Modern Distributed Computing

Theory and Applications

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 7
Tuesday, April 23, 2013

Part 3: Static Asynchronous Networks

1. I/O Automata Model
2. Distributed Data Structures
3. Time, Clocks and Ordering of Events
4. Synchronizers
5. Global Predicates
6. Termination Detection

Global State in Centralized Systems

- The state of a centralized system at a given time instance is given by the state of each active process
  - Let n processes.
  - Each process $\mathcal{P}_u$ at time instance i is at state κ_i^u.
- Describing the state of the system, at a given time instance i, requires storing the state of all processes at time instance i.
- In a centralized system storing the state of all processes can be fairly simple:
  - The process scheduler temporarily de-activates all processes.
  - A copy of the memory is dumped to the storage system.
  - This can take place in a fairly short period of time.

Global State in Distributed Systems

- How can we repeat this process for a distributed system ?
  - Temporary de-activation of the processes cannot take place at the same time.
  - Processes are executed at different computational units.
  - Control messages may arrive at different time instances.
  - How can we synchronize all processes to stop at a predetermined time instance ? (clock synchronization is not trivial and takes time)
- In real systems it is not reasonable to expect that all the processes of the system will be de-activated for a "visible" period of time – e.g., due to efficiency, or just because it is not acceptable.

Applications

Storing the global state of a distributed systems has numerous applications:

1. Assist in debugging the system, e.g., by checking for violations of desired invariants.
2. Produce backup versions of the global state for recovering purposes.
3. Detect if the execution has terminated.
4. Detect whether some of the processes of the system are involved in a "deadlock", that is, a situation in which several processes are all waiting for each other to do something.
5. Compute some global quantity (e.g., the total amount of money available in a set of accounts) being managed by the system.

Definitions – Ordering of Events

- We define as event σ^u an action ϵ that forces $\mathcal{P}_u$ to change its internal state.
- We assume that processes evolve strictly sequentially.
- We assume that sending or receiving a message is an event in a process.

Happened Before ($\rightsquigarrow$)

We say that an event a happened before b ($a \rightsquigarrow b$) if the following conditions are satisfied:

1. If a and b are events in the same process, and a comes before b.
2. If a is the sending of a message by one process and b is the receipt of the same message by another process.

Properties of Happened Before relation

- If $\sigma_i \rightsquigarrow \sigma_j$ and $\sigma_j \rightsquigarrow \sigma_k$, then $\sigma_i \rightsquigarrow \sigma_k$.
- The happened before relation defines only a partial ordering of the set of events that are observed during the execution of the system.
- Some events cannot be related by the happened before relation.
- Such events are called "concurrent":

$$\sigma_i^u \parallel \sigma_j^v \quad \Leftrightarrow \quad \left(\sigma_i^u \not\rightsquigarrow \sigma_j^v\right) \wedge \left(\sigma_j^v \not\rightsquigarrow \sigma_i^u\right)$$

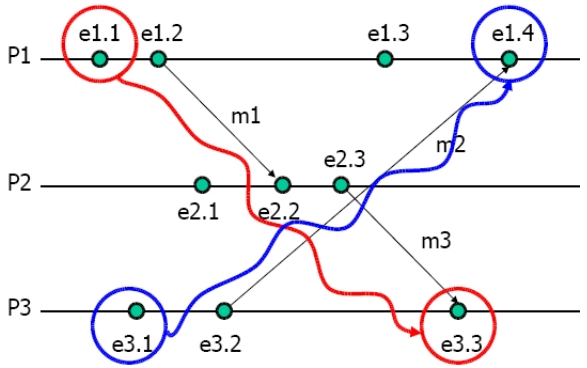
- Events that take place in the same process are related to each other.
- Based on the above definition, if two events are concurrent then it is not necessary that they take place at the same time instance.

