DISTRIBUTED SYSTEMS

03

Algorithmic Prerequisites

Konrad Iwanicki

Copyright notice

Acknowledgments

This lecture is (partly) based on:

1. C. Cachin, R. Guerraoui, and L. Rodrigues: *Introduction to Reliable and Secure Distributed Programming*, Second Edition, Springer-Verlag, (February 12, 2011), 386 pages, ISBN 978-3642152597, Chapters 1 and 2.

2. M. van Steen and A. Tanenbaum: *Distributed Systems*, CreateSpace Independent Publishing Platform, 3.01 edition (February 1, 2017), 596 pages, ISBN 978-1543057386, Chapter 8.

Issue

How to model the various aspects of distributed systems so that we can focus on solving common problems?

Processes

- The abstractions that are responsible for performing computations are referred to as ***processes***.
  - They need not correspond to actual OS processes.
- The set of all processes is denoted as Π (capital "pi").
- In the initial algorithms, we assume that:
  - Π is fixed.
  - $|\Pi| = N$ for some small N (typically 3 or 4).
  - Processes are named p, q, r, s, and so on (i.e., $\Pi = \{ p, q, r, s, \ldots \}$).
  - Each process knows Π (i.e., the name of every other process).
- Sometimes we will also use a function that associates each process with a unique identifier, a so-called ***rank*** : $|\Pi| \rightarrow \{ 1, 2, \ldots, N \}$.

Storage

- Each process has its local data ***memory*** that is inaccessible directly by other processes.
  - The memory is normally transient (i.e., its contents are lost upon a failure of the process).
- If needed, the process can also have some form of ***persistent storage***.
  - In such a case, data from/to transient to/from persistent storage have to be transferred explicitly.

Messages and Links

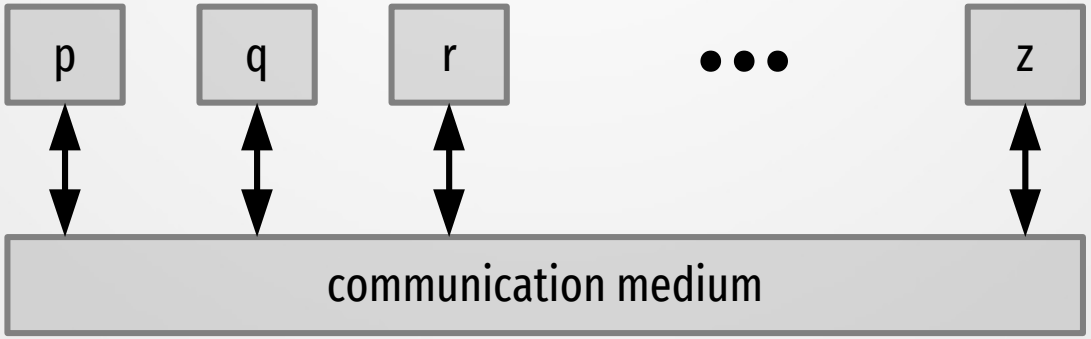
- The processes communicate by exchanging ***messages***.
- Each message is unique.
- Messages are exchanged over unidirectional ***links***.
- We normally assume that there is a link between each pair of processes, including a link from a process to itself (i.e., N^2 links in total).
- Link properties will be discussed shortly.

Distributed Algorithm

- Typically we assume that all processes run the same local algorithm.
- The union of these instances constitutes the actual ***distributed algorithm***.

Modeling Algorithms

- A distributed algorithm is modeled by a collection of ***automata***, one per process.
  - The automaton of a process regulates the way the process executes its computations (i.e., local algorithm).
  - Every process is implemented by a dedicated instance of the same automaton.
- The automata interact via a ***communication medium***.
  - The communication medium models message transfers between the processes corresponding to the particular automata.
  - The automata react to messages from the medium by executing specific computations.

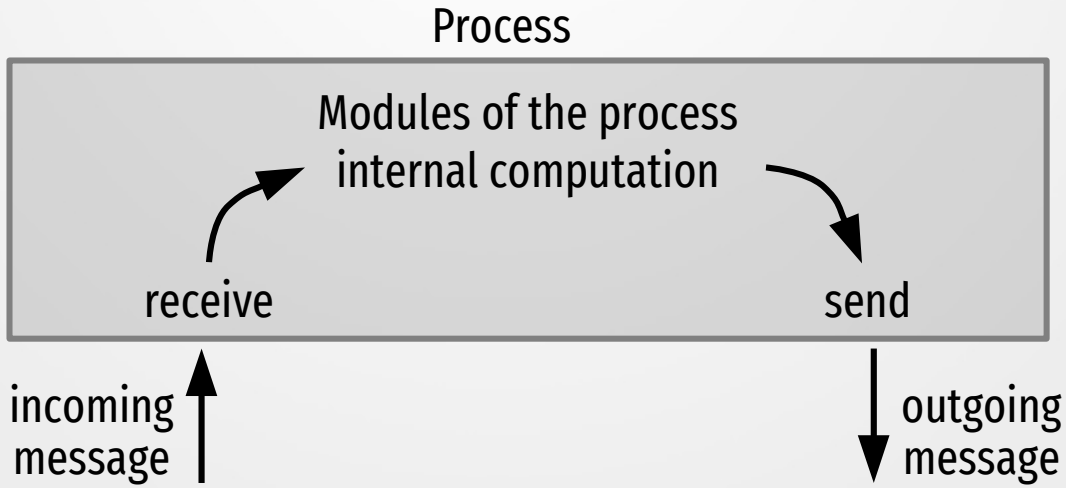


Modeling Algorithms

- An ***execution*** of a distributed algorithm is represented by a sequence of steps executed by the automata.
  - a partial execution = a finite sequence;
  - an infinite execution = an infinite sequence.
- We have a ***global clock*** but outside the control of the processes.
  - Provides a global and linear notion of time.
  - Regulates the execution of algorithms: one step of an automaton per tick.
- A ***correct*** (i.e., ***nonfailing***) process executes an infinite number of steps.
  - It gets an infinite share of time units.
  - There exists some global scheduler (again, outside the control of the processes).

Modeling Algorithms

- A **step** consists of:
  - receiving/delivering a message from another process (global event),
  - executing a local computation (local event),
  - sending a message to some process (global event).
- The execution of the local computation and the sending of a message is determined by the automaton (i.e., its local algorithm).
- Messages and steps can be *nil*.



Algorithm Correctness

There are two complementary classes of properties that we want our algorithms to satisfy:

- Safety and
- Liveness.

Algorithm Correctness

- A ***safety*** property essentially states that an algorithm should not do anything wrong.
- In a strict formulation, the property must never be violated.
- In a somewhat relaxed formulation, for any execution E of an algorithm in which the property is violated, there must not exist a partial execution E' of E such that the property will be continuously violated in some extension of E'.

Algorithm Correctness

- A ***liveness*** property ensures that eventually something good happens.
- It is typically a property such that at any time t, if specific conditions are met, there is a guarantee that the property will be satisfied at some time $t' \geq t$.

Algorithm Correctness

The challenge is to guarantee ***both*** liveness and safety.

"The difficulty is not in *talking* or *not lying* but in *telling the truth*."

Interested in formal proofs on algorithm correctness?
Take a look, for instance, at linear temporal logic (LTL).
A good introduction to LTL can be found in:
M. Ben-Ari: *Mathematical Logic for Computer Science*.
Springer-Verlag London, third edition, 2012.

Algorithm Performance

We will be interested mainly in two metrics:

- work and
- span.

These correspond to the concepts from parallel programming.
Their more formal, language-specific definitions can be found for instance in:
T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein:
Introduction to Algorithms, MIT Press, Cambridge, MA,
third edition, chapter 27, 2009,

Algorithm Performance

- The ***work*** of a algorithm is the total number of steps (of all automata) the algorithm requires to achieve its objectives.
- It essentially corresponds to a worst-case linearized execution of the algorithm on a given number of automata.

Algorithm Performance

- The ***span*** of an algorithm is the number of steps on the critical path.
- It is the number of steps that have to appear one after another irrespective of the underlying scheduling of computation and communication.

Algorithm Performance

- In our analyses, we distinguish:
  - a ***computation*** (sub-)***step*** – performing a local computation by a process;
  - a ***communication*** (sub-)***step*** – sending a message to another process.
- We assume that communication is more costly than computation and that it is the main source of latency.
- This need not be true in practice (e.g., cryptographic computations)...
- but will simplify our analyses.

Algorithm Performance

Under these assumptions:

- The ***work*** of an algorithm is the total number of messages it generates.
- The ***span*** of an algorithm is the number of communication steps that must be performed in sequence (the number of "communication rounds").

For some algorithms, we may use additional performance metrics (e.g., number of accesses to stable storage).

Failures

Distributed systems exhibit ***partial failures***
(i.e., failures of its components).

Failures

Sample failure types:

- ***Crash failure*** – a component halts but behaves correctly beforehand;
- ***Omission failure*** – a component fails to respond;
- ***Timing failure*** – a component's output is correct but outside the specified real-time interval (too slow ⇨ *performance failure*);
- ***Eavesdropping failure*** – a component leaks information to an outside entity;
- ***Response failure*** – a component's output is incorrect and this cannot be attributed to another component (wrong value is produced ⇨ *value failure*; wrong state emerges ⇨ *state transition failure*);
- ***Arbitrary (Byzantine) failure*** – a component may behave arbitrarily and multiple such components may collude.

