Distributed Systems

Dr. Kostas Kalpakis

kalpakis@umbc.edu

Distributed System Abstractions

Version: January 25, 2022

Need for Abstractions in Distributed Systems

Need to consider appropriate abstractions when reasoning, analyzing, designing, and implementing distributed systems and applications

In an attempt to optimize performance, one may mix relevant logic of primitives with application logic, making it

- prone to errors
- hard to adapt or extend since there is usually no single solution for a given distributed computing problem with a fluid solution space and operating environment
- "*Premature optimization is the root of all evil.*" (Donald Knuth)

Abstractions of primary importance are those representing

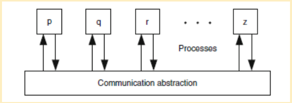
- processes
- communication links

The material in this section is based on the book "Introduction to Reliable and Secure Distributed Programming" by C. Cachin, L. Rodrigues, and R. Guerraoui (2011)

Process Abstraction

System

- A distributed system or application consists of a distributed collection of processes and a communication abstraction.



Process rank

Associate an a priori unique numeric rank $1 \dots N$ to each of the N processes in a system. Ranks are useful when choosing a process among those in a process group.

Process as a State Machine

State machine

Each process is constructed as a state-machine (automaton) whose transitions are triggered by the reception of events and may trigger some event.

- Request events are used by a component to invoke a service at another component or to signal a condition to another component.
- Indication events are used by a component to deliver information or to signal a condition to another component.

Events may be local or global depending on where they are triggered or received/delivered.

The internal state of a process may have a volatile and persistent part.

Process Executions

Process steps

Process execution is represented by a sequence of process steps

- receive a message
- do some local computation changing its internal state
- send another message.

Executions

An execution of a distributed algorithm, application, or system is a sequence of steps by its processes.

We mostly consider deterministic process steps.

A property of an execution is a

- safety property if once it is violated, it can not be satisfied again
- liveness property if at any time there is hope it may become satisfied at a later time

Construction

Layering

A process is often constructed by composing or layering multiple components, each of which is a state-machine implementing an algorithm

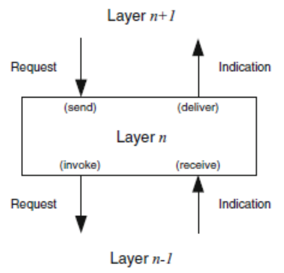


Figure 1.2: Layering

Process Failure

A process fails

when its state-machines does not behave according to its logic in executing its process steps (local computations and event handling).

Types of process failures

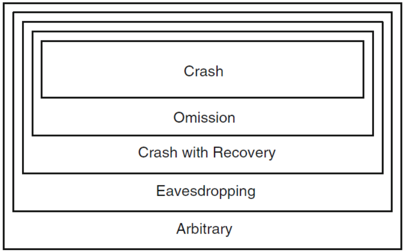


Figure 2.3: Types of process failures

Failure Types

A crash-stop fault

occurs at time t, if the process stops at time t and does not execute any more processing steps at any time $\geq t$ (never recovers).

A process is called faulty if it crashes during its execution, otherwise is called correct.

An omission fault

occurs at time t, if the process does not send or receive a message at t as expected according to its algorithm.

Crash with recovery fault

A process can crash and stop sending messages, but may recover later. Upon recovery, the process loses the volatile part of its internal state at the time of the crash, and may restore the persistent part of its internal state from stable storage (log).

A process is faulty if either the process crashes and never recovers or crashes and recovers infinitely often.

Failure Types

An eavesdropping fault

occurs when, in an untrusted environment, a process leaks information to an adversary violating the confidentiality of the information it handles.

It is often prevented by encrypting communication messages and stored data.

An arbitrary fault

occurs when the execution of a process deviates in any conceivable way from its algorithm.

A faulty process is allowed any kind of output or outgoing messages. Such failures are also called Byzantine for historical reasons.

Communication Links

Communication links and messages

A link is used to represent the network components of the distributed system.

- Links between processes are bidirectional.
- Processes are connected by a link as specified by an overlay topology (which defaults to a clique).
- Messages exchanged between processes over a link are unique. Every message includes enough information for the recipient to uniquely identify its sender.
- When two processes exchange messages in a request reply manner, they are able to pair a reply message to its corresponding request message by including appropriate information with the messages (e.g. identifiers, timestamps, etc), alleviating the need to explicitly introduce such information.