Examples of Happened Before relation

Execution Example

It holds that

- $e_{1.1} \rightsquigarrow e_{3.3}$ given that $e_{1.1} \rightsquigarrow e_{1.2} \rightsquigarrow e_{2.2} \rightsquigarrow e_{2.3} \rightsquigarrow e_{3.3}$
- $e_{3.1} \rightsquigarrow e_{1.4}$ given that $e_{3.1} \rightsquigarrow e_{3.2} \rightsquigarrow e_{1.4}$



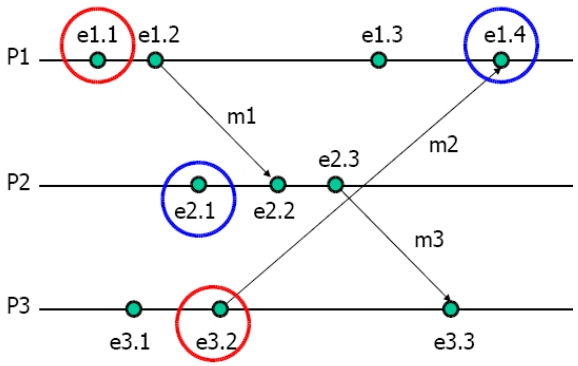
Examples of Happened Before relation

Execution Example

It holds that

- $e_{1.1} \parallel e_{3.2}$
- $e_{2.1} \parallel e_{1.4}$

Based on the example, event $e_{1.1}$ takes place before $e_{3.2}$ and $e_{2.1}$ before $e_{1.4}$



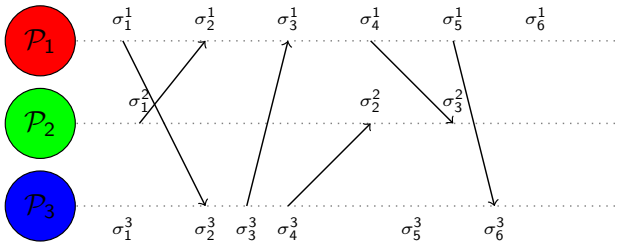
Definitions – Ordering of Events

- There is a strict ordering of events $\sigma_1^u, \sigma_2^u, \ldots$ for each process $\mathcal{P}_u$ such that $\sigma_k^u, \sigma_{k+1}^u$, that is σ_k^u happened before σ_{k+1}^u.

Local History

The local history of process $\mathcal{P}_u$ is noted as h_u and defines a sequence of events that take place in the process, e.g., $h_u = \sigma_1^u, \sigma_2^u, \sigma_3^u, \sigma_4^u$.

Example Execution – Send/Receive diagram



- Local History – $h_1 = \sigma_1^1, \sigma_2^1, \sigma_3^1, \sigma_4^1, \sigma_5^1, \sigma_6^1$
- Local History – $h_2 = \sigma_1^2, \sigma_2^2, \sigma_3^2$
- Local History – $h_3 = \sigma_1^3, \sigma_2^3, \sigma_3^3, \sigma_4^3, \sigma_5^3, \sigma_6^3$
- The local history of each process is temporally fully ordered.

Definitions – Global History / Global State

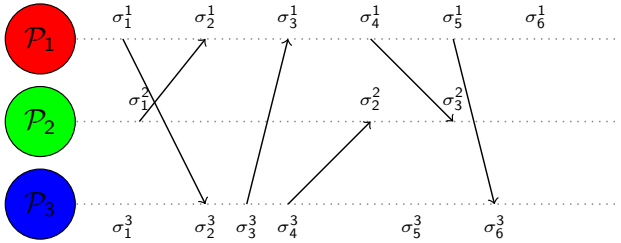
Global History

The global history $\mathcal{H}$ of a distributed system is defined as the union of the local histories of all the process participating in it, i.e., $\mathcal{H} = h_1 \cup \ldots \cup h_n$.

Global State

The global state of a distributed system defined as Σ is the union of the local states of all the processes participating in it, i.e., $\Sigma = \{\kappa^1, \ldots \kappa^n\}$.

