Distribution, Abstractions, Processes, and Links

A Course on Distributed Algortihms,

Spring 2013, Computer Science dept., Lund University

Amr Ergawy, Jörn Janneck

2 April 2013

Outline

- Part1: Distribution and Abstractions.
- Part2: Processes and Links.

References

Chapter 1 and sections 2.1, 2.2, and 2.4 of the course book:

Christian Cachin, Rachid Guerraoui, and Luis Rodrigues,
Introduction to Reliable and Secure Distributed Programming,
Springer, 2011,
ISBN 3-642-15259-7

For demonstration purposes, all figures are copied an pasted from the course.

Part 1: Distribution and Abstraction

1. Motivation.
2. Abstractions.
3. Inherent distributions.
4. Artificial distribution.
5. The end-to-end argument.
6. Software components.
7. Layering of process modules.
8. Modules and algortihms.
9. Algorithms classification.

Motivation (1/2)

- The need to ***distributed programming abstraction*** = abstractions + problems + robustness + processes = agreement abstractions.
- Over ***a modular strategy*** based on APIs.
- ***Concrete API*** = notation + event based invocation.
- In addition to concurrent execution, distributed processes may stop, e.g. crashing or disconnecting => ***partial failures***.
- A failure in a computer that you do not know it exited may harm your own computer.

Motivation (2/2)

- In addition to partial failures, cooperation is aimed to be robust against ***adversarial attacks***.

- ***Ensuring robustness is difficult***:
  1. distinguish process failure from network failure.
  2. a process controlled by a malicious adversary.

- ***An example is the client/server computing***:
  1. a server process continue working on a server instead of a failed one.
  2. a server process shall not stuck on the failure of one its clients.

- ***In addition to client-server***: we have the multiparty model, i.e. multiple devices and combining patterns, e.g. a server interacts with other severs is a multiparty manner.

Abstractions (1/2)

- ***System models*** = processes + links. Chapter 2 discusses system models.
- ***Abstractions***: interaction cooperation/agreement problems.
- ***Difficulty of abstraction***: to capture partial failures + malicious behaviour.
- Abstractions capture ***the common among a significant range of systems:***
  - ***So that we do not re-invent the wheel for every slight variant of the same problem***.
- Abstracting the underlying physical system = a system model = elements + their properties + interactions = mainly two abstractions = processes + links.
- ***A Process***: a computer, a process, a thread, a trust domain, an administrative domain, ... etc.
- ***A link***: any communication network.

Abstractions (2/2)

- We focus on ***robust cooperation*** problems that are modeled ***as distributed agreement*** problems:

1. agreement on ***address and data format***, i.e. protocol.

2. agreeing ***a common plan/value***, i.e. the consensus problem.

3. agreeing ***whether a set of actions shall take place based on conditions***, e.g. the atomic commitment problem for distributed transactions.

4. agreeing ***the order in which a set of actions shall take place***, e.g. the total order broadcast problem to replicate data to achieve fault tolerance.

Inherent Distribution (1/3)

- ***We distinguish two origins of abstractions***:

  1. from ***natural/inherent distribution*** of the application.
  2. when distribution is ***an engineering choice***.

- ***Examples of inherent distribution***:

  - info processing engines.
  - multiuser cooperative systems.
  - distributed shared spaces.
  - process control.
  - cooperative editors.
  - distributed data bases.
  - distributed storage systems.

- In Info-dissemination pub-sub systems we need ***to disseminate the same set of messages to all subscribers of the same topic. Otherwise an unfair service***.

Inherent Distribution (2/3)

- *The cooperation abstraction in a pub-sub audio stream*:

  Dissemination is enough ***best-effort***, the subscriber ***can tolerate losing some*** messages.

- *In the case of a pub-sub system for stock info dissemination*:

  The subscriber is interested in ***every message***, requiring the ***reliable broadcast*** abstraction, ***or even in order***.

