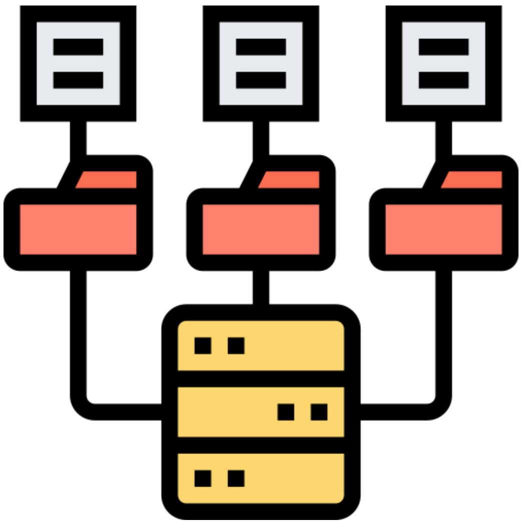


Dependable Distributed Systems

Professor: S. Bonomi

A.Y. 2022/23

Table of Contents

1 Modeling Distributed Systems 6

1.1 Modeling Processes and their interactions 6

1.2 Specification in terms of Safety and Liveness Property 6

1.3 Modeling Failures 6

1.4 Timing Assumptions 6

1.5 Abstracting Communication 6

1.5.1 Abstracting Link Failures 7

1.5.2 Fair-Loss Links 7

1.5.3 Stubborn Links 8

1.5.4 Perfect Links 9

1.5.5 Logged Perfect Links 11

1.5.6 Authenticated Perfect Links 13

2 Time in Distributed Systems 16

3 Logical Clock 17

4 Distributed Mutual Exclusion 18

5 Broadcast Communications 19

6 Consensus 20

7 Ordered Communications 21

8 Registers 22

9 Software Replication 23

10 Overview on Capacity Planning 24

11 Modeling the Workload of a System 25

12 Building a Performance Model 1 26

13 Building a Performance Model 2 27

14 Dependability Evaluation 28

15 Intro to Experimental Design 29

16 CAP Theorem 30

17 Consistency Criteria for Distributed Shared Memories 31

18 Publish-Subscribe Communication Paradigm **32**

19 Overlay Networks **33**

20 DLT and Blockchain **34**

21 Exercises **35**

22 Exams **36**

List of Figures

1 Summary on Timing Assumptions 6
2 The link abstraction and different instances 7
3 Interface of fair-loss point-to-point links 8
4 Interface of stubborn point-to-point links 8
5 Algorithm 2.1 - Retransmit Forever 9
6 Interface of perfect point-to-point links 10
7 Algorithm 2.2 - Eliminate Duplicate 10
8 Interface of logged perfect point-to-point links 12
9 Algorithm 2.3 - Log Delivered 13
10 Interface of Authenticated perfect point-to-point links 14
11 Algorithm 2.4 - Authenticate and Filter 14

