportional to its priority
      * the varaible delay helps preventing different nodes to intiate the merging concurrently
* concurrency:
  + a process moves to the ELECTION state, before:
    - sending the INVITIATION message, if candidate
    - sending the ACCEPT message, otherwise
  + a process moves to the NORMAL state, after:
    - sending the READY message, if candidate
    - receiving the READY message, otherwise
  + a process responds to
    - ARE-U-LEADER
    - INVITATION
  + ... messages only if its state is NORMAL (ex. it is not participating in an election)
    - a process dont not participate in more than one election at a time
* liveness
  + process failure is suspected using timeouts while waiting for the reception of some messages in response to:
    - ARE-U-THERE, sent to coordinator
    - INVITATION, send to other leaders
    - ACCEPT, sent to candidate
    - READY, sent to new group members
  + upon timeout, several possible actions:
    - advancing to the next phase of the protocol (ex. when waiting for responses to INVITATIOON)
    - initiating recovery procedure (ex. when waiting for responses to ACCEPT) that creates a singleton group
      * no need for communication