Failures

Basic terms:

- ***Failure*** = a situation when a component does not live up to its specification;
- ***Error*** = that part of a component that can lead to a failure;
- ***Fault*** = the cause of an error.

Failures

Dealing with faults:

- ***Fault prevention*** – prevent the occurrence of a fault;
- ***Fault removal*** – reduce the presence, number, and seriousness of faults;
- ***Fault forecasting*** – estimate the present number, future incidence, and consequences of faults;
- ***Fault tolerance*** – build components so that they can *mask* the presence of faults.

Failures

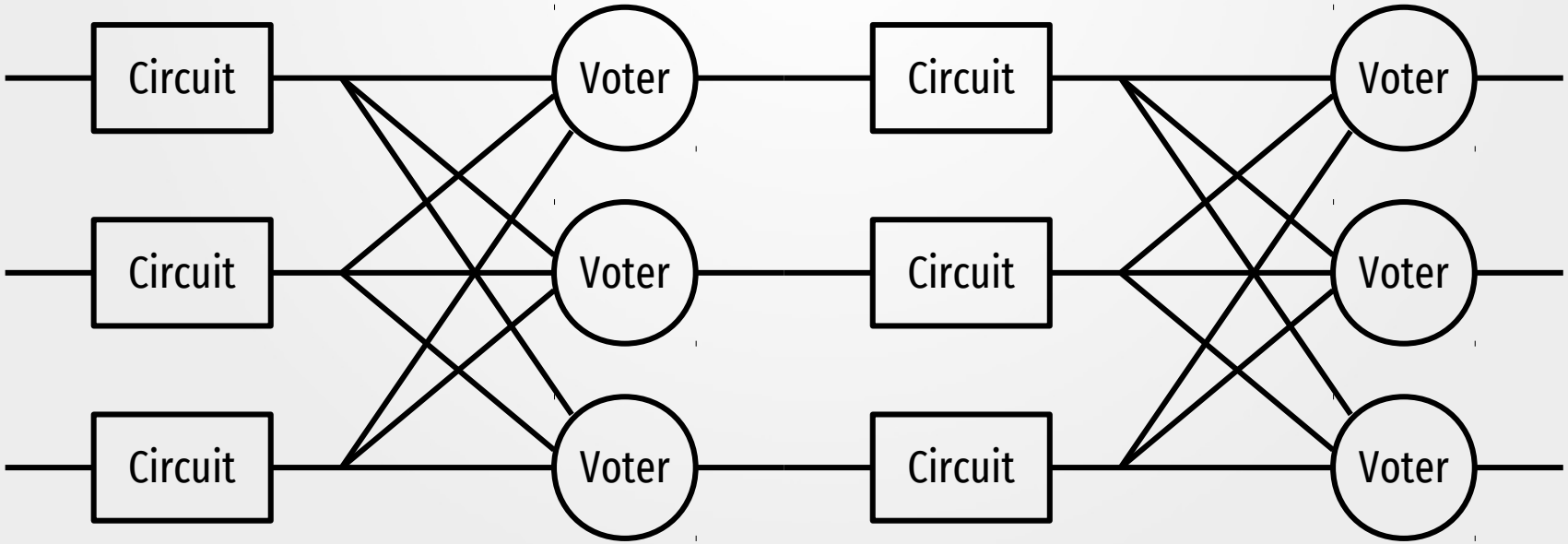
Dealing with faults:

- ***Fault prevention*** – prevent the occurrence of a fault;
- ***Fault removal*** – reduce the presence, number, and seriousness of faults;
- ***Fault forecasting*** – estimate the present number, future incidence, and consequences of faults;
- ***Fault tolerance*** – build components so that they can *mask* the presence of faults.

Failures

Fault masking is done through **redundancy**:

- Information redundancy,
- Time redundancy,
- Physical redundancy.



Modeling Failures

- A failure of a process occurs whenever the process does not behave according to its algorithm.
- (In contrast, we will model link failures shortly.)
- Our unit of failure is an entire process (and its memory).
- A process experiencing a failure is referred to as ***faulty***.
- Any other process is referred to as ***correct***.

Modeling Failures

- Algorithms normally assume that only a given number of processes, f, may be faulty.
- The relation between f and N is called ***resilience***.

Modeling Failures

- Algorithms normally assume that only a given number of processes, f, may be faulty.
- The relation between f and N is called ***resilience***.

Question: What can we manipulate to provide the right level of resilience?

Modeling Failures

- Algorithms normally assume that only a given number of processes, f, may be faulty.
- The relation between f and N is called ***resilience***.

Question: What can we manipulate to provide the right level of resilience?

Question: How to choose the right configuration?

Modeling Failures

We model the following failure types:

- Crashes,
- Crashes with recoveries,
- Byzantine failures.

Modeling Failures

We model the following failure types:

- Crashes,
- Crashes with recoveries,
- Byzantine failures.

- A process executes its algorithm correctly, including sending the required messages.
- At some point, it crashes:
  - stops executing any computation and
  - does not send any more messages.
- Never again is it part of the system (even if it recovers).
- This model is referred to as ***crash-stop***.

Modeling Failures

We model the following failure types:

- Crashes,
- Crashes with recoveries,
- Byzantine failures.

- A crashed process may recover.
- Upon recovery its transient memory is reset; persistent storage is intact.
- A process is considered faulty iff:
  - crashes and never recovers or
  - crashes and recovers infinitely often.
- This model is referred to as ***crash-recovery***.
- Is sometimes tricky to handle well.

Modeling Failures

We model the following failure types:

- Crashes,
- Crashes with recoveries,
- Byzantine failures.

- A faulty process may behave arbitrarily due to, for instance:
  - malicious intention
  - bugs.
- This model is referred to as ***arbitrary-fault***.
- Is the most expensive to handle well.
- We will focus on faults of intentional or malicious nature, handling which often requires cryptographic primitives.

Cryptographic Primitives

- Algorithms operating in untrusted environments must protect their messages.
- We consider three cryptographic primitives:
  - hash functions,
  - message-authentication codes (MACs),
  - digital signatures.
- Cryptographic abstractions typically require some keys to be present at the corresponding processes.
  - Key management is a topic on its own.
  - It is outside the scope of this lecture.
- We assume that processes hold the relevant keys.

Cryptographic Primitives

A ***cryptographic hash function*** maps a bit string of an arbitrary length to a short, "unique" representation.

- Provides only a single operation *H*:
  - Takes a bit string *x* of an arbitrary length;
  - Produces a value h, which is a short bit string of a fixed length.
- It is *collision-free*:
  - No process (not even Byzantine one) can find two distinct values *x* and *x′* such that *H(x) = H(x′)*.
- It is assumed that such functions are fast.

Cryptographic Primitives

A ***message authentication code*** (***MAC***) authenticates data between two entities using a shared symmetric key known only to the sender and receiver of the data but nobody else.

- Provides two operations:
  - *authenticate(p, q, m)* → *a* – invoked by process *p* for message *m* sent to process *q*, returns authenticator *a* for the message;
  - *verifyauth(q, p, m, a)* → *true* | *false* – invoked by process *q* for message *m* received from process *p* and putative authenticator *a*, returns *true* iff *a* is the actual authenticator for the message or *false* otherwise.
- It is fabrication-resistant:
  - No other entity than the sender and the receiver is able to craft a message that has never been authenticated and to produce an authenticator that the receiver accepts as valid during verification.
- In practice, MACs are based on hash functions, stream ciphers, or block ciphers.
- It is assumed that they can be computed and verified fast.

Cryptographic Primitives

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

- Provides two operations:
  - *sign(p, m)* → *s* – invoked by process *p* on message *m*, returns signature *s*;
  - *verifysig(q, m, s)* → *true* | *false* – invoked by any process on message *m* and putative signature *s* of process *q*, returns *true* iff *q* has previously invoked *sign(q, m)* obtaining *s* in return or *false* otherwise.
- It is also fabrication resistant.
- In practice, relies on asymmetric cryptography with public-key/private-key pairs:
  - The private key is given to the signing entity and must remain secret;
  - The public key is accessible to anyone.
- Adds considerable computational overhead compared to the symmetric cryptography.

Cryptographic Primitives

Question: What is the advantage then of digital signature schemes over those based on MACs?

Cryptographic Primitives

Question: What is the advantage then of digital signature schemes over those based on MACs?

- A MAC behaves like an oral message exchanged between people.
- A digital signature scheme models the exchange of unforgeable written messages.

Communication Links

- Our bi-directional all-to-all communication links are ***logical***.
- They can be ***physically*** implemented in different ways.

Communication Links

Physical links experience failures:

- message loss,
- message duplication,
- adversarial behaviors (e.g., message injection).

Modeling Communication Links

Each link provides two operations:

- send – invoked by the process from which the link exits;
- deliver – invoked by the process to which the link enters.

We show various link models depending on their assumptions on failures:

- fair-loss links,
- stubborn links,
- perfect links,
- logged perfect links,
- authenticated perfect links.

Modeling Communication Links

Fair-loss links satisfy the following properties:

FLL1: ***fair loss***: If a correct process, p, infinitely often sends a message, m, to a correct process, q, then q delivers m an infinite number of times.

FLL2: ***finite duplication***: If a correct process, p, sends a message, m, a finite number of times to a correct process, q, then m is not delivered an infinite number of times by q.

FLL3: ***no creation***: If some process, q, delivers a message, m, with sender p, then m must have been previously sent to q by process p.

