Modern Distributed Computing

Theory and Applications

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 8
Tuesday, April 30, 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

Introduction

- A process wishes to identify the global state of the distributed system.
  - We call this process the monitor
- It has to "collect" the local states of all the processes of the system.
- Due to the time free property of asynchronous computation, reconstructing the global state is a non-trivial task.
- Fundamental problem.

Passive Construction of Global Snapshots

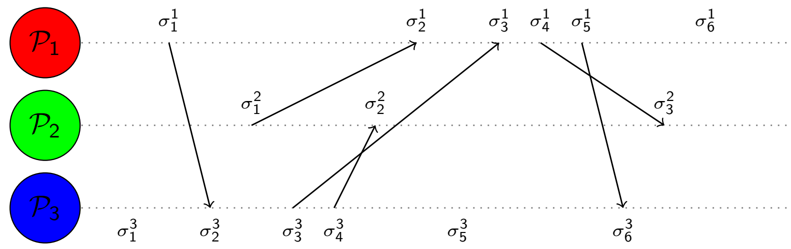
- Let process $\mathcal{P}_0$ the monitor that wishes to construct the global snapshot.
  - No messages are sent by the monitor – it passively collects info about the system.
- Whenever one of the other processes changes its state, it informs $\mathcal{P}_0$ by sending a special message.
- $\mathcal{P}_0$ constructs an observation of the run of the system by keeping track of the special messages.
- The observation is based on the sequence of events, as received by $\mathcal{P}_0$.

Properties of Observations

- Due to the uncertainty in the delivery of messages, two different monitoring processes may construct different observations for the same run.
- An observation may not reflect an actual run.

Execution Example

$$\mathcal{R} = \{\sigma_1^3, \sigma_1^1, \sigma_2^3, \sigma_1^2, \sigma_3^3, \sigma_4^3, \sigma_2^2, \sigma_2^1, \sigma_5^3, \sigma_3^1, \sigma_4^1, \sigma_5^1, \sigma_6^3, \sigma_3^2, \sigma_6^1\}$$

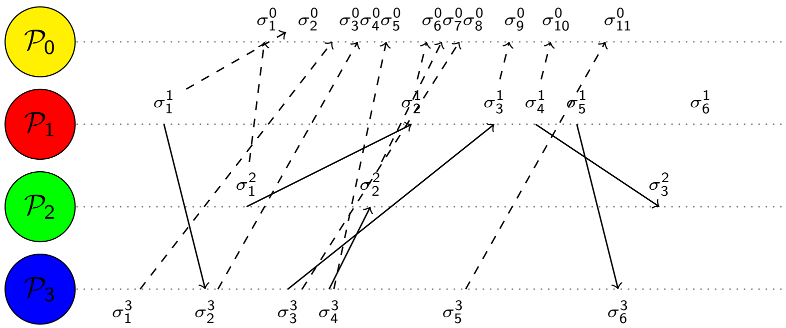


We identify the following observations of $\mathcal{R}$:

- $\mathcal{O}_1 = \{\sigma_1^2, \sigma_1^1, \sigma_1^3, \sigma_2^3, \sigma_4^3, \sigma_2^1, \sigma_2^2, \sigma_3^3, \sigma_3^1, \sigma_4^1, \sigma_5^3, \ldots\}$
- $\mathcal{O}_2 = \{\sigma_1^1, \sigma_1^3, \sigma_1^2, \sigma_2^3, \sigma_2^1, \sigma_3^1, \sigma_3^3, \sigma_4^3, \sigma_2^2, \sigma_5^3, \sigma_6^3, \ldots\}$
- $\mathcal{O}_3 = \{\sigma_1^3, \sigma_1^2, \sigma_1^1, \sigma_2^1, \sigma_2^3, \sigma_3^3, \sigma_3^1, \sigma_4^3, \sigma_4^1, \sigma_2^2, \sigma_5^1, \ldots\}$

Execution Example

$$\mathcal{O}_1 = \{\sigma_1^2, \sigma_1^1, \sigma_1^3, \sigma_2^3, \sigma_4^3, \sigma_2^1, \sigma_2^2, \sigma_3^3, \sigma_3^1, \sigma_4^1, \sigma_5^3, \ldots\}$$



- The observation does not reflect a real run of the system.
- The ordering of the events of $\mathcal{P}_3$ violates their ordering in the local history of the process.
- Event σ_4^3 appears before σ_3^3.