Execution Example – Send/Receive diagram



- Global history – $\mathcal{H} = h_1 \cup h_2 \cup h_3$
- The set is only partially ordered.
- Global state – $\Sigma_1 = \{\kappa_2^1, \kappa_1^2, \kappa_2^3\}$
- Global state – $\Sigma_2 = \{\kappa_3^1, \kappa_2^2, \kappa_4^3\}$

Definitions – Cut / Cut Frontier

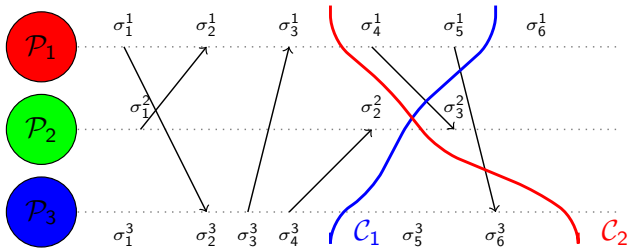
Cut

A cut $\mathcal{C}$ of a distributed system is a subset of the global history $\mathcal{H}$ that consists of events $\sigma^u \geq 0$ of initial events from each process $\mathcal{P}_u$, e.g., $\mathcal{C} = h_1^{\sigma^1} \cup \ldots \cup h_n^{\sigma^n}$. Thus a cut is defined via a vector $\{\sigma^1, \ldots \sigma^n\}$.

Cut Frontier

The set of last events $\{\max(\sigma^1), \ldots, \max(\sigma^n)\}$ that are included in athe cut $\mathcal{C}$ are called the cut frontier.

Execution Example – Send/Receive diagram



- $\mathcal{C}_1 = h_1^{\sigma_5^1} \cup h_2^{\sigma_2^2} \cup h_3^{\sigma_4^3} = \{\sigma_1^1, \ldots, \sigma_5^1, \sigma_1^2, \sigma_2^2, \sigma_1^3, \ldots, \sigma_4^3\}$
- Frontier cut of $\mathcal{C}_1$ is $\{\sigma_5^1, \sigma_2^2, \sigma_4^3\}$
- $\mathcal{C}_2 = h_1^{\sigma_3^1} \cup h_2^{\sigma_2^2} \cup h_3^{\sigma_6^3} = \{\sigma_1^1, \ldots, \sigma_3^1, \sigma_1^2, \sigma_2^2, \sigma_1^3, \ldots, \sigma_6^3\}$
- Frontier cut of $\mathcal{C}_2$ is $\{\sigma_3^1, \sigma_2^2, \sigma_6^3\}$

Discussion

- Every cut of a distributed system corresponds to a global state.
- Only specific cuts correspond to global states that may occur during an execution of the system.
- During the execution of the previous example, the cut $\mathcal{C}_1$ represents a "possible" global state.
- While cut $\mathcal{C}_2$ is "impossible" to occur since process $\mathcal{P}_3$ appears to be receiving a message from $\mathcal{P}_1$, that $\mathcal{P}_1$ has not yet transmitted based on the events defined by the cut.

Definition – Consistent Cut

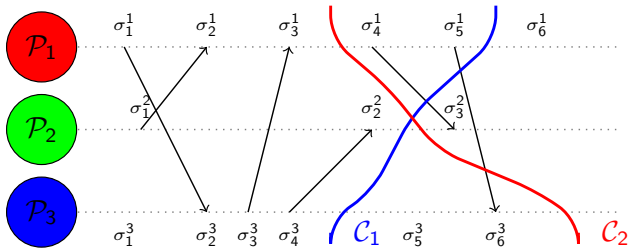
Consistent Cut

A cut $\mathcal{C}$ if for all pair of events σ and σ' it holds that

$$\sigma \in \mathcal{C} \wedge \left(\sigma' \rightsquigarrow \sigma\right) \Rightarrow \sigma' \in \mathcal{C}$$

- A global state is consistent if it corresponds to a consistent cut.
- The consistent global states are those that can appear in an actual execution of the distributed system.

Example Execution – Send/Receive diagram



Rule of thumb for identifying a consistent cut:

- All the Send/Receive arrows that "intersect" the cut must start from the left part of the cut and end up on the right part of the cut.

$\mathcal{C}_1$ is consistent – arrows $\sigma_4^1 \rightarrow \sigma_3^2$, $\sigma_5^1 \rightarrow \sigma_6^3$ intersect the cut from left to right.

$\mathcal{C}_2$ is not – arrow $\sigma_5^1 \rightarrow \sigma_6^3$ intersects the cut from right to left.