List of Tables

1 Modeling Distributed Systems

1.1 Modeling Processes and their interactions

1.2 Specification in terms of Safety and Liveness Property

1.3 Modeling Failures

1.4 Timing Assumptions

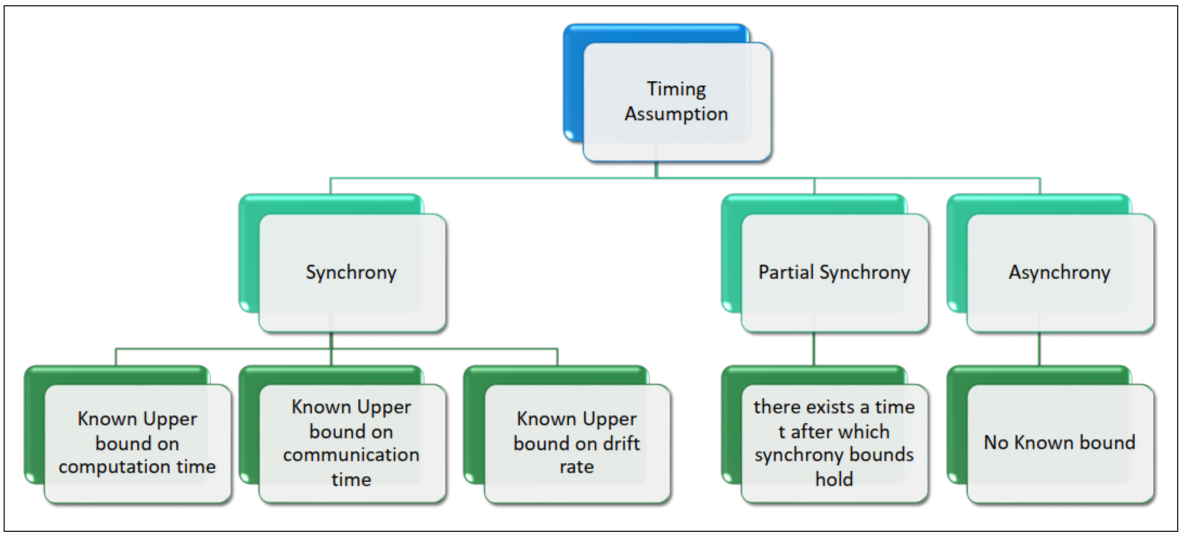

Timing Assumption
Synchrony
Partial Synchrony
Asynchrony
Known Upper bound on computation time
Known Upper bound on communication time
Known Upper bound on drift rate
there exists a time t after which synchrony bounds hold
No Known bound

Figure 1: Summary on Timing Assumptions

1.5 Abstracting Communication

The abstraction of a *link* is used to represent the network components of the distributed system.

Every pair of processes is connected by a bidirectional link, a topology that provides full connectivity among the processes.

Concrete examples of such architectures are illustrated in (Figure 2) include the use of (a) a fully connected mesh, (b) a broadcast medium, (c) a ring, (d) a mesh of links interconnected with bridges

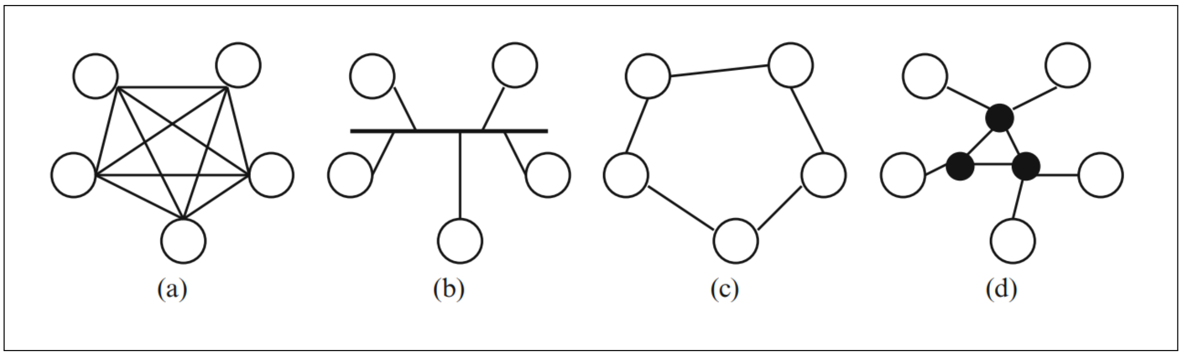


Figure 2: The link abstraction and different instances

1.5.1 Abstracting Link Failures

Here we will introduce the following link abstractions considering processes faults:

- Fair-Loss Links
- Stubborn Links
- Perfect Links
- Logged Perfect Links
- Authenticated Perfect Links

1.5.2 Fair-Loss Links

The **weakest** variant of the link abstraction.

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

Module:

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

Events:

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

Indication: $\langle$ *fll*, *Deliver* | p, m $\rangle$: 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.

Figure 3: Interface of fair-loss point-to-point links

1.5.3 Stubborn Links

The stubborn delivery property causes every message sent over the link to be delivered at the receiver an unbounded number of times.

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.

