

# Computer Science Course Notes — 1st Semester

Dario Loi



# Contents

<b>Distributed Systems</b>	<b>1</b>
First Lecture . . . . .	1
Second Lecture . . . . .	1
Definitions . . . . .	1
Consistency . . . . .	2
Run Reconstruction . . . . .	3
Third Lecture . . . . .	6
Vector Clock Properties . . . . .	6
Snapshots . . . . .	7
Fourth Lecture . . . . .	7
Generic Properties of Distributed Systems . . . . .	8
Two-Phase Commit . . . . .	8
Logs . . . . .	8



# Distributed Systems

## First Lecture

DA RECUPERARE !!!

A Space-Time diagram shows the evolution of a distributed system over time, it is composed of a number of timelines, each associated to a process  $p_i$ . Events, occur across this timelines, and are represented by points on the diagram. If an event  $e_1$  is causally related to another event  $e_2$ , then  $e_1$  is to the left of  $e_2$  and there is a line connecting them.

## Second Lecture

### Definitions

We now give a set of formal definitions for very natural concepts in distributed systems.

We define the *history* of a process  $p_i$  as the sequence of all events that occur in the timeline of  $p_i$ .

$$h_i = \langle e_i^1, e_i^2, \dots, e_i^n \rangle$$

We can then have the history of the computation, which is a collection of all the histories of all the processes.

$$H = \langle h_1, h_2, \dots, h_n \rangle$$

A *prefix history* is a sequence of events that occur in the timeline of a process up to a certain point.

$$h_i^k = \langle e_i^1, e_i^2, \dots, e_i^k \rangle$$

A *local state* of process  $i$  after event  $e_i^k$  is the result of the computation up to that point, and is denoted as  $\sigma_i^k$ .

A *global state* is the collection of all local states of all processes at a certain point in time.

$$\sigma^k = \langle \sigma_1^k, \sigma_2^k, \dots, \sigma_n^k \rangle$$

A *run*  $k'$  is a total re-ordering of the events such that the internal order of every process is preserved. This means that a *run* allows the interleaving of the events of different processes, but does not break the order of events in the history of a single process.

A *cut*  $C$  is a collection of the prefixes of every processes' history up to a certain point.

### Consistency

A cut  $C$  is said to be *consistent* iff:

$$e \rightarrow e' \wedge e' \in C \Rightarrow e \in C$$

Or, in other words, if an event is in the cut, then all the events that causally precede it are also in the cut.

By sequentializing the events in a run and extracting a prefix, we can trivially obtain a consistent cut.

### Consistency of Intersection

A possible course exercise is, given two cuts  $C_1$  and  $C_2$ , to prove that their intersection is also a consistent cut.

Given the definition of consistency accommodated for the intersection of two cuts:

$$e \rightarrow e' \wedge e' \in C_1 \cap C_2 \Rightarrow e \in C_1 \cap C_2$$

We know that if  $e'$  is in the intersection, then it is in both  $C_1$  and  $C_2$ . By the definition of consistency, we know that  $e$  must be in both  $C_1$  and  $C_2$ , and therefore in their intersection.

Formally:

$$e \rightarrow e' \wedge e' \in C_1 \rightarrow e \in C_1$$

$$e \rightarrow e' \wedge e' \in C_2 \rightarrow e \in C_2$$

Therefore:

$$e \rightarrow e' \wedge e' \in C_1 \cap C_2 \rightarrow e \in C_1 \wedge e \in C_2 \rightarrow e \in C_1 \cap C_2$$

### Consistency of Union

Given two consistent cuts  $C_1$  and  $C_2$ , we can prove that their union is also a consistent cut.

Given the definition of consistency accomodated for the union of two cuts:

$$e \rightarrow e' \wedge e' \in C_1 \cup C_2 \Rightarrow e \in C_1 \cup C_2$$

Then, formally:

$$e \rightarrow e' \wedge e' \in C_1 \rightarrow e \in C_1$$

$$e \rightarrow e' \wedge e' \in C_2 \rightarrow e \in C_2$$

Therefore:

$$e \rightarrow e' \wedge e' \in C_1 \cup C_2 \rightarrow e \in C_1 \vee e \in C_2 \rightarrow e \in C_1 \cup C_2$$

### Run Reconstruction

Assume that we have a set of processes  $P = \{p_1, p_2, p_3, \dots\}$ , and a monitor process  $p_0$  that is responsible for detecting deadlocks.

Everytime that an event is generated, its responsible process sends a message to the monitor, which then updates its local state.

The monitor process can then reconstruct the run by aggregating the local process states into a global state.