Introduction to Logical Clocks

- We begin with an abstract point of view.
- A clock is just a way of assigning a number to an event, where the number is thought of as the time at which the event occurred.
- We define a clock C_i for each process $\mathcal{P}_i$ to be a function which assigns a number (timestamp) $TS_i(a)$ to any event a in that process.
- The entire system of clocks is represented by the function C which assigns to any event b the number (timestamp) $TS(b)$ where $TS(b) = TS_i(b)$ if b is an event in process $\mathcal{P}_j$.
- Let's not make any assumption about the relation of the numbers $TS_i(a)$ to physical time.

Discussion on Logical Clocks

- We can think of the clocks C_i as logical rather than physical clocks.
- They may be implemented by counters with no actual timing mechanism.
- In order to understand what it means for such a system of clocks to be correct, we cannot base our definition of correctness on physical time.
- That requires introducing clocks which keep physical time.
- Our definition must be based on the order in which events occur.
- The strongest reasonable condition is that if an event a occurs before another event b, then a should happen at an earlier time than b.

Logical Clock Condition

- For each event σ we assign a timestamp $TS(\sigma) \in \mathcal{T}$, where $\mathcal{T}$ is a fully ordered set.
- We wish that the timestamp of σ_i where $\sigma_i \rightsquigarrow \sigma_j$ must have a smaller timestamp from σ_j that is $TS(\sigma_i) < TS(\sigma_j)$
- If we can implement such a clock then we can serialize all events without violating their logical relation.
- Such a serialization of the events dues not necessarily depict the actual ordering of events.
- However such a serialization avoids the necessity to synchronize physical clocks:
  - reduces time complexity,
  - reduces message complexity,
  - avoid periodic execution of the synchronization mechanism.

Lamport Logical Clock

- Lamport defines a simple mechanism that satisfies the logical clock condition.
- It named this mechanism the "logical clock".
- A logical clock is a monotonously strictly increasing counter, whose value does not need to be related with a physical clock.
- Each process $\mathcal{P}_u$ maintain its own internal logical clock LC_u.
- We use the values provided by the logical clock to timestamp each event observed by the process.

LamportTime Algorithm

Each process maintains a counter LC initially set to 0. For each internal event or a `send(m)` event they set $LC++$. For each message m sent, they include the value of LC. For each message m is received with timestamp $TS(m)$ they set $LC = \max\{LC, TS(m)\} + 1$.

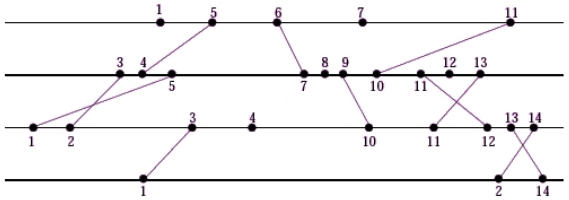
- If two events σ_i and σ_j it holds that $\sigma_i \rightsquigarrow \sigma_j$ the algorithm LamportTime guarantees that $TS(\sigma_i) < TS(\sigma_j)$.
- If $TS(\sigma_i) < TS(\sigma_j)$ it does not necessarily hold that $\sigma_i \rightsquigarrow \sigma_j$.

Lamport's Algorithm

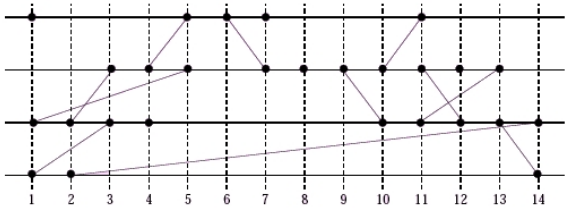
- It is possible that two events σ_i and σ_j that take place in two different processes are not related and yet they have the same timestamp.
- Yet, we do not wish two events to have identical timestamps.
- A simple solution is to use the unique identifiers of the processes when generating the timestamps.
  - we assume that processes have unique identifiers.
- Therefore the timestamps of σ_i^u and σ_i^v will look like $i.u$ and $i.v$ – e.g., 11.15 and 11.306.