Modeling Communication Links

Stubborn links satisfy the following properties:

SL1: ***stubborn delivery***: If a correct process, p, sends a message, m, to a correct process, q, (and never crashes afterward), then q delivers m an infinite number of times.

SL2: ***no creation***: If some process, q, delivers a message, m, with sender p, then m must have been previously sent to q by process p.

Modeling Communication Links

Perfect links satisfy the following properties:

PL1: ***reliable delivery***: If a correct process, p, sends a message, m, to a correct process, q, (and never crashes afterward), then q eventually delivers m.

PL2: ***no duplication***: No message is delivered by a process more than once.

PL3: ***no creation***: If some process, q, delivers a message, m, with sender p, then m must have been previously sent to q by process p.

Modeling Communication Links

Logged perfect links satisfy the following properties:

LPL1: ***reliable delivery***: If a correct process sends a message, m, to a correct process, q, and never crashes afterward, then q eventually log-delivers m.

LPL2: ***no duplication***: No message is log-delivered by a process more than once.

LPL3: ***no creation***: If some process, q, log-delivers a message, m, with sender p, then m must have been previously sent to q by process p.

Note: We will learn what "log-deliver" mean shortly.

Modeling Communication Links

Authenticated perfect links satisfy the following properties:

AL1: ***reliable delivery***: If a correct process, p, sends a message, m, to a correct process, q, (and never crashes afterward), then q eventually delivers m.

AL2: ***no duplication***: No message is delivered by a process more than once.

AL3: ***authenticity***: If some correct process, q, delivers a message, m, with sender p and process p is correct, then m must have been previously sent to q by p.

Modeling Communication Links

These are only models: when moving to concrete implementations many additional issues should be considered, such as:

- network topology,
- flow control,
- heterogeneity.

Timing

- An important part in an analysis of a distributed system is the behavior of its processes and link with the passage of time.
- There is no global clock in control of the processes...
- ... but perhaps sometimes we would like to assume something about time bounds on communication and (relative) process execution speeds.

Modeling Timing

Wrt. timing, the following common system models are considered:

- asynchronous systems,
- synchronous systems,
- eventually (always) synchronous systems.

Modeling Timing

In an ***asynchronous system***, there are no assumptions about timing:

- Computations can take arbitrarily long to complete.
- Message propagation delays are unbounded.

Modeling Timing

In a ***synchronous system***, the following properties are assumed to hold simultaneously:

- Synchronous computation – the time taken to execute a step is always less than some known upper bound.
- Synchronous communication – the period between the time a message is sent and the time the message is delivered by the destination process is always less than some known upper bound.

Modeling Timing

Asynchronous system model:

- is widely applicable (minimal assumptions)...
- ... but some key problems are proven to not have solutions is this model.

Synchronous system model:

- Enables providing several useful features, such as:
  - timed failure detection,
  - measure of message transit delays,
  - coordination based on time (e.g., leases),
  - worst-case performance inference,
  - synchronized clocks...
- ... but is difficult to ensure in practice.

Question: What about practical systems?

Modeling Timing

- Practical systems appear to be synchronous:
  - For *most* systems we know of, it is relatively easy to define physical time bounds that are respected *most of the time*.
- Yet, there are periods when they are asynchronous:
  - network is overloaded,
  - some process has memory shortage,
  - ...
- We may say that they are *partially synchronous*.

Modeling Timing

- To capture partial synchrony, we may assume that timing assumptions hold not always but only *eventually always*.
  - There is a time from which these assumptions hold forever but this time is not known.
- This is what is referred to as ***eventually (always) synchronous system*** model.

Modeling Timing

- To capture partial synchrony, we may assume that timing assumptions hold not always but only *eventually always*.
  - There is a time from which these assumptions hold forever but this time is not known.
- This is what is referred to as ***eventually (always) synchronous system*** model.

Question: What are the practical consequences of the model?

Modeling Timing

In practice, assuming an eventually always synchronous system model

does NOT mean that:

- There is a time from which the modeled system must be synchronous forever.
- The system needs to be initially asynchronous, and then only after some long period becomes synchronous.

captures that:

- The system need not only be synchronous and there is no bound on the period in which it is asynchronous.
- Yet, we expect that there are periods during which the system is synchronous and some of them are *long enough* for an algorithm to do something useful or terminate.

Modeling Timing

- The fact that a system is synchronous or eventually always synchonous could be captured by adding timing guarantees to process and link models.
- This, however, would lead to complex specifications.
- Therefore, instead, one typically encapsulates timing assumptions in models of supplementary functionalities, notably:
  - Failure detection,
  - Leader election.

Modeling Timing

- The fact that a system is synchronous or eventually always synchonous could be captured by adding timing guarantees to process and link models.
- This, however, would lead to complex specifications.
- Therefore, instead, one typically encapsulates timing assumptions in models of supplementary functionalities, notably:
  - Failure detection,
  - Leader election.

Question: For what failure types does failure detection make little sense?

Failure Detection

- The goal of failure detection is identifying faulty processes.
- Under crash-stop failures, crashes can be detected using timeouts.
- Such a way of detecting failures can be modeled as:
  - A *perfect failure detector*, in case of synchronous systems;
  - An *eventual failure detector* in case of eventually (always) synchronous systems.

Failure Detection

A ***perfect failure detector*** provides a single operation:

- *crash(p)* – detects at a process that another process, *p*, has crashed.

It satisfies two properties:

PFD1: ***strong completeness***: Eventually, every process that crashes is permanently detected by every correct process.

PFD2: ***strong accuracy***: If a process, p, is detected by any process, then *p* must have crashed.

Failure Detection

An ***eventual failure detector*** provides two operations:

- *suspect(p)* – notifies a process that another process, *p*, is suspected to have crashed.
- *restore(p)* – notifies a process that another process, *p*, is not suspected anymore.

It satisfies two properties:

EFD1: ***strong completeness***: Eventually, every process that crashes is permanently suspected by every correct process.

EFD2: ***eventual strong accuracy***: Eventually, no correct process is ever suspected by any correct process.

Leader Election

- Some algorithms need a distinguished process.
- Such a process should be selected *dynamically* from the population of the participating processes.
- This is the goal of *leader election*.
- This functionality requires timeouts or randomness, and some unique features of the processes (e.g., IDs or weights).
- We model it as:
  - *A perfect leader elector* in case of synchronous systems with crash failures,
  - *An eventual leader elector* in case of eventually (always) synchonous systems with crash failures,
  - *A Byzantine leader elector* in case of eventually (always) synchronous systems with arbitrary failures.

Leader Election

A ***perfect leader elector*** provides a single operation:

- *leader(p)* – indicates to a process that another process, *p*, is elected as leader.

It satisfies two properties:

PLE1: ***eventual detection***: Either there is no correct process or some correct process is eventually elected as the leader.

PLE2: ***strong accuracy***: If a process is leader, then all previously elected leaders must have crashed.

Leader Election

An ***eventual leader elector*** also provides a single operation:

- *trust(p)* – indicates to a process that another process, *p*, is trusted to be leader.

It satisfies two properties:

ELE1: ***eventual accuracy***: There is a time after which every correct process trusts some correct process.

ELE2: ***eventual agreement***: There is a time after which no two correct processes trust different correct processes.

Leader Election

A ***Byzantine leader elector*** also provides two operations:

- *trust(p)* – indicates to a process that another process, *p*, is trusted to be leader.
- *complain(p)* – issues a complaint against process *p*.

It satisfies two properties:

BLE1: ***eventual succession***: If more than *f* correct processes that trust some process, *p*, complain about *p*, then every correct process eventually trusts a different process than *p*.

BLE2: ***putsch resistance***: A correct process does not trust a new leader unless at least one correct process has complained against the previous leader.

BLE3: ***eventual agreement***: There is a time after which no two correct processes trust different processes.

Common Model Combinations

Typically, the following combinations of the previous models are considered:

- Fail-stop,
- Fail-noisy,
- Fail-silent,
- Fail-recovery,
- Fail-arbitrary.

Common Model Combinations

Typically, the following combinations of the previous models are considered:

- Fail-stop,
- Fail-noisy,
- Fail-silent,
- Fail-recovery,
- Fail-arbitrary.

Combines the following:

- Crash-stop failures,
- Perfect links,
- Perfect failure detector.

Substantially simplifies algorithm design...

... but poorly reflects reality.

Common Model Combinations

Typically, the following combinations of the previous models are considered:

- Fail-stop,
- Fail-noisy,
- Fail-silent,
- Fail-recovery,
- Fail-arbitrary.

Combines the following:

- Crash-stop failures,
- Perfect links,
- Eventually (always) perfect failure detector or eventual leader elector.

An intermediate case between the previous and the next combination.

Common Model Combinations

Typically, the following combinations of the previous models are considered:

- Fail-stop,
- Fail-noisy,
- Fail-silent,
- Fail-recovery,
- Fail-arbitrary.

Combines the following:

- Crash-stop failures,
- Perfect links,
- No failure detection or leader election.

Processes have no means to get any information about other processes having crashed.

Common Model Combinations

Typically, the following combinations of the previous models are considered:

- Fail-stop,
- Fail-noisy,
- Fail-silent,
- Fail-recovery,
- Fail-arbitrary.

Combines the following:

- Crash-recovery failures,
- Stubborn links,
- (Optionally) eventual leader elector.

Algorithms have to cope with process amnesia, to which end stable storage is used.

Common Model Combinations

Typically, the following combinations of the previous models are considered:

- Fail-stop,
- Fail-noisy,
- Fail-silent,
- Fail-recovery,
- Fail-arbitrary.