- *A set of processes controlling a manufacturing plant or a device, e.g. replicated or sharing a task, they provide sensor readings to the control algorithm*:

  They must ***agree an input value even*** if their associated sensors provide ***different readings*** or some of the processes have ***failed/crashed***. This is the ***consensus*** problem among these processes.

Inherent Distribution (3/3)

- *Sharing work space of a software or a doc, a distributed dialogue, an online chat, a virtual conference, etc*:

  A ***shared memory*** abstraction where processes ***agree the order of write and read*** operations to the shared space to maintain a ***consistent view*** of it.

- *In distributed databases, agreement abstractions help transactions managers to obtain a consistent view of the running transactions and can make consistent decisions on the transactions serialization*:

  The involved agreement transaction in the case of distributed databases is ***atomic commitment***.

- In distributed storage, a set of data is distributed over a set of storage nodes to overcome their limited capacity, i.e. natural distribution, and also to ensure system resilience, i.e. artifact distribution:

  The ***shared memory*** abstraction fits here, and the order of read/write operations is important for a consistent view.

Artificial Distribution

- ***Distribution may be an engineering decisions to achieve***:
  - fault tolerance.
  - load balancing.
  - fast sharing.

- ***Replication by distributions***:
  - service survives the failure of some replicas.
  - better performance.
  - service survives malicious attacks that stops some replicas.
  - replicas must be in a consistent state, for the illusion of one-service.
  - Two types of replicas:
    - ***deterministic***: full consistency to ensure that the ***same set of requests must reach all replicas in the same order***, requires ***total order broadcast***.
    - ***nondeterministic***: that requires a ***different ordering*** algorithm.
    - For both sets of abstractions, ***fault tolerance*** is the challenge.

The End to End Argument

- ***The end-to-end argument:***

  To push as much ***complexity as possible to the application*** level.

  I.e. to combine some of the functionality of the distributed abstractions with the application logic for optimization.

- ***Against this approach***

  - The distributed abstractions = complex + high inter-component dependency:

    The end-to-end monolithic approach is error-prone.

  - Depending on many factors, e.g. network and required quality of service, we chose a distributed abstraction:

    Oppositely to end-to-end, ***modular distribution*** **=** changing it as it fits, use-case/deployment environment and requirements.

Software Components (1/3)

- Interaction among processes and its components is ***asynchronous event based***.
- We use ***pseudo code*** is to describe distributed algorithms.
- ***Algorithms*** of the distributed abstraction are ***event handlers***.
- A component is ***a module = properties + events*** to send/receive, i.e. Interface.
- Components of a process = ***software stack***, the application at top and networking level is bottom. ***Distributed abstractions focuses on the components in-between***.

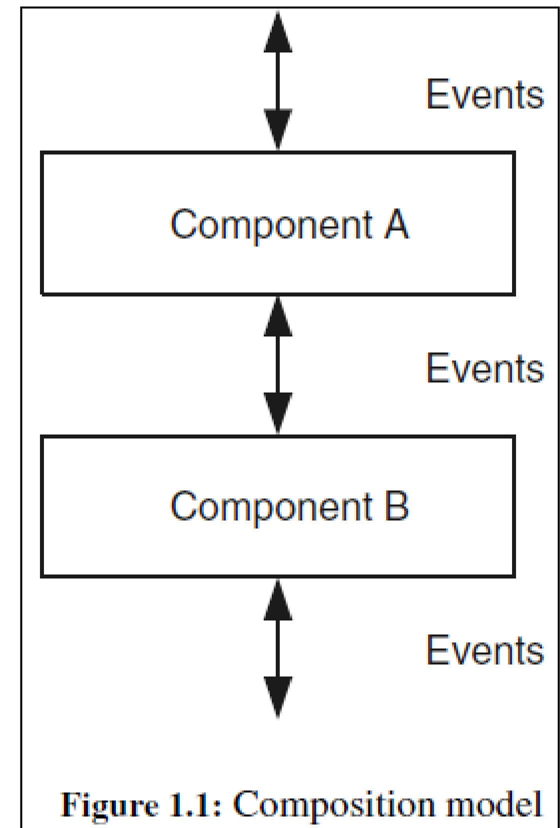

Figure 1.1: Composition model

Software Components (2/3)

- ***An event*** = type + attributes + intended for all components or only for one.