Passive Construction using Physical Clocks

- We assume that processes have access to a physical clock.
- We assume that the physical clocks are synchronized.
- We assume that the processes are aware of an upper bound $\mu \geq l + d$.
- The monitor, at time instance t records all messages received with timestamps up to $t - \mu$, in ascending timestamp order.
- The observations of the monitor may be used to construct global snapshots.
- This simple algorithm is based on the design of the Synchronizer by Tel and Leeuwen.

Synchronization of Physical Clocks

- Synchronizing physical clocks in a distributed system is not a trivial task.
- The construction of global snapshots may be violated even in weaker conditions
  - All we need is to guarantee the chronological ordering of the events!
- We can replace the physical clocks by logical clocks.
- We use the LamportTime algorithm for defining timestamps using the "happened-before" relation.

Passive Construction using Logical Clocks

- We assume that each process has access to a logical clock.
  - They execute the LaportTime algorithm.
- The monitor process, at time t records all messages received in increasing timestamp order.
- The observations of monitor may be used to construct global snapshots.
- for example $\mathcal{O}_4 = \{\sigma_1^1, \sigma_1^2, \sigma_1^3, \sigma_2^1, \sigma_2^3, \sigma_3^3, \sigma_3^1, \sigma_4^3, \sigma_4^1, \sigma_2^2, \sigma_5^3, \ldots\}$
  - is a consistent snapshot.

Passive Construction using Logical Clocks

- This algorithm needs a final modification to be correct.
- We may receive a message regarding event σ'' after receiving a message about event σ' while $LC(\sigma'') < LC(\sigma')$
- This is because two logical clocks cannot detect the gap that may exist between their local counters.

Gap Detection

Given two events σ and σ' with timestamps $LC(\sigma)$ and $LC(\sigma')$ for which $LC(\sigma) < LC(\sigma')$, decide if another event σ'' exists such that $LC(\sigma) < LC(\sigma'') < LC(\sigma')$

Passive Construction using Logical Clocks

- We assume that the channels are FIFO.
- Then if $\mathcal{P}_0$ receives a message m from $\mathcal{P}_u$ with timestamp $LC(m)$ it can assume that no other message m' can be received from $\mathcal{P}_u$ with timestamp $LC(m') < LC(m)$
- This is true since the logical clocks may not detect a gap between the timestamps of different processes
  - The message m is consistent.
- Thus, the monitor, at time t it notes down all the consistent messages that it has received using an increasing order.

Properties of Algorithm

- This algorithm is known as the *LogicalTimeSnapshot* algorithm.
- Note that the physical clocks are also unable to detect a possible gap.
- However, due to the assumption that the processes know an upper bound $\mu \geq l + d$ we can come up with an equivalent assumption:
  - At time t, all messages with timestamps smaller than $t - \mu$ are consistent.

Snapshots

- The two algorithms assume a passive process that takes up the role of the monitor.
- All the other processes are constantly updating $\mathcal{P}_0$.
- We wish to construct snapshots on demand.
- Thus $\mathcal{P}_0$ wishes to "look" the other processes of the system and record a "consistent" global snapshot.
- The snapshot is said to be consistent if it looks to the processes as if it were taken at the same instant everywhere in the system.

Definitions

Channel State

The state of a channel $\mathcal{C}_{uv}$ connecting $\mathcal{P}_u$ with $\mathcal{P}_v$, includes all the messages sent by $\mathcal{P}_u$ to $\mathcal{P}_v$, that have not been received by $\mathcal{P}_v$.

- We denote by $nbrs_u^{in} = \{v | (v, u) \in E\}$ all the incoming neighbors of u.
- We denote by $nbrs_u^{out} = \{v | (u, v) \in E\}$ all the outgoing neighbors of u.