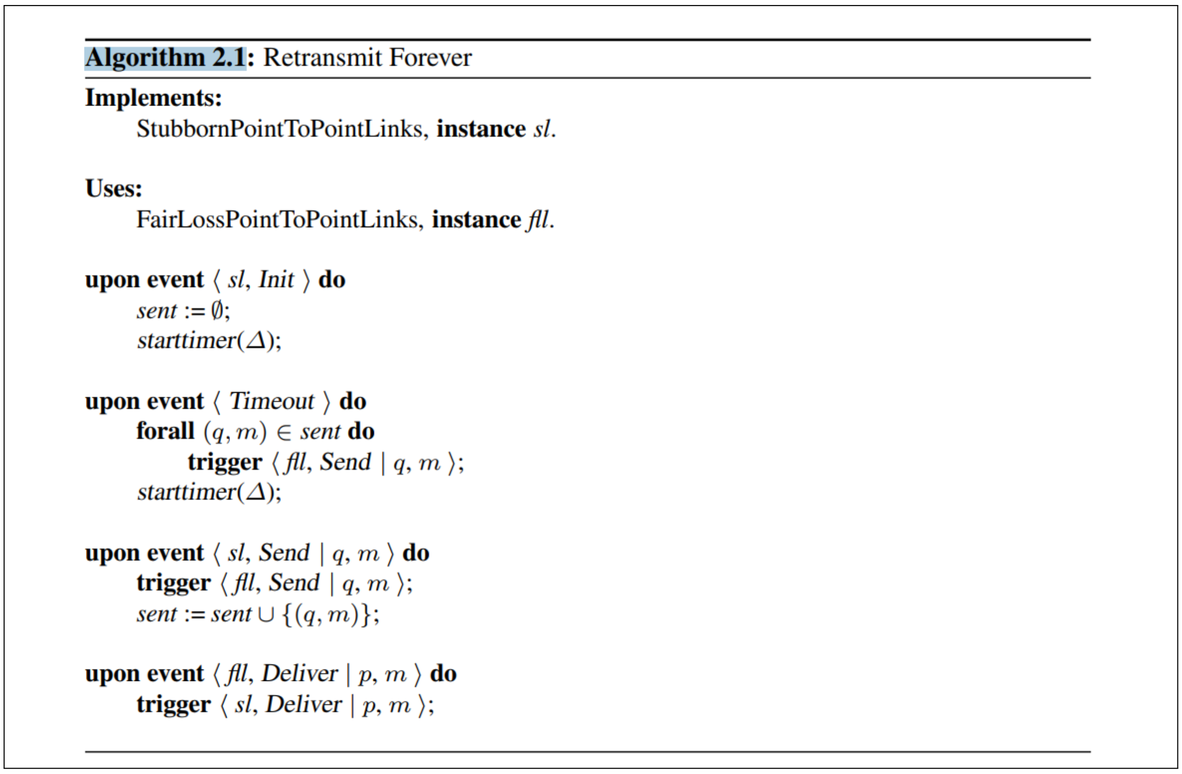
Properties:

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.