- A component may provide ***a handler*** for an event or it may ***filter*** it. A process handles ***only one event per time***. Once done with event, ***a process periodically checks for new events***. Implicitly assumed in pseudo code.

upon event $\langle co_1, Event_1 \mid att_1^1, att_1^2, \ldots \rangle$ **do**
  do something;
  trigger $\langle co_2, Event_2 \mid att_2^1, att_2^2, \ldots \rangle$;

upon event $\langle co_1, Event_3 \mid att_3^1, att_3^2, \ldots \rangle$ **do**
  do something else;
  trigger $\langle co_2, Event_4 \mid att_4^1, att_4^2, \ldots \rangle$;

- An event handler may ***trigger events*** for: same component, other components, other processes. ***FIFO ordering*** among ***components***, some ***other criteria among processes***.

- An event handler may ***handle conditions***, i.e. not external events maintained by local variables.

upon *condition* **do**
  do something;

Software Components (3/3)

- A handler may handle ***an external event only on an internal condition*** is satisfied.

upon event $\langle$ *co*, *Event* | $att_1^1, att_1^2, \ldots$ $\rangle$ **such that** *condition* **do**
    do something;

This may require ***buffering the external event*** in the run-time system. Which may request an unbounded buffer.

A solution:
- define a handler that ***stores a triggered external event in a local variable***.
- define a handler that is based only on the internal condition variable, that one recalls the locally stored external event upon the satisfaction of the condition.

Layering of Process Modules

- ***Requests***: events going downwards in the stack.
  - service-request or signaling messages.

- ***Indications***: events going downwards in the stack.
  - delivered data, e.g. message contents, or signaling messages.

- Both types of events may contain data payloads or just signaling info.

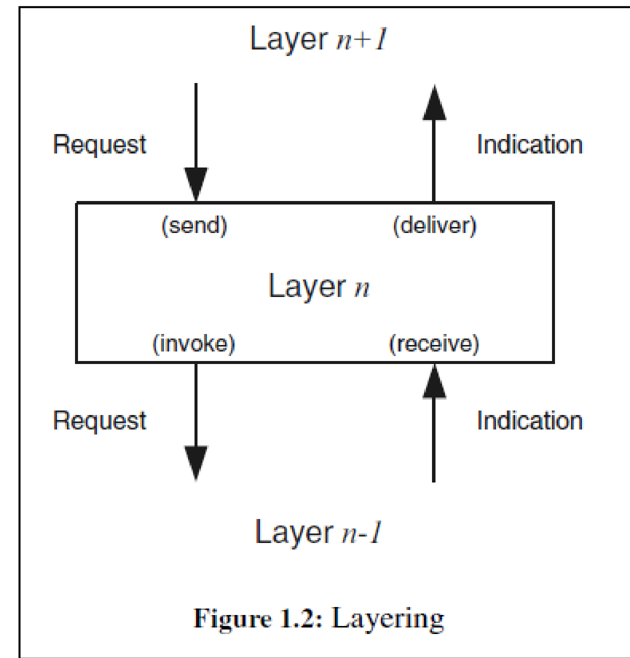

Figure 1.2: Layering

Modules and Algorithms

- Below, the ***interface*** and the ***properties*** of a job handler ***module***, and two ***implementations*** for it, ***a synchronous*** one and ***an asynchronous*** one.

Module 1.1: Interface and properties of a job handler

Module:

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

Events:

Request: $\langle$ *jh*, *Submit* | *job* $\rangle$: Requests a job to be processed.

Indication: $\langle$ *jh*, *Confirm* | *job* $\rangle$: Confirms that the given job has been (or will be) processed.