Combines the following:

- Byzantine failures,
- Authenticated perfect links,
- (Optionally) Byzantine leader elector.

The most difficult combination.

Issue

How to model software implementing distributed algorithms?

Software Composition Model

- For the initial algorithms, we will give actual pseudo-code to facilitate their understanding.
- Further into the lecture, we will give pseudo-code only for key parts, if at all, as algorithms will get increasingly involved.
- For the pseudo-code, we will use a variant of an event-based software composition model, somewhat resembling those from the labs.

Software Composition Model

- Every process hosts a collection of software components called *modules*.
- Each module:
  - is identified by a name,
  - offers an event-based interface for other modules,
  - has its behavior (with respect to the interface) characterized by a set of properties.
- Modules are composed into larger abstractions.

Software Composition Model

- Internally, a module:
  - has its private state (inaccessible to other modules),
  - is implemented as a state-machine, whose transitions are triggered by events,
  - can send events to other modules.
- Modules thus communicate only by events (in-process messages).
- The same mechanism is used to communicate with modules of other processes.
- Events can carry attributes.

Software Composition Model

- Each event is processed through a dedicated handler of a target module by the process hosting the module.
- Events of a module that are triggered by the same module are handled in the order in which they were triggered.
- Every process executes event handlers in a *mutually exclusive manner*.
- Some handlers may be qualified with a condition on the local variables of a module: the runtime system buffers external events until the condition on internal variables becomes satisfied.
- There are two special events:
  - *Init* – generated automatically by a process for each module that it hosts whenever the process starts (for the first time).
  - *Restore* – in the crash-recovery model, generated automatically by a process for each module that it hosts whenever the process restarts after a crash.

Software Composition Model

Examples:

upon event ⟨mod_1, $Event_1$ | $att_{1,1}$, $att_{1,2}$, ...⟩ **do**
  do something;
  trigger ⟨mod_2, $Event_2$ | $att_{2,1}$, $att_{2,2}$, ...⟩; *//< send some event to mod_2*

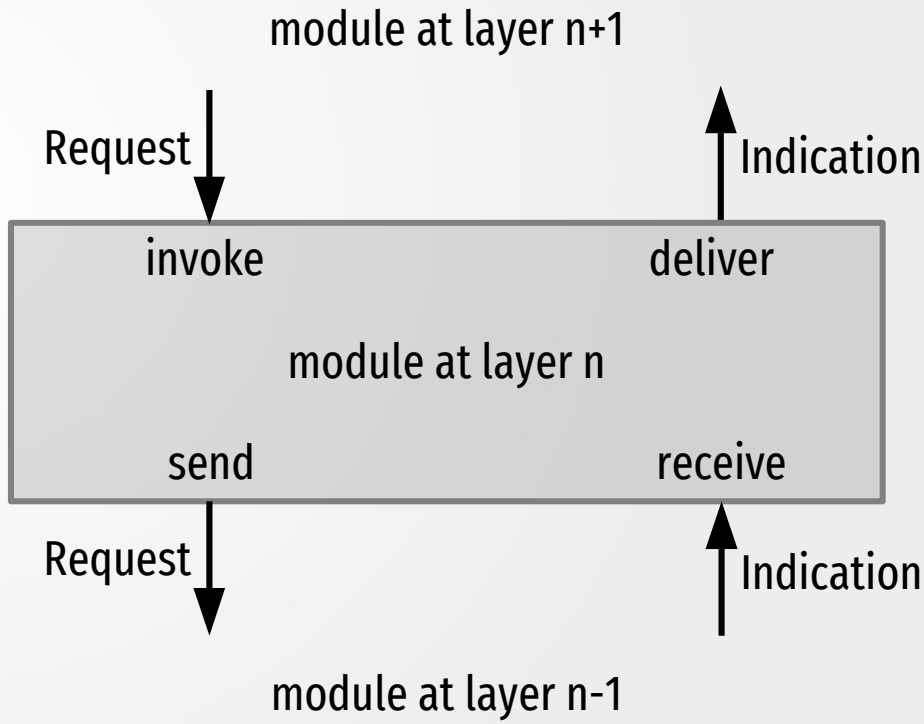
upon *condition* **do** *//< an internal event*
  do something;

upon event ⟨mod_3, $Event_3$ | $att_{3,1}$, $att_{3,2}$, ...⟩ **such that** *condition* **do** *//< a conditional event*
  do something;

Software Composition Model

Conceptually, there are two types of events:

- *Request* events – used by a module to *invoke* a service of another module or *signal* a condition to another module.
- *Indication* events – used by a module to deliver information or to signal a condition to another module.



Software Composition Model

Example of a module (its specification):

Module:
Name: JobHandler, **instance** *jh*.

Events:
Request: ⟨*jh*, *Submit* | *job*⟩: Requests a job to be processed.
Indication: ⟨*jh*, *Confirm* | *job*⟩: Confirms that the given job has been processed.

Properties:
JH1: *guaranteed response:* Every submitted job is eventually confirmed.

Software Composition Model

Example of an algorithm implementing the module:

Algorithm: Synchronous Job Handler

Implements:
JobHandler, **instance** *jh*.

upon event ⟨*jh*, *Submit* | *job*⟩ **do**
process(*job*);
trigger ⟨*jh*, *Confirm* | *job*⟩;

Software Composition Model

Example of an algorithm implementing the module:

Algorithm: Synchronous Job Handler

Implements:
JobHandler, **instance** *jh*.

upon event ⟨*jh*, *Submit* | *job*⟩ **do**
process(*job*);
trigger ⟨*jh*, *Confirm* | *job*⟩;

Question: Does it satisfy JH1 (guaranteed response)?

JH1: *guaranteed response*: Every submitted job is eventually confirmed.

Software Composition Model

Example of an algorithm implementing the module:

Algorithm: Asynchronous Job Handler

Implements:
JobHandler, **instance** *jh*.

upon event ⟨*jh*, *Init*⟩ **do**
buffer := ∅;

upon event ⟨*jh*, *Submit* | *job*⟩ **do**
buffer := *buffer* ∪ { job };

upon *buffer* ≠ ∅ **do**
job := *selectjob*(*buffer*);
process(*job*);
buffer := *buffer* \ { job };
trigger ⟨*jh*, *Confirm* | *job*⟩;

Software Composition Model

Example of an algorithm implementing the module:

Algorithm: Asynchronous Job Handler

Implements:
JobHandler, **instance** *jh*.

upon event ⟨*jh*, *Init*⟩ **do**
buffer := ∅;

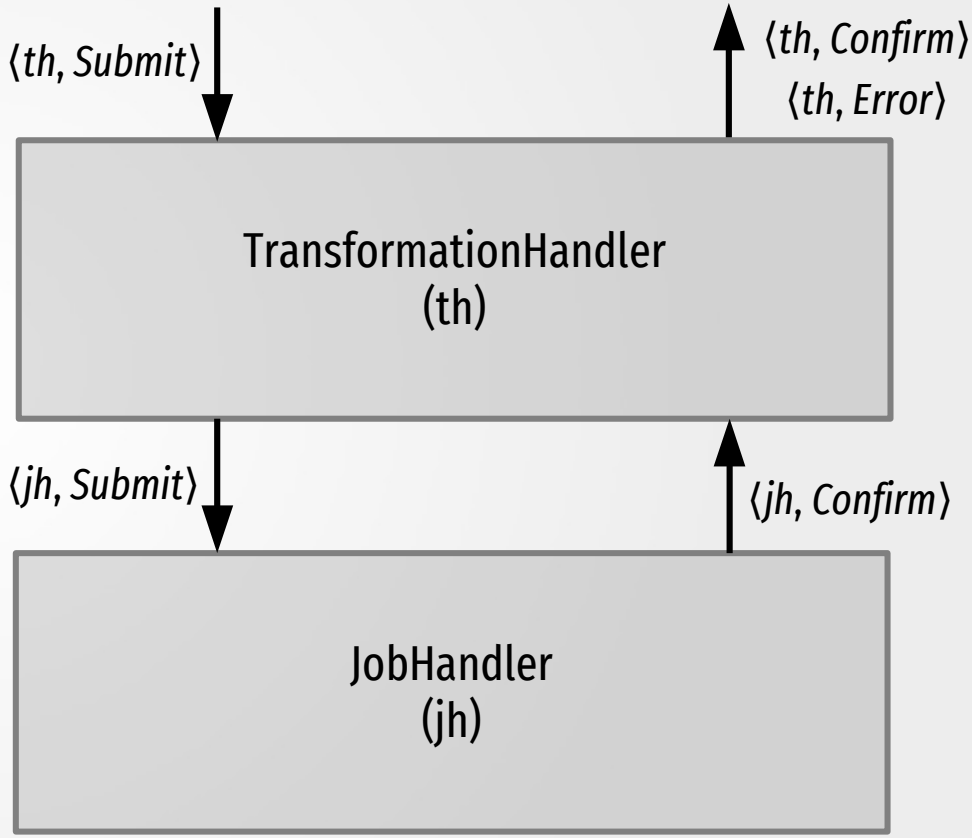
upon event ⟨*jh*, *Submit* | *job*⟩ **do**
buffer := *buffer* ∪ { job };

upon *buffer* ≠ ∅ **do**
job := *selectjob*(*buffer*);
process(*job*);
buffer := *buffer* \ { job };
trigger ⟨*jh*, *Confirm* | *job*⟩;

Question: Does it satisfy JH1 (guaranteed response)?