Execution Example

Actual Execution



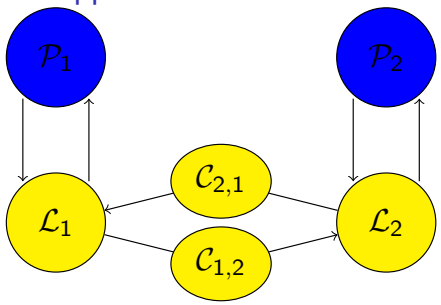
Execution Transformation



Implementing Lamport's Algorithm

1. Given an automaton $\mathcal{P}$ we can integrate Lamport's algorithm within $\mathcal{P}$.

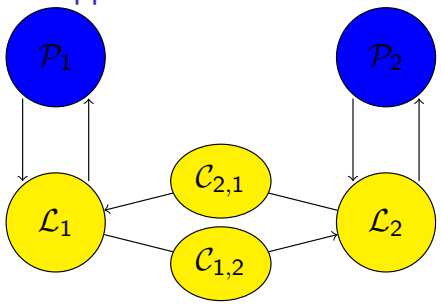
1^{st} Approach



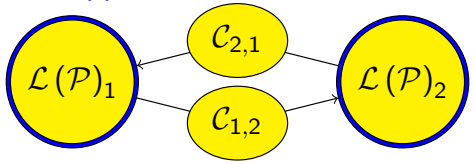
Implementing Lamport's Algorithm

1. Given an automaton $\mathcal{P}$ we can integrate Lamport's algorithm within $\mathcal{P}$.
2. We "extend" $\mathcal{P}$ by implementing Lamport's algorithm as LamportTime($\mathcal{P}$).

1^{st} Approach



2^{nd} Approach



Automaton LamportTime$_u$

Actions:

- Input actions in(LamportTime$_u$)
  1. $\texttt{send}(m)_u$ – where m a message
  2. $advanceClock_u$
- Output actions out(LamportTime$_u$)
  1. $\texttt{receive}(m)_u$ – where m a message

State:

- $\texttt{clock}_u$ – a logical clock, initially set to 0

Automaton LamportTime$_u$

Transitions:

- `send` $(m)_u$
  - effect:
    `clock++`
    `send(m, clock)`
- `advanceClock`$_u$
  - effect:
    `clock++`
- `receive` $(m)_u$
  - precondition:
    `receive(m, c)`
  - effect:
    `clock = max(clock, c) + 1`

Automaton LamportTime($\mathcal{A}$)$_u$

Actions:

- Same with $\mathcal{A}$
- We replace `send` $(m)_u$ of $\mathcal{A}$ with `send` $(m, c)_u$
- We replace `receive` $(m)_u$ of $\mathcal{A}$ with `receive` $(m, c)_u$

State:

- Same with $\mathcal{A}$
- `clock`$_u$ – a logical clock, initially set to 0

Transitions:

- Input/output/internal actions apart from *send*, *receive*
  - precondition:
    Same with $\mathcal{A}$
  - effect:
    `clock++`

Automaton LamportTime($\mathcal{A}$)$_u$

Transitions: (continued)

- `send` $(m, c)_u$
  - precondition:
    Same with *send*$(m)_u$ of $\mathcal{A}$
    `c = clock + 1`
  - effect:
    Same with *send*$(m)_u$ of $\mathcal{A}$
    `clock = c`
- `receive` $(m, c)_u$
  - effect:
    Same with *receive*$(m)_u$ of $\mathcal{A}$
    `clock = max(clock, c) + 1`

- Processes share a common (critical) resource.
- Access to this resource requires exclusive access from only one process.
- The part of the process that handles the resource exclusively is called the "critical section" (CS).
- We need to coordinate the actions of the processes.
- In centralized systems, various primitives are available such as
  - semaphores, locks, monitors ...
- The problem of mutual exclusion was introduced by Edsger Dijkstra in 1965.

Minimum Requirements

- Safety – only and only one process may access the critical resource at any given time instance.
- Liveness –
  - If a process wishes to enter the critical section then it will eventually succeed.
  - If the common resource is not used, then any process requesting access will be granted access within a finite period of time.
- Ordering – access to enter the critical section will be givenn according to the happened-before relation: the requests are served based on the order that they where issued.

Assumptions

1. Processes are assigned unique identifiers.
2. Each process a critical section.
3. Processes compete for 1 critical resource.
4. No global clock is available.
5. Processes communicate using messages.
6. Communication channels are reliable, FIFO.
7. The network is fully connected.