Properties:

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

Algorithm 1.1: Synchronous Job Handler

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

upon event $\langle$ *jh*, *Submit* | *job* $\rangle$ **do**
process(*job*);
trigger $\langle$ *jh*, *Confirm* | *job* $\rangle$;

Algorithm 1.2: Asynchronous Job Handler

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

upon event $\langle$ *jh*, *Init* $\rangle$ **do**
$buffer := \emptyset$;

upon event $\langle$ *jh*, *Submit* | *job* $\rangle$ **do**
$buffer := buffer \cup \{job\}$;
trigger $\langle$ *jh*, *Confirm* | *job* $\rangle$;

upon $buffer \neq \emptyset$ **do**
$job := selectjob(buffer)$;
process(*job*);
$buffer := buffer \setminus \{job\}$;

Algorithms Classification

- Classifications of distributed algorithms depends on the ***failure assumptions***, the ***environment and system parameters***, and ***other choices***.
- Not every distribution abstraction, e.g. shared space or best effort broadcast has a solution, i.e. an algorithm, of every algorithm class.

1. *fail-stop* algorithms, designed under the assumption that processes can fail by crashing but the crashes can be reliably detected by all the other processes;
2. *fail-silent* algorithms, where process crashes can never be reliably detected;
3. *fail-noisy* algorithms, where processes can fail by crashing and the crashes can be detected, but not always in an accurate manner (accuracy is only eventual);
4. *fail-recovery* algorithms, where processes can crash and later recover and still participate in the algorithm;
5. *fail-arbitrary* algorithms, where processes can deviate arbitrarily from the protocol specification and act in malicious, adversarial ways; and
6. *randomized* algorithms, where in addition to the classes presented so far, processes may make probabilistic choices by using a source of randomness.

Part 2: Processes and Links

1. Abstraction types.
2. Distributed computation: processes, messages, automata, steps, safety, and liveness.
3. Abstracting processes: models of process failure.
4. Abstracting communication: link models.

Abstraction Types

- The distributions abstractions and algorithms are not tied to a specific combination of system components, i.e. OS, file systems, middleware, ... etc.

- Three types of abstractions: ***processes, links, and failure detectors***, which are combined to form distributed system models.

- Because an abstraction is never generic enough to capture all ***varieties of physical systems***, we define ***different instances of an abstraction type***.

Distributed Computation (1/4): Processes and Messages

- ∏ is a typically ***static set of N process***, p, q, r, s, ... etc.
- ***A function rank***: ∏ -> {1, .., N} maps a process Id to an index.
- The process name ***self*** is the one that executes the algorithm.
- ***All processes run the same local algorithm***.
- Exchanged ***messages are uniquely identified***, e.g. (sender, sequence number).
- Messages are exchanged through ***communication links***.

Distributed Computation (2/4): Automata and Steps (1/2)

- A distributed algorithm is ***a collection of automata, one per process***. The automaton on a process ***defines how it reacts to a message***.

- The execution of a distributed algorithm is ***a sequence of steps executed by the processes***.

- We assume ***a global clock/scheduler that assigns time units to processes***, even if two algorithm steps execute on the same process.

- ***A process step*** = receiving + local computations + sending.

  - If any of the step components is not required by the algorithm, it is replaced by ***nil***: e.g. ***send nil*** message, ***receive nil*** message, or ***nil local computation***.

Distributed Computation (3/4): Automata and Steps (2/2)

- ***A distributed algorithm*** = ***computation steps***, including events among components, and ***communication steps***, for exchanging messages among processes. Considering ***timing assumptions*** of different steps.

- We focus on ***deterministic algorithms*** =

  (local computation step + state after computation + sending step)

  only based on (the prior state of the process + the received message).

- ***Non-deterministic algorithms*** may involve a randomized source.