Software Composition Model

- To demonstrate software composition, we will extend the previous module with a layer on top.
- The layer will apply arbitrary transformation to a job before the job is processed.
- We will assume that such a transformation may fail.



Software Composition Model

Another example of a module (its specification):

Module:
Name: TransformationHandler, **instance** *th*.

Events:
Request: ⟨*th*, *Submit* | *job*⟩: Submits a job for transformation and processing.
Indication: ⟨*th*, *Confirm* | *job*⟩: Confirms that the given job has been transformed and processed.
Indication: ⟨*th*, *Error* | *job*⟩: Indicates that the transformation of the given job has failed.

Properties:
TH1: *guaranteed response:* Every submitted job is eventually confirmed or its transformation fails.
TH2: *soundness:* A submitted job whose transformation fails is not processed.

Software Composition Model

Another example of an algorithm:

Algorithm: Job Transformation by Buffering

Implements:
TransformationHandler, **instance** *th*.

Uses:
JobHandler, **instance** *jh*.

upon event ⟨*th*, *Init*⟩ **do**
first := 1;
num := 0;
handling := FALSE;
buffer := $[\perp]^M$;

Software Composition Model

Another example of an algorithm:

Algorithm: Job Transformation by Buffering

Implements:
TransformationHandler, **instance** *th*.

Uses:
JobHandler, **instance** *jh*.

upon event ⟨*th*, *Init*⟩ **do**
first := 1;
num := 0;
handling := FALSE;
buffer := $[\perp]^M$;

upon event ⟨*th*, *Submit* | *job*⟩ **do**
if *num* = *M* **then**
trigger ⟨*th*, *Error* | *job*⟩;
else
buffer[(*first* + *num*) mod *M* + 1] := *job*;
num := *num* + 1;

Software Composition Model

Another example of an algorithm:

Algorithm: Job Transformation by Buffering

Implements:
TransformationHandler, **instance** *th*.

Uses:
JobHandler, **instance** *jh*.

upon event ⟨*th*, *Init*⟩ **do**
  first := 1;
  num := 0;
  handling := FALSE;
  buffer := $[\perp]^M$;

upon event ⟨*th*, *Submit* | *job*⟩ **do**
  if *num* = *M* **then**
    trigger ⟨*th*, *Error* | *job*⟩;
  else
    buffer[(*first* + *num*) mod *M* + 1] := *job*;
    num := *num* + 1;

upon *num* > 0 ∧ *handling* = FALSE **do**
  job := *buffer*[*first*];
  first := (*first* + 1) mod *M* + 1;
  num := *num* - 1;
  if transform(*job*) **then**
    handling = TRUE;
    trigger ⟨*jh*, *Submit* | *job*⟩;
  else
    trigger ⟨*th*, *Error* | *job*⟩;

Software Composition Model

Another example of an algorithm:

Algorithm: Job Transformation by Buffering

Implements:
TransformationHandler, **instance** *th*.

Uses:
JobHandler, **instance** *jh*.

upon event ⟨*th*, *Init*⟩ **do**
  first := 1;
  num := 0;
  handling := FALSE;
  buffer := $[\perp]^M$;

upon event ⟨*jh*, *Confirm* | *job*⟩ **do**
  handling := FALSE;
  trigger ⟨*th*, *Confirm* | *job*⟩;

upon event ⟨*th*, *Submit* | *job*⟩ **do**
  if *num* = *M* **then**
    trigger ⟨*th*, *Error* | *job*⟩;
  else
    buffer[(*first* + *num*) mod *M* + 1] := *job*;
    num := *num* + 1;

upon *num* > 0 ∧ *handling* = FALSE **do**
  job := *buffer*[*first*];
  first := (*first* + 1) mod *M* + 1;
  num := *num* - 1;
  if transform(*job*) **then**
    handling = TRUE;
    trigger ⟨*jh*, *Submit* | *job*⟩;
  else
    trigger ⟨*th*, *Error* | *job*⟩;

Software Composition Model

Another example of an algorithm:

Algorithm: Job Transformation by Buffering

Implements:
TransformationHandler, **instance** *th*.

Uses:
JobHandler, **instance** *jh*.

upon event ⟨*th*, *Init*⟩ **do**
  first := 1;
  num := 0;
  handling := FALSE;
  buffer := $[\perp]^M$;

upon event ⟨*jh*, *Confirm* | *job*⟩ **do**
  handling := FALSE;
  trigger ⟨*th*, *Confirm* | *job*⟩;

upon event ⟨*th*, *Submit* | *job*⟩ **do**
  if *num* = *M* **then**
    trigger ⟨*th*, *Error* | *job*⟩;
  else
    buffer[(*first* + *num*) mod *M* + 1] := *job*;
    num := *num* + 1;

upon *num* > 0 ∧ *handling* = FALSE **do**
  job := *buffer*[*first*];
  first := (*first* + 1) mod *M* + 1;
  num := *num* - 1;
  if transform(*job*) **then**
    handling = TRUE;
    trigger ⟨*jh*, *Submit* | *job*⟩;
  else
    trigger ⟨*th*, *Error* | *job*⟩;

Question: Does it satisfy TH1 and TH2?

Revisiting Previous Models

Let's get our hands dirty...

Revisiting Communication Links

Module:
Name: FairLossLinks, **instance** *fll*.

Events:
Request: ⟨*fll*, *Send* | *q*, *m*⟩: Requests to send message *m* to process *q*.
Indication: ⟨*fll*, *Deliver* | *p*, *m*⟩: Delivers message *m* from process *p*.

Properties:
FLL1: *fair loss*: If a correct process, *p*, infinitely often sends a message, *m*, to a correct process, *q*, then *q* delivers *m* an infinite number of times.
FLL2: *finite duplication*: If a correct process, *p*, sends a message, *m*, a finite number of times to a correct process, *q*, then *m* is not delivered an infinite number of times by *q*.
FLL3: *no creation*: If some process, *q*, delivers a message, *m*, with sender *p*, then *m* must have been previously sent to *q* by process *p*.

Revisiting Communication Links

Module:
Name: StubbornLinks, **instance** *sl*.

Events:
Request: ⟨*sl*, *Send* | *q*, *m*⟩: Requests to send message *m* to process *q*.
Indication: ⟨*sl*, *Deliver* | *p, m*⟩: Delivers message *m* from process *p*.

Properties:
SL1: *stubborn delivery*: If a correct process, *p*, sends a message, *m*, to a correct process, *q*, (and never crashes afterward), then *q* delivers *m* an infinite number of times.
SL2: *no creation*: If some process, *q*, delivers a message, *m*, with sender *p*, then *m* must have been previously sent to *q* by process *p*.

Revisiting Communication Links

Algorithm: Retransmit Forever

Implements:
StubbornLinks, **instance** *sl*.

Uses:
FairLossLinks, **instance** *fll*.

upon event ⟨*sl*, *Init*⟩ **do**
sent := ∅;
starttimer(Δ);

upon event ⟨*Timeout*⟩ **do**
forall (*q*, *m*) ∈ *sent* **do**
trigger ⟨*fll*, *Send* | *q*, *m*⟩;
starttimer(Δ);

upon event ⟨*sl*, *Send* | *q*, *m*⟩ **do**
trigger ⟨*fll*, *Send* | *q*, *m*⟩;
sent := *sent* ∪ { (*q*, *m*) };

upon event ⟨*fll*, *Deliver* | *p*, *m*⟩ **do**
trigger ⟨*sl*, *Deliver* | *p*, *m*⟩;

Revisiting Communication Links

Module:
Name: PerfectLinks, **instance** *pl*.

Events:
Request: ⟨*pl*, *Send* | *q*, *m*⟩: Requests to send message *m* to process *q*.
Indication: ⟨*pl*, *Deliver* | *p*, *m*⟩: Delivers message *m* from process *p*.

Properties:
PL1: *reliable delivery*: If a correct process, *p*, sends a message, *m*, to a correct process, *q*, (and never crashes afterward), then *q* eventually delivers *m*.
PL2: *no duplication*: No message is delivered by a process more than once.
PL3: *no creation*: If some process, *q*, delivers a message, *m*, with sender *p*, then *m* must have been previously sent to *q* by process *p*.

Revisiting Communication Links

Algorithm: Eliminate Duplicates

Implements:
PerfectLinks, **instance** *pl*.

Uses:
StubbornLinks, **instance** *sl*.

upon event ⟨ *pl*, *Init* ⟩ **do**
delivered := ∅;

upon event ⟨ *pl*, *Send* | *q*, *m* ⟩ **do**
trigger ⟨ *sl*, *Send* | *q*, *m* ⟩;

upon event ⟨ *sl*, *Deliver* | *p*, *m* ⟩ **do**
if *m* ∉ *delivered* **then**
delivered := *delivered* ∪ { *m* };
trigger ⟨ *pl*, *Deliver* | *p*, *m* ⟩;

Revisiting Communication Links

Module:
Name: LoggedPerfectLinks, **instance** *lpl*.

Events:
Request: ⟨*lpl*, *Send* | *q*, *m*⟩: Requests to send message *m* to process *q*.
Indication: ⟨*lpl*, *Deliver* | *delivered*⟩: Notifies of potential updates to variable *delivered* in stable storage.