Communication Events

Events

Three kinds of events

- Send request event for the link requesting it to send a message
- Receive indication event at the receiver signaling it that a message is available for it
- Deliver indication event triggered by a receiver to signal the availability of a message for another component (link or process)

A message is received (delivered) when the receiver handles the Receive (triggers a Deliver) indication event.

Communication delays

Links have latencies and bandwidth that affect the elapsed time between the send and receive/deliver events for a message.

Fair-loss Links

Fair-loss links

Messages might be lost but the probability for a message not to be lost is nonzero.

These links have these properties:

- fair-loss: if both the sender and the receiver are non-faulty, and if the sender keeps retransmitting a message, the receiver eventually delivers the message.
- finite duplication: a message is delivered a finite number of times
- no creation: if a message is delivered it was previously sent

Fair-loss links may experience (receive/delivery) omission faults.

Stubborn Links

Stubborn links

They have the no creation property and

- the stubborn delivery property that causes every message sent over the link to be delivered an unbounded number of times

Stubborn links overcome omission faults of fair-loss links.

Implementation

A pedagogical but impractical implementation of a stubborn link over a fair-loss link is to keep retransmitting all messages.

Since stubborn links may retransmit the same message several times, in practice, an optimization mechanism is needed to acknowledge the messages and stop the retransmission.

Perfect Links

Perfect (reliable) links

A perfect (reliable) link has these properties

- reliable delivery: if the sender and receiver of a message are non-faulty, the receiver eventually delivers the message
- no duplication: each message is delivered at most once
- no creation: if a message is delivered it was previously sent

Implementation

A simple implementation of a perfect link over a stubborn link is for the receiver to deliver a received message only if it was not previously delivered.

Logged perfect links

A logged perfect link extends a perfect link with the ability to trigger Delivery indication events by maintaining a collection of sender/message pairs in stable storage.

FIFO Perfect Links

FIFO perfect links

A FIFO perfect link is a perfect link with the additional

- FIFO property: if the sender sends m_1 and it later sends m_2, the receiver does not deliver m_2 before it delivers m_1

Implementation

To implement a FIFO perfect link over a perfect link:

- sender attaches a seqno to each message (counting the messages sent on the link), and pass it to the perfect link
- receiver, upon receiving a message from the perfect link, queues the message and delivers it only when its seqno equals to 1 + seqno of last delivered message

Authenticated Links

Authenticated links

Fair loss or perfect links are not very useful for processes with Byzantine faults.

Can extend these links with the

- Authenticity property: if both the sender p and receiver q of a message m are non-faulty, then m was sent from p to q.

Eliminating message forgery on links between non-faulty processes.

To implement an authenticated perfect link over a perfect link, the sender p includes a MAC authenticator for the (p, q, m) triplet with the message m.

The receiver q, upon receiving a message, verifies the authenticator and then delivers m.

Topology Abstraction

Issues

- What other processes does a particular process has a communication link to?
- Is a process's neighborhood static or dynamic?
- How do processes and links map to physical machines and communication channels of the system?

Overlay topology

The process neighborhoods in a distributed system induce a natural graph *G* for the processes and their links, the system's overlay topology:

- a graph in a family of graphs (e.g. cliques, lines, rings, trees, hypercubes, Chord rings, etc)
- a subgraph of a member of a graph family

We assume that the overlay topology is always a connected graph.

Topology Changes

Static vs dynamic overlays

An overlay topology can be static (never changes) or dynamic (changes over time).

An overlay topology becomes dynamic by changes to the collection of system's processes and links over time (e.g. new process join/leave the system, links established/dropped, crash–stop failures of processes or links, etc).

We assume that the overlay topology does not change due to crash–recovery failures of processes or links.

Topology Embedding

Guest vs Host

The overlay topology, called a guest graph, is implemented by mapping its vertices and edges to the physical topology graph of the underlying real physical machines (nodes) and communication channels, called the host graph.

Embedding a guest onto a host

Each guest vertex is mapped to a host vertex, and each guest edge is mapped to path of host edges between the host vertices of the guest edge's incident guest vertices.

Such a mapping is called an embedding of the guest graph into the host graph.