Distributed Computation (4/4): Safety and Liveness

- ***A distributed algorithm*** may be executed in **a infinite number of inter-leavings of its steps**, however it must satisfy specific ***properties of the abstractions it implements***, which falls into classes ***safety*** and ***live-ness***.

- ***Safety*** is a property that once violated at time t, it is never satisfied again:
  - E.g. a ***perfect link does not initiate messages*** be itself.
  - To prove ***by contradiction***, identify ***a time t*** or ***a partial execution sequence*** that violaye the safety property.

- ***Live-ness*** is the property that eventually some thing good will happen.
  - E.g. a **perfect link eventually delivers a message**.
  - To prove that ***for any time t***, the property ***will be satisfied at time t' >= t***.

- ***A combined example:*** an inter-process communication service where ***every message is delivered, live-ness***, and ***not duplicated, safety***.

Abstracting Processes (1/3)

- The ***process is the unit of failure***, it fails all its components fail.

- ***Failure types***: crash-stop, omission, crash with recovery, eavesdropping, arbitrary.

- In ***a crash-stop*** process abstraction:
  - a process ***stops executing*** algorithm steps.
  - a process may recover but it is ***not any-more part of the running instance*** of the distributed algorithm.
  - An algorithm is designed with a ***resilience,*** measured in number of processes f that is assumed to may fail out of N processes.
  - It is also good practice to study ***which properties are preserved when failure goes beyond*** the resilience threshold.

Abstracting Processes (2/3)

- In ***omission-crash*** process abstraction:
  - process ***does not send/receive a message that is shall*** send/receive in a step of the algorithm. Mainly because of communication channel/buffer congestion/fullness.
  - We don't focus on omission faults, instead we generalize to crash-recovery, next.

- In ***crash-recovery*** process abstraction:
  - After-recovery, the failed process ***may re-participate*** in the running algorithm instance that is failed in.
  - a process ***may crash and recover for an infinite number of times***, ***losing messages*** as in omission crash, and ***losing its internal state***.
  - ***A stable storage*** of the internal state, called log, is used and accessed via store/retrieve operations. Shall minimize accessing it.

Abstracting Processes (3/3)

- In ***eavesdropping-crash*** process abstraction:
  - A process that its ***internal state or exchanging messages is eavesdropping failed***.
  - The minimal protection is encrypting the messages.
  - We do not focus on eavesdropping faults.

- In ***arbitrary-failure*** process abstraction:
  - ***No assumptions*** on the behavior of faulty processes.
  - Byzantine or malicious failures => a Byzantine/arbitrary-fault process
  - The ***most expensive to tolerate***, but the ***only suitable abstraction when unpredictable faults*** may occur, by malicious attacks or even bugs/errors.
  - ***We use defensive techniques***: Cyclic Redundancy Check for bugs/errors, and cryptographic measures for malicious attacks.

Absrtacting Communication

- The abstraction of ***a link is bidirectional*** between two processes.

- ***A distributed algorithm*** may refine the abstract network view to ***utilize a specific network topology***.

- ***Every message*** must include info to uniquely identify its sender.
  - ***Ensured*** in ***crash-stop*** and ***crash-recovery*** process abstractions.
  - In ***arbitrary/Byzantine faults***, this property may ***not be satisfied***, e.g. an adversary may ***insert messages*** to the network. Algorithms depend on ***cryptography*** to provide correct sender identification.

- Processes that exchange messages in ***a request-reply manner*** must provide ***a numbering scheme*** that associates which request or reply.

Link Failures (1/2)

- We assume that ***the probability that a message reaches its destination is non-zero***, eventually a message is delievered.

- Two main ways to overcome network failures:
  - Re-sending messages.
  - Using cryptography against adversary.

- For ***point-to-point*** communication, we introduce ***five link abstractions***.

- ***Link abstractions***:
  - ***fair-loss links***: a message not to be lost has a non-zero probability of delivery. For fail-stop process abstractions.
  - ***stubborn and perfect links***: retransmission over fair-loss. For fail-stop process abstractions.
  - ***logged perfect links***, for crash recovery process abstractions.
  - ***authenticated links***, for arbitrary failed processes.