Properties:
LPL1: *reliable delivery*: If a correct process sends a message, *m*, to a correct process, *q*, and never crashes afterward, then *q* eventually log-delivers *m*.
LPL2: *no duplication*: No message is log-delivered by a process more than once.
LPL3: *no creation*: If some process, *q*, log-delivers a message, *m*, with sender *p*, then *m* must have been previously sent to *q* by process *p*.

Revisiting Communication Links

Algorithm: Log Deliver

Implements:
LoggedPerfectLinks, **instance** *lpl*.

Uses:
StubbornLinks, **instance** *sl*.

upon event ⟨*lpl*, *Init*⟩ **do**
delivered := ∅;
store(*delivered*);

upon event ⟨*lpl*, *Recovery*⟩ **do**
retrieve(*delivered*);
trigger ⟨*lpl*, *Deliver* | *delivered*⟩;

upon event ⟨*lpl*, *Send* | *q*, *m*⟩ **do**
trigger ⟨*sl*, *Send* | *q*, *m*⟩;

upon event ⟨*sl*, *Deliver* | *p*, *m*⟩ **do**
If not exists (*p'*, *m'*) ∈ *delivered* **such that** m' = m **then**
delivered := *delivered* ∪ { (*p*, *m*) };
store(*delivered*);
trigger ⟨*pl*, *Deliver* | *delivered*⟩;

Revisiting Communication Links

Module:
Name: AuthenticatedPerfectLinks, **instance** *al*.

Events:
Request: ⟨*al*, *Send* | *q*, *m*⟩: Requests to send message *m* to process *q*.
Indication: ⟨*al*, *Deliver* | *p*, *m*⟩: Delivers message *m* from process *p*.

Properties:
AL1: *reliable delivery*: If a correct process, *p*, sends a message, *m*, to a correct process, *q*, (and never crashes afterward), then *q* eventually delivers *m*.
AL2: *no duplication*: No message is delivered by a process more than once.
AL3: *authenticity*: If some correct process, *q*, delivers a message, *m*, with sender *p* and process *p* is correct, then *m* must have been previously sent to *q* by *p*.

Revisiting Communication Links

Algorithm: Authenticate and Filter

Implements:
AuthenticatedPerfectLinks, **instance** *al*.

Uses:
StubbornLinks, **instance** *sl*.

upon event ⟨*al*, *Init*⟩ **do**
delivered := ∅;

upon event ⟨*al*, *Send* | *q*, *m*⟩ **do**
a := *authenticate*(*self*, *q*, *m*);
trigger ⟨*sl*, *Send* | *q*, [*m*, *a*]⟩;

upon event ⟨*sl*, *Deliver* | *p*, [*m*, *a*]⟩ **do**
if $m \notin delivered \land verifyauth(self, p, m, a)$ **then**
delivered := *delivered* ∪ { *m* };
trigger ⟨*al*, *Deliver* | *p*, *m*⟩;

Revisiting Previous Models

Let's get our hands even more dirty...

Revisiting Previous Models

Assume for a while crash-stop failures.

Revisiting Failure Detection

Module:
Name: PerfectFailureDetector, **instance** $\mathcal{P}$.

Events:
Indication: $\langle \mathcal{P}, \mathit{Crash} \mid p \rangle$: Detects that process p has crashed.

Properties:
PFD1: *strong completeness*: Eventually, every process that crashes is permanently detected by every correct process.
PFD2: *strong accuracy*: If a process, p, is detected by any process, then p must have crashed.

Revisiting Failure Detection

Algorithm: Exclude on Timeout

Implements:
PerfectFailureDetector, **instance** $\mathcal{P}$.

Uses:
PerfectLinks, **instance** *pl*.

upon event ⟨ $\mathcal{P}$, *Init* ⟩ **do**
  alive := Π;
  detected := $\emptyset$;
  starttimer(Δ);

upon event ⟨ *Timeout* ⟩ **do**
  forall $p \in \Pi$ **do**
    if $p \notin alive \wedge p \notin detected$ **then**
      detected := *detected* ∪ { *p* };
      trigger ⟨ $\mathcal{P}$, *Crash* | *p* ⟩;
    trigger ⟨ *pl*, *Send* | *p*, [HeartbeatRequest] ⟩;
  alive := $\emptyset$;
  starttimer(Δ);

upon event ⟨ *pl*, *Deliver* | *q*, [HeartbeatRequest] ⟩ **do**
  trigger ⟨ *pl*, *Send* | *q*, [HeartbeatReply] ⟩;

upon event ⟨ *pl*, *Deliver* | *p*, [HeartbeatReply] ⟩ **do**
  alive := *alive* ∪ { *p* };

Revisiting Failure Detection

Correctness of the algorithm:

- Keep in mind that:
  - Perfect links do not lose, duplicate, or invent messages.
  - Processing and communication time is bounded (the system is synchronous).
  - Failures are crash-stop.
- Assume that:
  - The timeout, Δ, is “large enough.”
- Strong completeness:
  - If a process crashes, it stops replying to heartbeat messages.
  - Every process will stop delivering its messages (perfect links).
  - Every correct process will thus detect the crash.
- Strong accuracy:
  - A crash of a process, *p*, is detected only if another process, *q*, does not deliver a message from *p* before the timeout period.
  - This can happen only if *p* has crashed because the algorithm, the synchrony, and the timeout configuration together ensure that otherwise *p* must have sent a message and the message must have been delivered by *q* before the timeout period.

Revisiting Failure Detection

Performance of the algorithm:

- The work of the algorithm (i.e., the number of messages) is $2N^2$ per timeout period.
- The span of the algorithm is proportional to the timeout period, which is in turn at least 2 message communication rounds (request-reply). [We assume best-case communication scheduling across different links.]

Revisiting Failure Detection

Algorithm: Exclude on Timeout

Implements:
PerfectFailureDetector, **instance** $\mathcal{P}$.

Uses:
PerfectLinks, **instance** *pl*.

upon event ⟨ $\mathcal{P}$, *Init* ⟩ **do**
  alive := Π;
  detected := $\emptyset$;
  starttimer(Δ);

upon event ⟨ *Timeout* ⟩ **do**
  forall $p \in \Pi$ **do**
    if $p \notin alive \wedge p \notin detected$ **then**
      detected := *detected* ∪ { *p* };
      trigger ⟨ $\mathcal{P}$, *Crash* | *p* ⟩;
    trigger ⟨ *pl*, *Send* | *p*, [HeartbeatRequest] ⟩;
  alive := $\emptyset$;
  starttimer(Δ);

upon event ⟨ *pl*, *Deliver* | *q*, [HeartbeatRequest] ⟩ **do**
  trigger ⟨ *pl*, *Send* | *q*, [HeartbeatReply] ⟩;

upon event ⟨ *pl*, *Deliver* | *p*, [HeartbeatReply] ⟩ **do**
  alive := *alive* ∪ { *p* };

Question: What would happen if we used one-way heartbeats?

Revisiting Failure Detection

Module:
Name: EventualFailureDetector, **instance** $\Diamond\mathcal{P}$.

Events:
Indication: $\langle \Diamond\mathcal{P}, \mathit{Suspect} \mid p \rangle$: Notifies that process p is suspected to have crashed.
Indication: $\langle \Diamond\mathcal{P}, \mathit{Restore} \mid p \rangle$: Notifies that process p is not suspected anymore.

Properties:
EFD1: *strong completeness*: Eventually, every process that crashes is permanently suspected by every correct process.
EFD2: *eventual strong accuracy*: Eventually, no correct process is ever suspected by any correct process.

Revisiting Failure Detection

Algorithm: Increasing Timeout

Implements:
EventualFailureDetector, **instance** $\Diamond\mathcal{P}$.

Uses:
PerfectLinks, **instance** *pl*.

upon event $\langle \Diamond\mathcal{P}, \mathit{Init} \rangle$ **do**
  $\mathit{alive} := \Pi$;
  $\mathit{suspected} := \emptyset$;
  $\mathit{delay} := \Delta$;
  starttimer(*delay*);

upon event $\langle \mathit{pl}, \mathit{Deliver} \mid q, [\text{HeartbeatRequest}] \rangle$ **do**
  trigger $\langle \mathit{pl}, \mathit{Send} \mid q, [\text{HeartbeatReply}] \rangle$;

upon event $\langle \mathit{pl}, \mathit{Deliver} \mid p, [\text{HeartbeatReply}] \rangle$ **do**
  $\mathit{alive} := \mathit{alive} \cup \{ p \}$;

upon event $\langle \mathit{Timeout} \rangle$ **do**
  if $\mathit{alive} \cap \mathit{suspected} \neq \emptyset$ **then**
    $\mathit{delay} := \mathit{delay} + \Delta$;
  forall $p \in \Pi$ **do**
    if $p \notin \mathit{alive} \wedge p \notin \mathit{suspected}$ **then**
      $\mathit{suspected} := \mathit{suspected} \cup \{ p \}$;
      trigger $\langle \Diamond\mathcal{P}, \mathit{Suspect} \mid p \rangle$;
    else if $p \in \mathit{alive} \wedge p \in \mathit{suspected}$ **then**
      $\mathit{suspected} := \mathit{suspected} \setminus \{ p \}$;
      trigger $\langle \Diamond\mathcal{P}, \mathit{Restore} \mid p \rangle$;
    trigger $\langle \mathit{pl}, \mathit{Send} \mid p, [\text{HeartbeatRequest}] \rangle$;
  $\mathit{alive} := \emptyset$;
  starttimer(*delay*);

Revisiting Failure Detection

Correctness of the algorithm:

- Keep in mind that:
  - Perfect links do not lose, duplicate, or invent messages.
  - Processing and communication time is eventually always bounded but we do not know this bound (the system is eventually always synchronous).
- Strong completeness (like previously):
  - If a process crashes, it stops replying to heartbeat messages.
  - Consequently, every correct process will detect the crash and never revisit its judgment.
- Eventual strong accuracy:
  - Consider the moment time in time from which the system becomes synchronous and the timeout delay becomes “sufficiently large.”
  - Starting from this moment, for any heartbeat request message sent by a process to a correct process, a heartbeat reply message is delivered within the timeout period.
  - In effect, any correct process that is wrongly suspecting another correct process will revise its suspicion and never raise it again.