Consistent Global Snapshots with Physical Clocks

- We assume that processes have access to a physical clock.
- We assume that the physical clocks are synchronized.
- We assume that the processes are aware of an upper bound $\mu \geq l + d$.
- The algorithm assumes that all processes record their state at the same physical time instance.
- The monitor, selects a suitable time instance t_*, such that it can guarantee that a message currently in transit will be received by all the processes of the system before t_*.

Consistent Global Snapshots with Physical Clocks

- Initially, $\mathcal{P}_0$ transmits the message *TakeSnapshot*(t_*) to all other processes.
- At time t_* each process $\mathcal{P}_u$
  1. Records its local state σ_u,
  2. Transmits a marker message to all $nbrs_u^{out}$,
  3. Sets each state *state*(C_{vu}) to an empty state,
  4. Records all messages received from $nbrs_u^{in}$.
- When $\mathcal{P}_u$ receives from $\mathcal{P}_v$ a message with *timestamp*$(m) > t_*$
  1. Stops the recording of incoming messages from $\mathcal{P}_v$,
  2. Transmits to $\mathcal{P}_0$ the *state*(C_{vu}).

Discussion

- For each $\mathcal{P}_v \in nbrs_u^{in}$ the state of C_{vu} includes
  - The set of messages sent by $\mathcal{P}_v$ before time t_* that were received by $\mathcal{P}_u$ after time t_*.
  - That is, all messages that at time t_* where in transit.
- The marker messages guarantee that $\mathcal{P}_u$ will eventually receive a message m for which *timestamp*$(m) \geq t_*$

Discussion

- Let an event σ belong to the cut $\mathcal{C}_*$, that is related to the constructed global state, then *timestamp*$(\sigma) < t_*$
- Thus,

$$(\sigma \in \mathcal{C}_*) \wedge (\text{timestamp}(\sigma') < \text{timestamp}(\sigma)) \Rightarrow \sigma' \in \mathcal{C}_*$$

- Since the physical clocks guarantee the clock property, the above relation suffices to prove that the cut $\mathcal{C}_*$ is consistent ant thus the global state is consistent.

Consistent Global Snapshots with Logical Clocks

- Since logical clocks also guarantee the clock property, we can replace the physical clocks with logical.
- However how can we define a time instance t_* using logical clocks?
- Also, in the previous algorithm we assumed that $\mathcal{P}_0$ can somehow select such a time instance t_*.
- We now assume that $\mathcal{P}_0$ may compute a logical time instance ω_*, big enough, such that no logical clock can reach this value
  - Weaker assumption.

Consistent Global Snapshots with Logical Clocks

- Initially, process $\mathcal{P}_0$ transmits the message *TakeSnapshot*(ω_*) to all the other processes and sets its logical timestamp to ω_*.
- At time instance ω_* each process $\mathcal{P}_u$
  1. Records its local state σ_u,
  2. Sends a marker message to all $nbrs_u^{out}$,
  3. Starts recording the messages received from $nbrs_u^{in}$.
- When $\mathcal{P}_u$ receives a message from $\mathcal{P}_v$ with *timestamp*(m)$\geq \omega_*$
  1. Stops recording messages received from $\mathcal{P}_v$,
  2. Notifies $\mathcal{P}_0$ of *state*(C_{vu}).

Chandly and Lamport algorithm

- Chandy and Lamport observe that the monitor process does not participate in the computation between the time instance that the message *TakeSnapshot*(ω_*) is transmitted and until a marker message is received by another process.
- Thus the logical clock is forced to take the value ω_*.
- We can replace the message *TakeSnapshot*(ω_*) by a simple message *TakeSnapshot*
  - the process records its local history upon receiving the message *TakeSnapshot*.
- Based on this observation, Chandy and Lamport propose an algorithm that integrates the idea of logical clocks.