Link Failures (2/2)

- The ***properties of each link abstraction*** are described in terms of two events:
  - a send request.
  - a deliver indication.

- The term ***“deliver” is favored over the term “receive”***, because we talk from the ***perspective of the link instead of the receiver*** process.

- Before a message is delivered to upper layers, it is ***stored in a buffer***, then an ***algorithm is executed to ensure the properties of the required link*** abstraction.

Fair-Loss Links

- The simplest abstraction - I think its properties are that of physical links.

Module 2.1: Interface and properties of fair-loss point-to-point links

Module:

Name: FairLossPointToPointLinks, **instance** *fll*.

Events:

Request: ⟨ *fll*, *Send* | q, m ⟩: Requests to send message m to process q.

Indication: ⟨ *fll*, *Deliver* | p, m ⟩: Delivers message m sent by 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 process q, then m cannot be delivered an infinite number of times by q.

FLL3: *No creation:* If some process q delivers a message m with sender p, then m was previously sent to q by process p.

Stubborn Links

- The stubborn property ***delivers a message an infinite number*** of times.

- The algorithm ***Retransmit Forever*** implements ***stubborn/fair-loss***:
  - ***Correctness***: stubborn properties are satisfied fair-loss properties.
  - ***Performance***: book-keeps all sent messages, not efficient.

Module 2.2: Interface and properties of stubborn point-to-point links

Module:

Name: StubbornPointToPointLinks, **instance** *sl*.

Events:

Request: $\langle$ *sl*, *Send* | q, m $\rangle$: Requests to send message m to process q.

Indication: $\langle$ *sl*, *Deliver* | p, m $\rangle$: Delivers message m sent by process p.

Properties:

SL1: *Stubborn delivery:* If a correct process p sends a message m once to a correct process q, 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 was previously sent to q by process p.

Algorithm 2.1: Retransmit Forever

Implements:
    StubbornPointToPointLinks, **instance** *sl*.

Uses:
    FairLossPointToPointLinks, **instance** *fll*.

upon event $\langle$ *sl*, *Init* $\rangle$ **do**
    sent := $\emptyset$;
    starttimer(Δ);

upon event $\langle$ *Timeout* $\rangle$ **do**
    forall $(q, m) \in$ *sent* **do**
        trigger $\langle$ *fll*, *Send* | q, m $\rangle$;
    starttimer(Δ);

upon event $\langle$ *sl*, *Send* | q, m $\rangle$ **do**
    trigger $\langle$ *fll*, *Send* | q, m $\rangle$;
    sent := *sent* $\cup \{(q, m)\}$;

upon event $\langle$ *fll*, *Deliver* | p, m $\rangle$ **do**
    trigger $\langle$ *sl*, *Deliver* | p, m $\rangle$;

Perfect Links

- The perfect link ***detect message duplicates*** and ***allows retransmission*** too. The properties ***reliable delivery*** and ***no duplication*** ensure the ***every sent message is delivered only once***.

- The algorithm ***Eliminate Duplicate*** implements ***perfect link/stubborn***:
  - ***Correctness***: reliable delivery/stubborn, no duplication/conditioned delivery, inherited no creation.
  - ***Performance***: book-keeps sent and delivered messages, not efficient.

  A solution: acknowledge delivered messages, and reject earlier time-stamped retransmissions.

Module 2.3: Interface and properties of perfect point-to-point links

Module:

Name: PerfectPointToPointLinks, **instance** *pl*.

Events:

Request: $\langle$ *pl*, *Send* | q, m $\rangle$: Requests to send message m to process q.

Indication: $\langle$ *pl*, *Deliver* | p, m $\rangle$: Delivers message m sent by process p.

Properties:

PL1: *Reliable delivery:* If a correct process p sends a message m to a correct process q, 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 was previously sent to q by process p.