Revisiting Failure Detection

Performance of the algorithm:

- The work of the algorithm (i.e., the number of messages) is $2N^2$ per timeout period, this time variable one (may be set more aggressively).
- The span of the algorithm is proportional to the timeout period, which is in turn at least 2 message communication rounds (request-reply).

Revisiting Leader Election

Module:
Name: PerfectLeaderElector, **instance** *ple*.

Events:
Indication: ⟨*ple*, *Leader* | *p*⟩: Notifies that process *p* is elected as leader.

Properties:
PLE1: *eventual detection*: Either there is no correct process or some correct process is eventually elected as the leader.
PLE2: *accuracy*: If a process is leader, then all previously elected leaders must have crashed.

Revisiting Leader Election

Algorithm: Monarchical Leader Election

Implements:
PerfectLeaderElector, **instance** *ple*.

Uses:
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨*ple*, *Init*⟩ **do**
suspected := ∅;
leader := ⊥;

upon event ⟨$\mathcal{P}$, *Crash* | *p*⟩ **do**
suspected := *suspected* ∪ { *p* };

upon *leader* ≠ *maxrank*(*Π* \ *suspected*) **do**
leader := *maxrank*(*Π* \ *suspected*);
trigger ⟨*ple*, *Leader* | *leader*⟩;

Where:

$maxrank(S) = \arg\max_{p \in S} \{ rank(p) \}$

in addition assuming that:

$maxrank(\emptyset) = \bot$.

Revisiting Leader Election

Correctness of the algorithm:

- Keep in mind that:
  - Failures are crash-stop.
  - We have a perfect failure detector, $\mathcal{P}$.
- Eventual detection (follows from strong completeness of $\mathcal{P}$):
  - If a process crashes, every correct process will detect the crash.
  - Therefore, if the crashed process is the leader, a new leader will be elected...
  - ... unless there are no correct processes left.
- Accuracy (follows from strong accuracy of $\mathcal{P}$):
  - If p's crash is detected, then p must have crashed.
  - The detection monotonically changes the set of suspected processes.
  - A new leader is elected only at the beginning or when this set changes.
  - A crashed process never recovers.
  - Therefore, if a leader is elected, all previous leaders must have crashed.

Revisiting Leader Election

Performance of the algorithm:

- The work of the algorithm is 0 (there is no communication).
- The span of the algorithm is also 0 (again, only local operations are involved).

Revisiting Leader Election

Module:
Name: EventualLeaderElector, **instance** Ω.

Events:
Indication: $\langle \Omega, \mathit{Trust} \mid p \rangle$: Indicates that process p is trusted to be leader.

Properties:
ELE1: *eventual accuracy*: There is a time after which every correct process trusts some correct process.
ELE2: *eventual agreement*: There is a time after which no two correct processes trust different correct processes.

Revisiting Leader Election

Algorithm: Monarchical Eventual Leader Election

Implements:
EventualLeaderElector, **instance** Ω.

Uses:
EventualFailureDetector, **instance** $\Diamond\mathcal{P}$.

upon event $\langle \Omega, Init \rangle$ **do**
$suspected := \emptyset$;
$leader := \perp$;

upon event $\langle \Diamond\mathcal{P}, Suspect \mid p \rangle$ **do**
$suspected := suspected \cup \{ p \}$;

upon event $\langle \Diamond\mathcal{P}, Restore \mid p \rangle$ **do**
$suspected := suspected \setminus \{ p \}$;

upon $leader \neq maxrank(\Pi \setminus suspected)$ **do**
$leader := maxrank(\Pi \setminus suspected)$;
trigger $\langle \Omega, Trust \mid leader \rangle$;

Revisiting Leader Election

Correctness of the algorithm:

- Keep in mind that:
  - Failures are crash-stop.
  - We have an eventual failure detector, $\Diamond\mathcal{P}$.
- Eventual accuracy (follows from strong completeness of $\Diamond\mathcal{P}$):
  - A process does not trust a process that it suspects.
  - There is a time after which a process permanently suspects every process that has crashed.
- Eventual agreement (follows from the previous property plus strong accuracy of $\Diamond\mathcal{P}$):
  - Every correct process eventually aways suspects exactly the set of crashed processes.

Revisiting Leader Election

Performance of the algorithm (the same as for Monarchical Leader Election):

- The work of the algorithm is 0 (there is no communication).
- The span of the algorithm is also 0 (again, only local operations are involved).

Revisiting Previous Models

What about crash-recovery failures?

Revisiting Previous Models

- Failure detection does not make a lot of sense...
- ... but we can implement leader election.
- Assumptions:
  - Crash-stop or crash-recovery failures,
  - Eventually always synchronous system,
  - At least one process is correct (i.e., eventually never crashes).
- Idea:
  - Every process maintains an epoch number, keeping track of the number of recoveries.
  - The goal is to elect a process that has crashed and recovered the least often.
  - Like previously, an increasing timeout is employed to discover the communication bounds when the system eventually becomes synchronous forever.
- For selecting the leader, we have function *select*(*S*) that operates on a set, *S*, of pairs (*process*, *epoch*) and works as follows:
  - Consider those pairs in *S* that have the lowest epoch.
  - Among those pairs, select the process with the highest *rank*.

Revisiting Leader Election

Module:
Name: EventualLeaderElector, **instance** Ω.

Events:
Indication: $\langle \Omega, \mathit{Trust} \mid p \rangle$: Indicates that process p is trusted to be leader.

Properties:
ELE1: *eventual accuracy*: There is a time after which every correct process trusts some correct process.
ELE2: *eventual agreement*: There is a time after which no two correct processes trust different correct processes.

Revisiting Leader Election

Algorithm: Elect Lower Epoch

Implements:
EventualLeaderElector, **instance** Ω.

Uses:
PerfectLinks, **instance** *pl*.

upon event $\langle \Omega, Init \rangle$ **do**
epoch := 0;
store(*epoch*);
candidates := $\emptyset$;
trigger $\langle \Omega, Recovery \rangle$;

Revisiting Leader Election

Algorithm: Elect Lower Epoch

Implements:
EventualLeaderElector, **instance** Ω.

Uses:
PerfectLinks, **instance** *pl*.

upon event $\langle \Omega, Init \rangle$ **do**
  $epoch := 0$;
  store(*epoch*);
  $candidates := \emptyset$;
  trigger $\langle \Omega, Recovery \rangle$;

upon event $\langle \Omega, Recovery \rangle$ **do**
  $leader := maxrank(\Pi)$;
  trigger $\langle \Omega, Trust \mid leader \rangle$;
  $delay := \Delta$;
  retrieve(*epoch*);
  $epoch := epoch + 1$;
  store(*epoch*);
  forall $p \in \Pi$ **do**
    trigger $\langle pl, Send \mid p, [\text{Heartbeat}, epoch, \text{TRUE}] \rangle$;
  $candidates := \emptyset$;
  starttimer(*delay*);

Revisiting Leader Election

Algorithm: Elect Lower Epoch

Implements:
EventualLeaderElector, **instance** Ω.

Uses:
PerfectLinks, **instance** *pl*.

upon event $\langle \Omega, Init \rangle$ **do**
  $epoch := 0$;
  store(*epoch*);
  $candidates := \emptyset$;
  trigger $\langle \Omega, Recovery \rangle$;

upon event $\langle pl, Deliver \mid q, [\text{Heartbeat}, ep, echo] \rangle$ **do**
  if exists $(s, e) \in candidates$ **such that**
    $s = q \land e < ep$ **then**
    $candidates := candidates \setminus \{ (q, e) \}$;
  $candidates := candidates \cup \{ (q, ep) \}$;
  if *echo* **then**
    trigger $\langle pl, Send \mid q, [\text{Heartbeat}, epoch, \text{FALSE}] \rangle$;

upon event $\langle \Omega, Recovery \rangle$ **do**
  $leader := maxrank(\Pi)$;
  trigger $\langle \Omega, Trust \mid leader \rangle$;
  $delay := \Delta$;
  retrieve(*epoch*);
  $epoch := epoch + 1$;
  store(*epoch*);
  forall $p \in \Pi$ **do**
    trigger $\langle pl, Send \mid p, [\text{Heartbeat}, epoch, \text{TRUE}] \rangle$;
  $candidates := \emptyset$;
  starttimer(*delay*);

Revisiting Leader Election

Algorithm: Elect Lower Epoch

Implements:
EventualLeaderElector, **instance** Ω.