Chandly and Lamport algorithm

- Initially, process $\mathcal{P}_0$ sends the message *TakeSnapshot* to itself.
- When a process $\mathcal{P}_u$ receives the message *TakeSnapshot* from process $\mathcal{P}_\pi$ for the first time
  1. Records its local state σ_u,
  2. Transmit a message *TakeSnapshot* to all $nbrs_u^{out}$,
  3. Sets the set *state*($C_{\pi u}$) to an empty set.
  4. Records all messages received from $nbrs_u^{in}$ except from $\mathcal{P}_\pi$.
- When $\mathcal{P}_u$ receives a second *TakeSnapshot* message from $\mathcal{P}_\delta$
  1. Stops recording messages received from $\mathcal{P}_\delta$.
  2. Notifies $\mathcal{P}_0$ of the *state*($C_{\delta u}$).

Discussion

- The message *TakeSnapshot* is transmitted to all outgoing channels of each process, as soon as the process receives the message for the first time.
  - If the system is strongly connected, then it is guaranteed that the message *TakeSnapshot* will traverse each channel exactly once.
- When a process receives the message *TakeSnapshot* from all its incoming channels, the contribution of the process to the construction of the global state is complete.
  - The process terminates.

Correctness of Chandy and Lamport algorithm

Theorem (ChandyLamportSnapshot.1)

The ChandyLamportSnapshot algorithm records a consistent snapshot for application A.

Proof: Let α an execution of the higher application A.

- Let's assume that during the execution of A, at state Σ_ϵ the *ChandyLamportSnapshot* is activated, that terminates at state Σ_τ and records state Σ_*.
- Let α_1 the part of α before state Σ_ϵ.
- Let α_2 the part of α after state Σ_τ.

Correctness of Chandy and Lamport algorithm

- The global snapshot Σ_* is consistent if the exists an execution α' such that
  - no process can distinguish α from α',
  - execution α' starts with α_1 and concludes with α_2,
  - States $\Sigma_\epsilon, \Sigma_*, \Sigma_\tau$ appear with the same order in α'.
- Our goal is to re-order the events of α in a way such that we end up with an execution α' in which $\Sigma_\epsilon, \Sigma_*, \Sigma_\tau$ appear with the same order.
- Essentially we re-arrange logically independent events.

Correctness of Chandy and Lamport algorithm

- Let σ_k and σ_{k+1} to consecutive events in α that take place in processes $\mathcal{P}_u$ and $\mathcal{P}_v$ and are after and before the *TakeSnapshot* (respectively).
- Thus, σ_k cannot be the transmission of a message m and σ_{k+1} the reception of m.
  - When $\mathcal{P}_u$ recorded it state, it transmitted the message *TakeSnapshot* to $\mathcal{P}_v$.
  - Since channels are FIFO the message reached $\mathcal{P}_v$ before m as σ_{k+1} happens after the recording, which is a contradiction.
- Moreover, the state of $\mathcal{P}_v$ after σ_{k+1} is not affected by σ_k as it takes place in another process.
- Also, the state of $\mathcal{P}_u$ after σ_k is not affected by σ_{k+1}.
- Thus, we can re-order σ_k and σ_{k+1}.

Correctness of Chandy and Lamport algorithm

- We continue such re-orderings until we end up with α' where all events before the *TakeSnapshot* precede the events after the markers.
- Then α' starts with α_1 and ends up with α_2.
- Σ_* appears in α' immediately before α_2.
- All re-orderings are related to events after Σ_ϵ and before Σ_τ.
- Σ_* is the state of the network after the last event recorded before the *TakeSnapshot* in execution α' and before the first event after the markers.
- In this way we end up with execution α' where no process can distinguish α from α'.

$\blacksquare$

Properties of the Chandy and Lamport algorithm

- The algorithm is correct – it constructs consistent global snapshots.
- The communication complexity is $\mathcal{O}(|E|)$.
- The time complexity is not easy to compute since the higher level application is executed in parallel.
- If we ignore possible delays that may arise due to delays in the delivery of the messages transmitted by the higher level application, the *ChandyLamportSnapshot* algorithm terminates within $\mathcal{O}(\delta(l + d))$ time.

Stable Property Detection

- In many fundamental problems of distributed computing, e.g.,
  - Deadlock detection
  - Termination detection
  - Debugging
  - Resource sharing
  - Garbage collection
  - Token detection
- We need to evaluate a global property
  - We construct a global state,
  - We evaluate the global predicate for this state.