Embedding Metrics

Metrics

The

- load (number of guest vertices mapped to a host vertex),
- dilation or stretch (length of host path of guest edge), and
- congestion (number of occurrences of a host edge in host paths mapping guest edges)

are important metrics that characterize the performance of a host-guest embedding, since it induces a slowdown of

- a guest process step by a $O(\textit{max load})$ factor, and
- a message exchange on a guest link is by a $O(\textit{max}[\textit{dilation} + \textit{congestion}])$ factor

An iso–embedding, which is the natural embedding when the guest is an isomorphic subgraph of the host, simultaneously achieves the least possible load, stretch, and congestion of 1.

An unitary embedding achieves $O(1)$ load, stretch, and congestion.

Default Overlay Topology Embedding

Unless stated otherwise, we assume a clique overlay topology (fully–connected graph) and an iso–embedding of the overlay topology into a physical topology.

The physical topology and the overlay topology's embedding into it will be specified when needed in a discussion.

Cryptographic Concepts

Communication in untrusted environments, which may be exposed to a malicious adversary, often rely on cryptographic primitives for protection.

Cryptographic hash functions

H map a bit string of arbitrary length to a short, unique representation.

Function H is collision free in the sense that no process, not even one subject to arbitrary faults, can find two distinct values x and y such that $H(x) = H(y)$

Symmetric cryptosystems

assume a secret key and encryption/decryption functions $D_{SK}(E_{SK}(m)) = m$

Asymmetric cryptosystems

assume pair of public-secret keys and encryption/decryption functions $D_{SK}(E_{PK}(m)) = m$

Message Authentication Code

Message Authentication Code (MAC)

A MAC authenticates a message between a sender and a receiver.

- The sender can compute an authenticator for a message of its choice to the receiver
- The receiver, given the message and the authenticator, can verify that the message has indeed been authenticated by the sender
- It is infeasible for any other entity to come up with a message that was never authenticated and to produce an authenticator that the receiver accepts as valid during verification

MACs are often based on symmetric cryptographic primitives in practice (such as hash functions, stream/block ciphers), they can be computed and verified very fast

Digital Signature Schemes

Digital signatures

A digital signature scheme provides data authentication in systems with multiple entities that need not share any information beforehand

- are often based on public-key/private-key (asymmetric) cryptographic primitives, and add considerable computational overhead compared to symmetric cryptography
- the private key is given to an entity and must remain secret; its public key is accessible to anyone
- with its private key, the entity can produce a signature for a message of its choice
- everyone with access to the sender's public key can verify that the signature on the message is valid
- it is infeasible for any entity without the sender's private-key to forge their signature on a message

Timing Abstractions

Need

The behavior of the processes and links of a distributed system with respect to the passage of time is important in their analysis, design, and implementation

Happened before relation

The happened before relation $e_1 \longrightarrow e_2$ captures potential causal dependencies among events

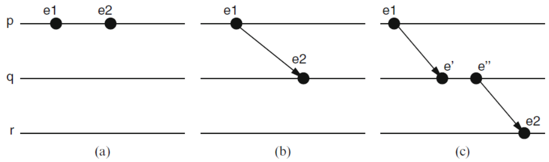


Figure 2.5: The *happened-before* relation

Synchronicity in Distributed Systems

Asynchronous distributed system

We do not make any timing assumptions about processes and links.

Synchronous distributed system

We assume

- Synchronous computation: there is a known upper bound on processing delays
- Synchronous communication: there is a known upper bound on message transmission delays
- Synchronous physical clocks (optional): every process has a local physical clock with a known bound on its deviation from a global real-time clock

Synchronicity in Distributed Systems

Partially synchronous distributed system

Distributed systems often appear to be, but are not, synchronous.

One way to capture partial synchrony is to assume that

- the synchronous timing assumptions only hold eventually, without stating when exactly
- there is no bound on the period during which the system is asynchronous
- there are long periods where the system is synchronous for it to achieve a useful computation

Failure Detectors

Failure detectors

A failure detector provides (not always accurate) information about which processes have crashed and which are correct.

Failure detectors

- alleviate the need for extending the process and link abstractions introduced earlier with timing assumptions
- enable us to reason about their behavior without explicit references to physical time

Perfect Failure Detectors

Properties

A perfect failure detector for crash-stop processes has these properties

- strong completeness: eventually, every process that crashes is permanently detected by every correct process
- strong accuracy: if a process is detected, then it has crashed

"*Exclude on timeout*" algorithm

A simple algorithm implementing a perfect failure detector for a synchronous system is as follows

- exchanges heartbeat messages among all processes and uses a timeout timer with a timeout delay Δ
- upon a timeout, any processes with no response are declared crashed, and the previous step if repeated

The timeout Δ is sufficiently large to enable the heartbeat exchange and processing of all processes

Eventually Perfect Failure Detector

The Eventually Perfect Failure Detector is useful with partially synchronous crash-stop process failures: may make mistakes for a while, but eventually detects crashes accurately.