Figure 4: Interface of stubborn point-to-point links

Example: Algorithm 2.1 (Retransmit Forever)



```
Algorithm 2.1: Retransmit Forever

Implements:
    StubbornPointToPointLinks, instance sl.

Uses:
    FairLossPointToPointLinks, instance fll.

upon event ⟨ sl, Init ⟩ do
    sent := $\emptyset$;
    starttimer($\Delta$);

upon event ⟨ Timeout ⟩ do
    forall $(q, m) \in$ sent do
        trigger ⟨ fll, Send | $q$, $m$ ⟩;
    starttimer($\Delta$);

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

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

Figure 5: Algorithm 2.1 - Retransmit Forever

The algorithm keeps retransmitting all messages. Here we will consider *Correctness* and *Perfromance*.

Correctness: for an algorithm implementing *StubbornPointToPointLinks* and using *FairLossPointToPointLinks* upon sl-sending a message by a correct process, the message is *fll*-delivered infinitely often by the target process. Because the algorithm keeps *fll*-sending those messages infinitely.

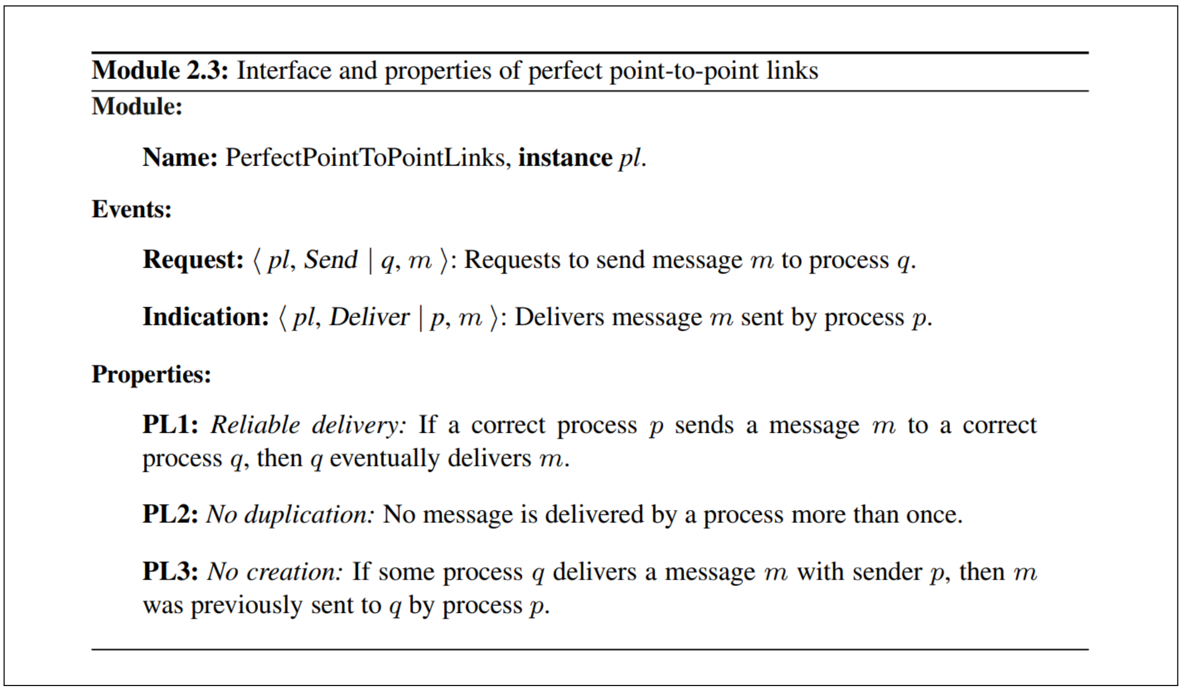
The *no-creation* property is also preserved by the underlying links.

Performance: The algorithm is not efficient. There are two ways to make it better:

- When the algorithm has ended they use of the stubborn links, there is no need to keep on sending the messages.
- Add an acknowledgment mechanism which notifies the sender process to stop sending the messages as they are delivered to the target process

1.5.4 Perfect Links

With the stubborn links abstraction, it is up to the target process to check whether a given message has already been delivered or not. Adding mechanisms for detecting and suppressing message duplicates, in addition to mechanisms for message retransmission, allows us to build an even higher-level primitive: the **perfect links** abstraction, sometimes also called the *reliable links* abstraction.



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

Module:

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

Events:

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

Indication: ⟨ *pl*, *Deliver* | p, m ⟩: Delivers message m sent by process p.

Properties:

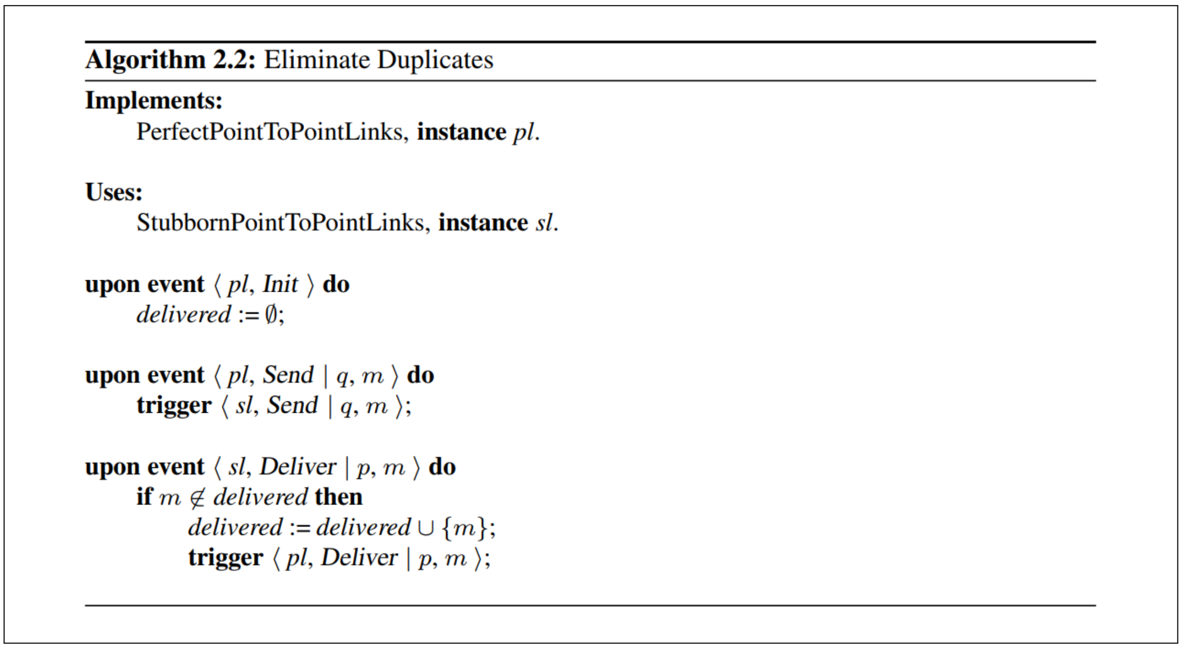
PL1: *Reliable delivery:* If a correct process p sends a message m to a correct process q, then q eventually delivers m.

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.

Figure 6: Interface of perfect point-to-point links

Example: Algorithm 2.2 (Eliminate Duplicates)



Algorithm 2.2: Eliminate Duplicates

```
Implements:
    PerfectPointToPointLinks, instance pl.