Performance Measures

1. Correctness – the conditions of safety, liveness, ordering are preserved.
2. Communication Complexity – processing of requests to enter critical section minimize total number of message exchanges.
3. Latency – time elapsed between the issue of a request and the access of the resource is minimized.

Coordinator Algorithm

During an initialization phase, processes elect a leader – the coordinator process $\mathcal{P}_c$. A process that wants to enter its CS sends a request message to $\mathcal{P}_c$. $\mathcal{P}_c$ adds the request at the tail of a queue. If the critical resource is free, and the queue is not empty, $\mathcal{P}_c$ informs the process whose request is a the head of the queue. When the process exists its CS it informs $\mathcal{P}_c$ that the critical resource is free.

- Inspired by centralized systems.
- Solves the problem: satisfies all 3 conditions.
- Easy solution – only 3 types of messages required: `request`, `reply`, `release`

Properties of Coordinator Algorithm

- For a process to enter the CS only 2 messages are required – the latency is related to the time of transmitting these two messages or the roundtrip delay.
- Low scalability – the coordinator is a point of congestion: a single process is servicing the whole network.
- Poor robustness – the coordinator is a single point of failure.
  - In term of failure, the processes need to elect a new coordinator,
  - The new coordinator needs to reconstruct the queue of requests, guaranteeing the ordering condition.

LamportME Algorithm

All processes maintain a local logical clock LamportTime and a queue for incoming requests. A process that wishes to enter its CS it sends a request (with a current timestamp) to all other processes and adds its request to the tail of the queue. When a process receives a request, it adds the request to the tail of its local queue and confirms the request. The process whose request has the smallest timestamp enters its CS. When the process exists its CS it notifies all other processes and removes its request from the head of the queue – similarly all other processes remove the request from the head of their queue by receiving the release message.

- First fully distributed solution proposed in 1978.
- Solves the problem: all conditions are guaranteed.

Properties of LamportME

- Uses 3 types of messages: `request`, `reply`, `release`
- For each request a total of $3\,(n-1)$ messages are exchanged:
  - $n-1$ request messages,
  - $n-1$ reply messages,
  - $n-1$ release messages.
- The latency of the algorithm for one request is $2\delta + \mathcal{O}(I)$
- The delay from the point when a process requires to enter the CS is relevant to the time needed to exchange $2\,(n-1)$ messages.

Correctness of LamportME

Lemma (LamportME.1)

LamportME guarantees the safety condition.

Proof: By contradiction.

- Let an execution of the system where two processes u and v are within CS at the same time.
- Let the message $\texttt{request}_u$, sent by u at logical time t_u and the message $\texttt{request}_v$ sent by v at logical time t_v.
- Let assume that $t_u < t_v$.
- Then for v to enter CS, the queue of v must include some message from u with timestamp greater than t_v and thus greater then t_u.

Correctness of LamportME

- Since communication channels are FIFO, for this to happen we need v to receive the message $\texttt{request}_u$ while executing its CS.
- However, to do so, v must have received $\texttt{release}_u$ before entering CS.
- Thus u must have already existed its CS at the time when v was executing its CS.
- YEt, we assumed that u and v are executing their CS at the same time instance.

■

Correctness of LamportME

Lemma (LamportME.2)

LamportME guarantees the liveness condition.

Proof: The property of liveness is guaranteed due to the usage of logical clocks, the processing of requests based on the timestamps assigned by the logical clocks to the the `request` messages.

- We need to show that the process that sent the `request` message with the smallest timestamp will the one to enter first its CS.
- Based on the assumption that channels are reliable and FIFO.
- Since all `request` messages with timestamp smaller than a given event are finite, by induction we can show that all requests will be served.

■

Correctness of LamportME

Lemma (LamportME.3)

LamportME guarantees the ordering condition.

Proof: The property of ordering is guaranteed due to the usage of logical clocks, the processing of requests based on the timestamps assigned by the logical clocks to the the `request` messages and the assumption that communication channels are FIFO.

■

Theorem (LamportME.4)

LamportME solves the problem of mutual exclusion.

Proof: Due to the LamportME.1, LamportME.2, LamportME.3.

■