To implement it,

1. use a perfect failure detector with some initial timeout delay Δ
2. mark as suspect any processes with no heartbeat
3. upon receiving a heartbeat from a suspect, restore the suspect and increase the timeout

Leader Election

Leader election selects one process as a unique representative of the processes in the system.

When the current leader crashes a new leader is elected.

Leader election properties

- eventual detection: either there is no correct process, or some correct process is eventually elected as the leader
- accuracy: if a process is leader, then all previously elected leaders have crashed

Leader election in primary-backup replication schemes

A leader of the non-faulty processes, called primary, handles client requests on behalf of the other replicas, called backups and updates the backups before returning a reply to the client. If the leader crashes, one of the backups is elected as the new leader. Provides the illusion of a highly available service that tolerates faults of some processes.

Monarchical Leader Election

For crash-stop processes, leader election can be accomplished using a perfect failure detector

- select as leader the process with the maximum a priori rank, among all the non-crashed processes given by the perfect fault detector

If instead of the perfect failure detector an eventually perfect failure detector is used, an eventual leader is elected, i.e. there is a time after which the accuracy and agreement properties of leader election hold.

Byzantine Eventual Leader Election

Byzantine eventual leader–detector

It is applicable to fail-arbitrary processes with $\leq f$ faulty among $N > 3f$ processes.

Any correct processes monitor the actions of the current leader, and complain to the leader detector if the current leader does not meet the desired goal after waiting long enough for the leader to achieve that goal.

Byzantine eventual leader-detector properties

- eventual succession: if more than f correct processes that trust a process p complain about p, then every correct process eventually trusts a different process than p
- Putsch resistance: a correct process does not trust a new leader unless at least one correct process has complained against the previous leader
- eventual agreement: there is a time after which no two correct processes trust different processes

Rotating Byzantine Eventual Leader Detection Algorithm

Processes maintain an increasing round variable r to derive a leader at round r:

the leader is simply the process p whose rank is $(r \mod N) + 1$.

The algorithm proceeds as follows

1. When the detector receives a complain request for the current leader, it broadcasts a [COMPLAIN, round] message to all.
2. Whenever a process receives more than $2f$ COMPLAIN messages against the current leader, it switches to the next round
3. When a process receives more than f messages [COMPLAIN, round] but has not sent a COMPLAIN message itself in the current round, it joins the complaining processes and also sends a [COMPLAIN, round] message

Rotating Byzantine Eventual Leader Detection Algorithm

Correctness

- Since every complaining correct process broadcasts a [COMPLAIN, round] message, and there are $\geq N - f > 2f$ correct processes, round advances, and the current leader is replaced.
- Eventual agreement is achieved since
  - When all COMPLAIN messages have been received subsequently, every correct process is in the same round and trusts the same process.
  - The correct processes eventually cease to complain against a correct leader provided they wait long enough for the leader to achieve its goal.

Distributed System Models

A combination of a process, link, and possible failure-detection and leader-detection abstractions defines a distributed system model.

We will consider the following five models: fail–stop, fail–noisy, fail–silent, fail–recovery, and fail–arbitrary.

Fail-stop model

Assumes the crash-stop process, perfect link, and perfect failure-detection abstractions.

Fail-noisy model

Assumes the crash-stop process, perfect link, and eventually perfect failure-detection or eventual leader-detection abstractions.

Distributed System Models

Fail-silent model

This model assumes the crash-stop process and perfect link abstraction.

It does not assume any failure or leader detection abstraction, i.e. processes have no means to get any information about other processes having crashed.

Fail-recovery model

This model uses the crash-recovery process, stubborn link, and eventual leader election abstraction.

Algorithms for this model have to copy with the consequences of amnesia by recovered processes.

Fail-arbitrary model

This most general model uses the fail-arbitrary (or Byzantine) process and the authenticated perfect links abstractions. Also called fail-silent-arbitrary model.

This model augmented with a Byzantine eventual leader detector is called the fail-noisy-arbitrary model.