Uses:
PerfectLinks, **instance** *pl*.

upon event $\langle \Omega, Init \rangle$ **do**
  $epoch := 0$;
  store(*epoch*);
  $candidates := \emptyset$;
  trigger $\langle \Omega, Recovery \rangle$;

upon event $\langle pl, Deliver \mid q, [\text{Heartbeat}, ep, echo] \rangle$ **do**
  if exists $(s, e) \in candidates$ **such that**
      $s = q \land e < ep$ **then**
    $candidates := candidates \setminus \{ (q, e) \}$;
  $candidates := candidates \cup \{ (q, ep) \}$;
  if *echo* **then**
    trigger $\langle pl, Send \mid q, [\text{Heartbeat}, epoch, \text{FALSE}] \rangle$;

upon event $\langle \Omega, Recovery \rangle$ **do**
  $leader := maxrank(\Pi)$;
  trigger $\langle \Omega, Trust \mid leader \rangle$;
  $delay := \Delta$;
  retrieve(*epoch*);
  $epoch := epoch + 1$;
  store(*epoch*);
  forall $p \in \Pi$ **do**
    trigger $\langle pl, Send \mid p, [\text{Heartbeat}, epoch, \text{TRUE}] \rangle$;
  $candidates := \emptyset$;
  starttimer(*delay*);

upon event $\langle Timeout \rangle$ **do**
  $newleader := select(candidates)$;
  if $newleader \neq leader$ **then**
    $delay := delay + \Delta$;
    $leader := newleader$;
    trigger $\langle \Omega, Trust \mid leader \rangle$;
  forall $p \in \Pi$ **do**
    trigger $\langle pl, Send \mid p, [\text{Heartbeat}, epoch, \text{TRUE}] \rangle$;
  $candidates := \emptyset$;
  starttimer(*delay*);

Revisiting Leader Election

Correctness of the algorithm – eventual accuracy:

- Assume, by contradiction, that there is a time after which a correct process *p* repeatedly trusts some faulty processes.
- Consider any such process, *q*. Two cases are possible:
  - Case (1): Process *q* eventually crashes and never recovers.
    - Process *q* will send a finite number of heartbeat messages to *p*.
    - There is a time after which *p* stops delivering these messages (*no creation* and *no duplication* of *pl*).
    - Eventually *q* will be permanently excluded from *p*'s candidate lists and *p* will never select *q* as the leader.
  - Case (2): Process *q* keeps crashing and recovering forever.
    - Its epoch number keeps increasing forever.
    - For any correct process *r*, there is a time after which its epoch number is forever lower than that of *q*. After this time either:
      - Process *p* will permanently stop delivering heartbeat messages from *q* (if it crashes and recovers too fast to send any message), or
      - Process *p* will deliver some heartbeat messages from *q* but with a higher epoch number than that of *r*.
    - In either case, *p* will never select *q* as the leader.
- In both Case (1) and (2), process *q* is thus eventually never selected by *p* as the leader- Contradiction!

Revisiting Leader Election

Correctness of the algorithm – eventual agreement:

- Let $C \subseteq \Pi$ be a subset of all correct process in a given execution of the algorithm.
- Consider the time moment after which *all* of the following hold:
  - the system becomes synchronous,
  - the processes in C never crash,
  - the epoch number of each process in C stops increasing,
  - the epoch number of each candidate $p' \in C$ at any process in C is equal to the actual epoch of p',
  - for every $p \in C$ and every $q \in \Pi \setminus C$, either p stops delivering heartbeat messages from q, and hence removes q from its candidate set, or the epoch number of candidate q at p is strictly larger than the largest epoch of any process in C.
- Such a moment exists because of the partial synchrony, the definition of correct process under crash-recovery failures, the properties of perfect links, and the algorithm itself.
- After it is reached, every process in C can trust only a candidate from C and, for any such candidate $p' \in C$, whenever the timeout occurs at any process $p \in C$, the epoch number of p' at p is the same as the epoch number of p' at any other process $q \in C$ and never changes.
- Therefore, from the definition of function *select*, any two process $p, q \in C$ eventually always trust the same candidate $p' \in C$.

Revisiting Failure Detection

Performance of the algorithm:

- The work of the algorithm (i.e., the number of messages) is $2N^2$ per timeout period, this time variable one (may be set more aggressively).
- The span of the algorithm is proportional to the timeout period, which is in turn at least 2 message communication rounds (request-reply).
- The number of timeout periods, R, is unknown and may vary depending on the number of process crashes and recoveries and the moment the system becomes synchronous.
- The work and span thus have to be multiplied by R.

Revisiting Previous Models

What about Byzantine failures?

Revisiting Previous Models

- Under Byzantine failures, failure detection by timeouts makes little sense.
- We will use instead an approach referred to as “trust but verify:”
  - We assume that a process (e.g., a leader), should perform some action within some time bounds.
  - If the action is performed wrongly or outside these bounds, other correct processes detect this and complain.
  - If sufficiently many correct processes complain, the process is considered faulty.
- In an eventually always synchronous system, we may sometimes complain against correct but slow processes.
- Therefore, subsequent tested processes are given progressively more time, which is captured by cyclic (rotating) *rounds*.

Revisiting Leader Election

Module:
Name: ByzantineLeaderElector, **instance** *ble*.

Events:
Indication: ⟨*ble*, *Trust* | *p*⟩: Indicates that process *p* is trusted to be leader.
Request: ⟨*ble*, *Complain* | *p*⟩: Receives a complaint about process *p*.

Properties:
BLE1: *eventual succession*: If more than *f* correct processes that trust some process, *p*, complain about *p*, then every correct process eventually trusts a different process than *p*.
BLE2: *putsch resistance*: A correct process does not trust a new leader unless at least one correct process has complained against the previous leader.
BLE3: *eventual agreement*: There is a time after which no two correct processes trust different processes.

Revisiting Previous Models

- Assumptions:
  - Byzantine failures.
  - Eventually (always) synchronous system.
  - $N > 3 \cdot f$ processes, where f is the number of tolerated faulty processes.
- We have a function that derives the leader from a round number:

$$leader(r) = p \in \Pi \text{ such that } rank(p) \bmod N = r \bmod N$$

- For a set or a vector S, $\#(S)$ denotes the number of elements in the set or the number of vector elements that are non-$\perp$.

Revisiting Leader Election

Algorithm: Rotating Byzantine Leader Election

Implements:
ByzantineLeaderElector, **instance** *ble*.

Uses:
AuthenticatedPerfectLinks, **instance** *al*.

upon event ⟨*ble*, *Init*⟩ **do**
round := 1;
complainlist := $[\perp]^N$;
complained := FALSE;
trigger ⟨*ble*, *Trust* | *leader*(*round*)⟩;

upon event ⟨*ble*, *Complain* | *p*⟩ **such that**
p = *leader*(*round*) ∧ *complained* = FALSE **do**
complained := TRUE;
forall $q \in \Pi$ **do**
trigger ⟨*al*, *Send* | *q*, [Complaint, *round*]⟩;

upon event ⟨*al*, *Deliver* | *p*, [Complaint, *r*]⟩ **such that**
r = *round* ∧ *complainlist*[*p*] = ⊥ **do**
complainlist[*p*] := Complaint;
if #(*complainlist*) > f ∧ *complained* = FALSE **then**
complained := TRUE;
forall $q \in \Pi$ **do**
trigger ⟨*al*, *Send* | *q*, [Complaint, *round*]⟩;
else if #(*complainlist*) > $2 \cdot f$ **then**
round := *round* + 1;
complainlist := $[\perp]^N$;
complained := FALSE;
trigger ⟨*ble*, *Trust* | *leader*(*round*)⟩;

Revisiting Leader Election

Correctness of the algorithm:

- Keep in mind that:
  - Failures are Byzantine but there is a plenty of correct processes: $N > 3 \cdot f$.
  - Authenticated perfect links do not lose, duplicate, or invent messages.
  - Processing and communication time of correct processes is eventually always bounded but we do not know this bound (the system is eventually always synchronous).
- Eventual succession (is straightforward):
  - If at least $f + 1$ correct process complain, they send Compaint messages to the other processes.
  - The messages are delivered by all $N - f$ correct processes of which those that have not complained are triggered to join the complaining, in effect generating at least $N - f$ complaints in total.
  - Since $N - f > 2 \cdot f$, each correct process eventually changes its round, and hence the leader.
- Putsch resistance (is trivial):
  - There are at most f Byzantine processes, and thus, if no correct process complains, no correct process changes its round number because there are insufficient complaints to trigger doing so.
- Eventual agreement:
  - The correct processes eventually permanently stop complaining against a correct leader because it is given enough time to perform verification work.
  - When all complaints about the previous leaders have been delivered, all correct processes are in the same round, and hence have the same (correct) leader.

Revisiting Leader Election

Performance of the algorithm:

- The work of the algorithm can be N^2 per algorithm round.
- The span of the algorithm is 1 per algorithm round (sending complaints).
- The number of rounds, R, necessary for the leader to stabilize depends on the communication and computational bounds, and is thus unknown.
- The work and span thus have to be multiplied by R.

Revisiting Leader Election

- In general, algorithms for the fail-arbitrary model are typically more involved than those for the other models.
- Therefore, we will first omit this model when discussing distributed algorithms for the three fundamental abstractions...
- ... but return to Byzantine failures only when non-Byzantine algorithms for all these abstractions have been discussed.

Summary

We have:

- covered models of processes, memory, messages, links, failures, timing;
- showed how to combine them into popular system models;
- presented a notation for distributed algorithms;
- introduced cryptographic primitives, failure detection and leader election abstractions and algorithms;
- demonstrated how to approach analyzing their correctness and performance.

Next Lecture

- Will be about distributed algorithms for the first of the three fundamental abstractions: communication.
- More specifically, we will play with group communication: reliable broadcast.