Uses:
    StubbornPointToPointLinks, instance sl.

upon event ⟨ pl, Init ⟩ do
    delivered := $\emptyset$;

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

upon event ⟨ sl, Deliver | p, m ⟩ do
    if $m \notin$ delivered then
        delivered := delivered $\cup \{m\}$;
        trigger ⟨ pl, Deliver | p, m ⟩;
```

Figure 7: Algorithm 2.2 - Eliminate Duplicate

Correctness: according to the algorithm, a process p that pl-sends a message m

to target process q. Assuming both these processes are correct and based on the *sl* property, q eventually sl-delivers m, at least once, hence pl-delivers m. So, the reliable delivery is guaranteed. The *no-duplication* property comes from the fact that the messages are being checked if they are delivered before or not and then they are pl-delivered. *No creation* property follows the *no creation* property of the underlying stubborn layer.
Performance: Maintaining physical memory for the ever growing set of messages delivered at every process. Notice that having an periodically acknowledgment and promising not to send those messages anymore is not sufficient as a message can be in transit and uppon delivery, it violates the *no creation* property.

1.5.5 Logged Perfect Links

Suitable for crash-recovery process abstraction.
We log the output in a stable storage, which is accessed through *store* and *retrieve* operations.
Notice that perfect point-to-point links abstraction uses crash-stop processes, where a process may crash only once and a correct process never crashes. But the crash-recovery process abstraction, as used by logged perfect links, may crash a finite number of times and still called *correct* only if it recovers from a crash.

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.

Figure 8: Interface of logged perfect point-to-point links

Example: Algorithm 2.3 (Log Delivered)

It is direct adaptation of *Eliminate Duplicates* that implements perfect point-to-point links from stubborn links. It keeps a record of all the messages that have been log-delivered in the past. It stores this record in stable storage and exposes it also to the upper layer.

```
Algorithm 2.3: Log Delivered
Implements:
    LoggedPerfectPointToPointLinks, instance lpl.

Uses:
    StubbornPointToPointLinks, instance sl.

upon event ⟨ lpl, Init ⟩ do
    delivered := $\emptyset$;
    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 ⟩;
```

Figure 9: Algorithm 2.3 - Log Delivered

Correctness: here is just like *Eliminate Duplicates* algorithm. except that delivering here means logging the messages into a stable storage.
Performance: In terms of messages it is like *Eliminate Duplicates* algorithm. However, here we require one log operation every time a new message is received.

1.5.6 Authenticated Perfect Links

In both fair-loss and stubborn point-to-point links abstractions *no creation* property can not be guaranteed in presence of Byzantine Processes.
Authenticated Perfect Link uses the same interface as the other point-to-point links abstractions. *Authenticity* property is stronger that *no creation* property.

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.

Figure 10: Interface of Authenticated perfect point-to-point links

Example: Algorithm 2.4 (Authenticate and Filter)

Uses a MAC to implement authenticated perfect point-to-point link over a stubborn links abstraction.

Algorithm 2.4: Authenticate and Filter

```
Implements:
    AuthPerfectPointToPointLinks, instance al.

Uses:
    StubbornPointToPointLinks, instance sl.

upon event ⟨ al, Init ⟩ do
    delivered := $\emptyset$;

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($self, p, m, a$) $\wedge$ $m \notin$ delivered then
        delivered := delivered $\cup$ $\{m\}$;
        trigger ⟨ al, Deliver | p, m ⟩;
```

Figure 11: Algorithm 2.4 - Authenticate and Filter

Correctness: The *reliable delivery* and *no duplication* follows the same argument as for algorithm 2.2. For *authenticity* property

Performance: The same issue arises for the set *delivered* that grows without bound. Cryptographic authentication adds a modest computational overhead.

2 Time in Distributed Systems

3 Logical Clock

4 Distributed Mutual Exclusion

5 Broadcast Communications

6 Consensus

7 Ordered Communications

8 Registers

9 Software Replication

10 Overview on Capacity Planning

11 Modeling the Workload of a System

12 Building a Performance Model 1

13 Building a Performance Model 2

14 Dependability Evaluation

15 Intro to Experimental Design

16 CAP Theorem

17 Consistency Criteria for Distributed Shared Memories

18 Publish-Subscribe Communication Paradigm

19 Overlay Networks

20 DLT and Blockchain

21 Exercises

Notice: The exercises are from 2022-2023 academic year.

22 Exams