Assuming asynchrony and unbounded message delays, the monitor process does not reconstruct a run, since the order of the individual processes can be inverted.

If we assume a FIFO message delivery, then the monitor process can reconstruct a run. This can be trivially achieved by having a sequence number, in a system such as TCP.

This does *not* recover consistency, since the order of separate processes can still be inverted. However, this level of *local consistency* is sufficient for deadlock detection.

### Simplifying Assumptions for Consistency

Assume that I have:

1. A finite  $\delta_i$  notification delay on each process  $p_i$ .
2. A global clock  $C$  that is synchronized with all processes. We call an idealized version of this clock *Real Clock* (RC).

Then, the monitor process can offline-reconstruct a run by using the global clock to order the events.

Online-reconstruction can be achieved (and is preferred) by exploiting the additional  $\delta_i$  information. Once we receive a notification, we await time  $\max \delta_i$  before committing the event to the global state, in the meantime, we store the event in a buffer that is ordered by the global clock.

### Real Clock Property

The clock property relates the *happens before* relation of messages to the real time of the system.

$$e \rightarrow e' \Rightarrow TS(e) < TS(e')$$

We can obtain a trivial clock by designing a system as such:

1. Each process has associates a sequence number  $k \in \mathbb{N}$  to each event.
2. Each event  $i$ , which receives a message  $m$  from event  $j$ , has a sequence number given by the formula  $\max(k_i, k_j) + 1$ .

This gives us the clock property *by construction*, since each event has a sequence number that is always greater than the sequence number of the event that caused it and of any event that it is casually related to.

This device is called a *Lamport Clock*, or a *Logical Clock*.

### History of an event

Given an event  $e$ , we can define its history as the sequence of events that causally precede it. These are all the events that the monitor process must have received before the event  $e$  is committed to the global state.

$$\Theta(e) = \{e' \mid e' \rightarrow e\}$$

This is a *consistent* cut.



**Opaque detection of consistent cuts**

Given a notification about an event, for example,  $e_5^4$ , the monitor process can ensure that  $e_1^4$  can be committed to global state by checking that for every process, we have received a notification with lamport clock  $\geq 4$ .

We might have some processes that consistently lag behind, and therefore do not reach the lamport clock in a timely manner. We can trivially solve this by sending a liveness check to all processes for which a sufficiently high lamport number has not been received. If the process is alive but not working, it will re-adjust its lamport clock and send a notification.

This will not impact throughput, since the monitor process is not a bottleneck, and the liveness check is a very lightweight operation, that is performed only when necessary.

**Building a Better Clock**

We can build a *better clock*, that is, one with the enhanced property:

$$e \rightarrow e' \iff TS(e) < TS(e')$$

This is called a *Vector Clock*.

It works by communicating the history of the current event in a compact way, we do this by sending a vector of length  $n$  where  $n$  is the number of processes in the system. The vector's entries are updated to be the highest lamport number detected for that relative process, either from a message that has been received, or from the local lamport clock (in the case of the current process).

This allows us to check whether:

$$VC(e) \subseteq VC(e')$$

$$e \rightarrow e' \iff VC(e) \subseteq VC(e')$$

When messages are sent, the vector clock is attached to the message, and the receiving process updates its vector clock by taking the maximum of the two vectors, and summing 1 to the current process.

The monitor process  $p_0$  keeps track of its own vector, where each entry is the highest lamport number that it has received from each process. When it receives a notification, if the vector clock is consistent with the monitor's vector clock, then the event is committed to the global state.

Since vectors are *not* sets, we have to use component-wise comparison to check for causality.

$$VC(e) < VC(e') \iff \forall i \in \{1, 2, \dots, n\} \quad VC(e)_i < VC(e')_i$$

So we are back to the original definition of the happens before relation.

## Third Lecture

We now discuss some more advanced vector clock properties.

### Vector Clock Properties

#### Strong Clock Property

Assume we have two events  $e$  and  $e'$ . Then, we have that:

$$e \rightarrow e' \iff VC(e) \leq^* VC(e')$$

Where  $\leq^*$  is a *partial order* relation, such as:

$$\forall VC(e_i)[k] \leq VC(e_j)[k] \quad \wedge \quad \exists k' VC(e_i)[k'] < VC(e_j)[k']$$

Meaning that all the elements of the vector clock of  $e$  are less than or equal to the corresponding elements of the vector clock of  $e'$ , and at least one element is strictly less, so that we have a *strict* partial order.