Properties of the Problem

- Recording the global state
  - Actively or Passively,
  - Requires message exchanges,
  - The system may encounter failures.
- The state may not be consistent.
- A global state (or a global snapshot)
  - May be be inconsistent
  - May be obsolete
  - Two different monitors may construct two different global states for the same execution.

Global Predicate

- A global predicate Φ is a function of the set of consistent global states of a system to the set $\{\texttt{true, false}\}$.
- The Global Predicate Evaluation (GPE) determines if a global predicate Φ holds for a given global state.

Stable Predicates

- Some properties of the system, at some point during the execution become true, and remain true for the remainder of the execution.
  - We call such properties, stable
- A predicate that describes stable properties is said to be stable.
  - When a system reaches a state at which the predicate evaluates to true,
  - It remains true for all future states that are reachable from its current state.
- Examples of stable predicates:
  - Deadlock
  - Termination
  - Loss of token
  - Garbage collection

Stable Predicates

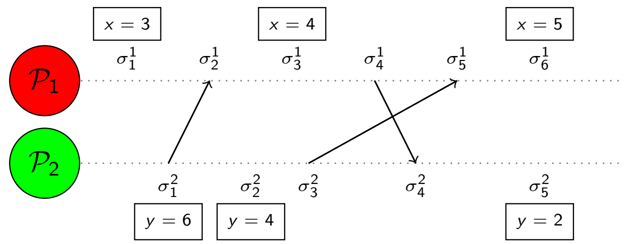
- Let α an execution of a higher level application A.
- We assume the execution of a correct global snapshot algorithm.
  - During the execution of A, the algorithm is activated at Σ_ϵ.
  - It terminates at Σ_τ.
  - It records Σ_*.
- If Φ is stable, it holds that
  - $(\Phi$ is true in $\Sigma_*) \Rightarrow (\Phi$ is true in $\Sigma_\tau)$
  - $(\Phi$ false in $\Sigma_*) \Rightarrow (\Phi$ false in $\Sigma_\tau)$

Non Stable Predicates

- Some cases that we wish to detect cannot be described by stable predicates.
  - Monitor of two queues – notify user when the sum goes above a certain threshold.
  - The queues dynamically change during the execution – the predicate that records the global property is non stable.
- If we evaluate a non-stable global predicate at a given time instance,
  - If may evaluate false – and at some later (or earlier) time instance it may become true.
  - It may evaluate true – while all other time instances it is false.

Non Stable Predicates

Execution Example – send/receive diagram



- $\Phi_1 : x == y$
- $\Phi_2 : y - x == 2$
- If a non-stable predicate is true for a given global state, then the predicate was probably true at the time of the actual execution.

Possibly or Definitely

- We extend global predicate such that
  - They can be applied to the distributed computation,
  - Rather than a specific time instance or specific global states of the executions.
- Our goal is to detect cases when
  - A global predicate is definitely true at some point of the execution that we observer.
  - A global predicate is possibly true.
- In some cases we wish to identify if a property possibly holds.
- In other cases we wish to know if something definitely happened in an execution.

Evaluating Possibly

Possibly(Φ)

There exists at least one consistent observation of the execution Π such that predicate Φ is true in a global state $\Sigma(\Pi)$ of the observation.

- If at least one global state exists for which Φ is true, then there exists at least one execution that is reachable from this state.
- Evaluating Possibly(Φ) requires to searching among all consistent global states.
- Only if $\Phi(\Sigma)$ is false for all consistent global states Σ we can rule out Possibly(Φ).

Evaluating Definitely

Definitely(Φ)

For every consistent observation of the execution Π, there exists a global state $\Sigma(\Pi)$ of the observation such that predicate Φ is true.

- All possible executions of a computation need to be reachable from a given global state for which Φ holds.
- We need to identify a set of states, for which all possible execution are reachable from at least one state sate, and for each such state Φ is true.
- Searching is linear to the number of events.
- Searching is exponential to the number of processes.
  - Let $\max(\sigma)$ bet he maximum number of events, then the number of global states is $\mathcal{O}(\max(\sigma)^n)$.