Register abstraction

Register

A register r stores values that can be accessed by a process via $read(r)$ and $write(r, v)$ operations, where each operation at a process starts with an invocation request event and completes with an indicator response event

Processes use registers to communicate with one another, whereas a process accesses registers in sequential manner (no concurrent operations at a single process).

Registers resemble CPU registers and disks accessed over the network.

Register values

- each register is initialized with distinct special value (null)
- assume (for simplicity and without loss of generality) that each value written to a register is distinct [eg store each value together with its writer's rank and #writes invoked]

An (M, N) register is a register where M and N processes can write and read respectively.

Register operations

Completed vs failed operations

A read/write register operation by a process p

- completes if both of its invocation and response events occur at p
- fails if p crashes before the response indicator event occurs at p

A register r is robust (or wait-free) iff every correct process has all of its operations on the register r eventually complete.

When $o : write(r, v)$ completes we say value v "was written by" o, and when $o : read(r)$ returns v, which was written by completed $o' : write(r, v)$, we say that o "reads from" o'

Semantics

The semantics of register operations vary in the face of concurrency and failures. We will consider safe, regular, and atomic register semantics

Concurrent register operations

Precedence and concurrency

For any two operations o_1 and o_2 on a register r

- o_1 precedes o_2 if o_1's response event occurs before o_2's invocation event (with respect to the latent real physical time)
- o_1 and o_2 are called concurrent iff neither preceded the other, else are called sequential

Note

A failed write operation o_1 by a crash–stop process is concurrent with each read operation that is invoked after o_1's invocation

Safe register semantics

Private register

An (1,1) register without any failures has these properties (semantics)

- liveness: eventually every operation completes
- safety: each read operation returns the value written by the last write operation

Safe register semantics

A register *r* is safe iff: a read that is not concurrent with a write returns the last value written; a read that is concurrent with a write can return any value.

Regular register semantics

Regular register

An $(1, N)$ register is regular if: a read that is not concurrent with a write returns the last value written; a read that is concurrent with a write returns the last value written or the value concurrently written.
There is no agreed upon definition of (N, N) regular registers.

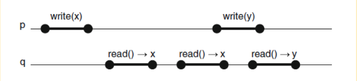


Figure 4.2: A regular register execution

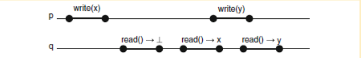


Figure 4.1: A register execution that is not regular because of the first read by process q

A read–one write–all quorum protocol can implement (1,N) regular register r in a fail–silent system model

- each reader process has a local copy (replica) of r's value
- a write operation updates all replicas of r for a write-quorum
- a read operation returns the latest value of a read quorum

Quorum systems

Quorum system

A collection of subsets $Q_i \subseteq U$ of a universe U such that all pairwise intersections $Q_i \cap Q_j$ are non-empty.

Read–Write quorum system

A collection of subsets $Q_i \subseteq U$ of a universe U that

- distinguishes quorums Q_i as read or write quorums
- requires non-empty intersections for all pairs of read-write and write-write quorums

(1,N) atomic register semantics

A $(1, N)$ register is atomic if it is regular and in addition:

> *whenever a read returns a value v and a subsequent read returns a value w, then the write of w does not precede the write of v*

(the read and write operations may happen at any process)

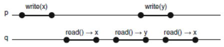


Figure 4.4: A register execution that is not atomic because of the third read by process *q*

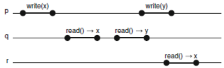


Figure 4.5: Violation of atomicity in the “Read-One Write-All” regular register algorithm

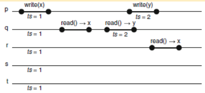


Figure 4.6: Violation of atomicity in the “Majority Voting” regular register algorithm

A read–impose write–all quorum protocol can implement a $(1, N)$ atomic register in a fail-silent system model: modify the read–one write–all protocol so that

> *the read operation writes–back the value is about to return to all other replicas in its chosen read quorum*

ROWA alone is insufficient for atomicity

(N,N) atomic register semantics

An (N, N) atomic register is a (1,N) atomic register where in addition,

each read operation returns the most recently written value in a hypothetical execution where each failed write appears to be either completed or never invoked and every complete operation appears to have been executed at some instance between its invocation and its completion

An (N,N) atomic register r can be implemented

- with multiple (1,N) atomic registers, one per writer process, and
- having a writer p read all (1,N) registers to determine a max timestamp to use for the update to its (1,N) register