Algorithm 2.2: Eliminate Duplicates

Implements:
PerfectPointToPointLinks, **instance** *pl*.

Uses:
StubbornPointToPointLinks, **instance** *sl*.

upon event $\langle$ *pl*, *Init* $\rangle$ **do**
delivered := $\emptyset$;

upon event $\langle$ *pl*, *Send* | q, m $\rangle$ **do**
trigger $\langle$ *sl*, *Send* | q, m $\rangle$;

upon event $\langle$ *sl*, *Deliver* | p, m $\rangle$ **do**
if $m \notin$ *delivered* **then**
delivered := *delivered* $\cup \{m\}$;
trigger $\langle$ *pl*, *Deliver* | p, m $\rangle$;

Logged Perfect Links

- ***Events are locally logged*** to the delivered storage.
- Same properties as the perfect link, with the exception of reliable-delivery that assumes ***a never-crash sender***, that does not use a stable store. ***Crash-recovery is assumed only on delivery***.
- The algorithm ***Log Delivered modifies*** the algorithm ***Eliminate Duplicates*** to implement a ***logged perfect link***:

  - The ***correctness and performance*** are the same of the "eliminate duplicates" algorithm, with the addition of considering the stable storage.

Module 2.4: Interface and properties of logged perfect point-to-point links

Module:

Name: LoggedPerfectPointToPointLinks, **instance** *lpl*.

Events:

Request: $\langle$ *lpl, Send* | q, m $\rangle$: Requests to send message m to process q.

Indication: $\langle$ *lpl, Deliver* | *delivered* $\rangle$: Notifies the upper layer of potential updates to variable *delivered* in stable storage (which log-delivers messages according to the text).

Properties:

LPL1: *Reliable delivery:* If a process that never crashes sends a message m to a correct process q, 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 was previously sent to q by process p.

Algorithm 2.3: Log Delivered

Implements:
LoggedPerfectPointToPointLinks, **instance** *lpl*.

Uses:
StubbornPointToPointLinks, **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 ⟨ lpl, Deliver | delivered ⟩;
```

Authenticated Perfect Links

- The difference between this property and the no-creation property of the perfect link is emphasizing the "correctness" of both sender or receiver processes as legitimate processes.

- The algorithm "***Authenticate and Filter***" implements ***authenticated perfect/stubborn***.
  - It uses a Message Authentication Code, MAC, to generate an authenticator for a message in terms of the sender, the message, and the destination.
  - The destination does the opposite.
  - The correctness and performance are inherited from eliminate duplicates, which authenticity is ensured by using the MAC.

Module 2.5: Interface and properties of authenticated perfect point-to-point links

Module:

Name: AuthPerfectPointToPointLinks, **instance** *al*.

Events:

Request: $\langle$ *al*, *Send* | q, m $\rangle$: Requests to send message m to process q.

Indication: $\langle$ *al*, *Deliver* | p, m $\rangle$: Delivers message m sent by process p.

Properties:

AL1: *Reliable delivery:* If a correct process sends a message m to a correct process q, then q eventually delivers m.

AL2: *No duplication:* No message is delivered by a correct 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 was previously sent to q by p.

Algorithm 2.4: Authenticate and Filter

Implements:
AuthPerfectPointToPointLinks, **instance** *al*.

Uses:
StubbornPointToPointLinks, **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 verifyauth(self, p, m, a) ∧ m ∉ delivered then
        delivered := delivered ∪ {m};
        trigger ⟨ al, Deliver | p, m ⟩;
```

Consideration on Link Abstractions

- The perfection of a link is more delegated to physical links or transport layer, e.g. TCP.
- In studying distributed algorithms, it is not relevant to get deep into the implementation details, but in practice it is relevant to get into these details, e.g. using sequence numbers and message retransmissions.
- Some times the upper layers may became responsible for some functionality, e.g. remembering the delivered messages in a more expensive logged perfect links.

Thanks! ☺