#### Retrieving History Cardinality

Given a vector clock  $VC(e)$ , we can retrieve the number of events that causally precede it by taking the sum of all the elements of the vector clock.

$$|\Theta(e)| = \sum_{i=1}^n VC(e)[i]$$

#### Retrieving Causality

Given two events  $e_i$  and  $e_j$ , we can check whether they are causally related by checking the vector clocks.

$$e_i \rightarrow e_j \iff VC(e_i)[i] \leq VC(e_j)[i]$$

#### Existence of a Causal Event

We want to check whether

$$\exists e_k : \neg(e_k \rightarrow e_i) \wedge (e_k \rightarrow e_j)$$

This is true if and only if:

$$\exists k : VC(e_i)[k] < VC(e_j)[k]$$

Meaning that there is a certain event that ticked the clock of  $e_j$  but not the clock of  $e_i$ .

## Snapshots

Now we assume that the processes do *not* send notifications, rather, the monitoring process  $p_0$  polls processes with a snapshot request.

This was already discussed in the first lecture, and we know that what we retrieve cannot retrieve a consistent cut.

We can change the protocol to improve this method. Now  $p_0$  sends a snapshot request to all processes, however, when a process receives a snapshot request, it has to broadcast it to any other process.

Only requests from  $p_0$  are broadcasted, so that we do not flood the network with snapshot requests.

When a process receives a snapshot request, it sends a snapshot of its local state to  $p_0$ .

With this behavior, under the reasonable FIFO-channel assumption, we can reconstruct a consistent cut.

## Proof of Consistency

Assume that we have  $e_i$  and  $e_j$  such that  $e_i \rightarrow e_j$ , and that  $e_i$  is in the snapshot  $C$ , we want to prove that  $e_j$  is also in the snapshot.

By contradiction, assume that  $e_j \notin C$ , then  $p_j$  received the snapshot request before  $e_i$ , but then, since the channels are FIFO, and  $e_j$  was spawned by a received message from  $p_i$ , then the broadcasted snapshot request sent to  $p_j$  reached  $p_j$  before  $e_j$  was spawned, and therefore  $e_j$  is also *not* in the snapshot.

This protocol is known as the *Chandy-Lamport* protocol.

## Fourth Lecture

We want to design a protocol for *atomic commits*, this protocol should have the following properties:

1. If processes reach a decision (commit/abort), it must be the same one.
2. A process cannot reverse its decision after reaching it.
3. The commit decision can be reached only if all processes vote *Yes*.
4. If *there are no failures*, and all processes vote *Yes*, then the commit decision is reached.

Naturally, this could be satisfied by a *trivial* protocol, such as one that *always* aborts, or one that *always* commits. These protocols are not useful since they are not *safe*.

## Generic Properties of Distributed Systems

Usually, there are two sets in which properties can be divided:

1. *Safety* properties, which basically ask “*if* something happens, it does not go *wrong*”.
2. *Liveness* properties, which basically ask “*if* time goes on, *something* will happen”.

You can trivially satisfy safety by doing nothing, you can trivially satisfy liveness by doing anything.

You *want* a tradeoff, such that you get as much liveness as possible while preserving safety.

## Two-Phase Commit

We assume the existence of *two* different processes, *coordinators* and *participants*, we describe the protocol as follows:

1. The coordinator sends a *Prepare* message to all participants.
2. The participants decide Yes/No and send the vote to the coordinator. If a participant votes *No*, then the coordinator must abort.
3. The coordinator receives all votes, if all votes are *Yes*, then the coordinator sends a *Commit* message to all participants, otherwise it sends an *Abort* message.
4. The participants apply the decision.

If the coordinator fails, then the participants will communicate with each other to reach a decision.

If a vote fails to reach the coordinator after a certain timeout period, it will be assumed to be a *No* vote. This preserves the safety property.

This protocol satisfies the properties of atomic commits, and is known as the *Two-Phase Commit* protocol. This is *not* a *live* protocol, since it can deadlock/timeout.

## Logs

Logs are a sequence of events that are written to a file, and are used to recover the state of a system in case of failure. We assume the records to be consistent, so that system failures do not corrupt the log. This means that the log is written to a stable storage, such as a disk.

Logs allow a system to be fault-tolerance since they can be used to recover the state of the system in case of failure.

You can have *two* types of behavior when logging, depending on the order of the operations:

- Log and then act
- Act and then log

In both cases, you might *die* in the middle of the operation, and you might have to recover the state of the system. Both systems are recoverable as a coordinator, but you *must* act in a consistent way to recover the state of the system.

As a participant, recoverability is *not* guaranteed if you act before you log. The best way to ensure recoverability is to log before you act, so that you can be sure about if you voted *Yes* or *No* before you die.

If your vote was *No*, then you can *always* abort. If your vote was *Yes*, then you can re-send the vote to the coordinator, and await the decision.

