Modern Distributed Computing

Theory and Applications

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 6
Tuesday, April 16, 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

Asynchronous Message Passing Model

We study distributed systems where

1. the entities of the system execute their actions
   - with **arbitrary** order,
   - and with **arbitrary** speed relative to the other entities,
   - we do not assume any rate of execution actions.
2. the communications channels of the system deliver messages with **arbitrary** speeds relative to the other channels
   - we do not assume any message delivery rate.

Asynchronous Message Passing Model

- We model this undetermined temporal behavior using Input/Output automata
  - Each process is modeled as an I/O automaton,
  - Each communication channel is modeled as an I/O automaton.
- The I/O automata model is generic enough.
  - We can use it to describe almost all types of asynchronous message passing systems.

Input/Output Automata

- An I/O automaton models an entity of the distributed system that interacts with other entities of the system.
- It is a state automaton (state machine) where transitions between states are connected by a set of actions.
- The actions of I/O automaton $\mathcal{A}$ are grouped:
  1. Input Actions – $in(\mathcal{A})$
  2. Output Actions – $out(\mathcal{A})$
  3. Internal Actions – $int(\mathcal{A})$

Input/Output Automata

- The input and output actions are used to model the communication of the automata with their environment
  - Example – an input action is the reception of a message by a neighboring process.
  - Example – an output action is the delivery of a message from a communication channel.
- Internal actions are visible only by the automaton performing the action.
- An automaton does not determine when an input action will be invoked – this depends on its neighboring automata.
  - It can only determine when an output action or and internal action will be executed.

Input/Output Automata

- A set of states $states(\mathcal{A})$
  - Some states are the **initial states** – $start(\mathcal{A})$
  - Some states are the **halting states** – $halt(\mathcal{A})$
- A state transition function $trans(\mathcal{A}) \subseteq states(\mathcal{A}) \times (in(\mathcal{A}) \cup out(\mathcal{A}) \cup int(\mathcal{A})) \times states(\mathcal{A})$
  - For each state κ and for each action ϵ
  - There is a transition $(\kappa, \epsilon, \kappa') \in trans(\mathcal{A})$

Input/Output Automata

An execution of the I/O automaton $\mathcal{A}$ is defined as follows:

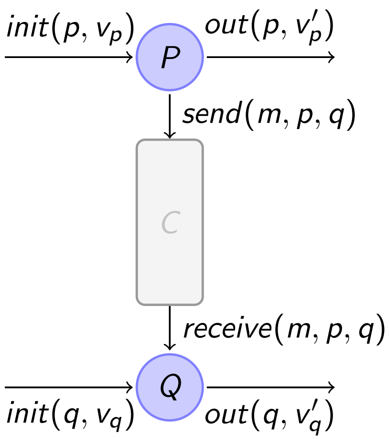
$$\kappa_0,\ \epsilon_1,\ \kappa_1,\ \epsilon_2,\ \ldots\ \epsilon_r,\ \kappa_r,\ \ldots$$

where for each $r \geq 0$ it holds that $(\kappa_r, \epsilon_{r+1}, \kappa_{r+1}) \in trans(\mathcal{A})$

Given the three sets of actions $in(\mathcal{A})$, $out(\mathcal{A})$, $int(\mathcal{A})$, we define

- external actions: $ext(\mathcal{A}) = in(\mathcal{A}) \cup out\mathcal{A}$
- internal actions: $local(\mathcal{A}) = out(\mathcal{A}) \cup int(\mathcal{A})$
- all actions: $actions(\mathcal{A}) = in(\mathcal{A}) \cup out(\mathcal{A}) \cup int(\mathcal{A})$

Modeling Processes

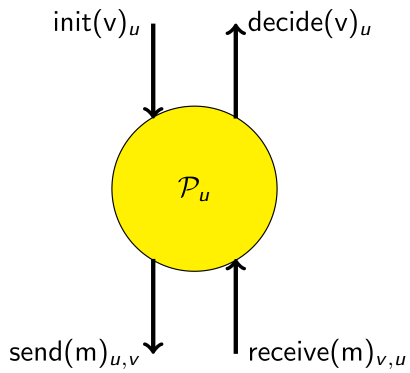


- **I/O automata**
- Defined by a state
  - Special states: initial, halt states
- and a state transition function: $states \times (message_{in}, process) \rightarrow states \times (message_{out}, process)$
- Some tasks may be executed internally $states \times \{(message_{in}, process) \cup \emptyset\} \rightarrow states \times \{(message_{out}, process) \cup \emptyset\}$
- We need to define the notion of fairness

An example of a Process I/O Automaton

- Executes a distributed consensus algorithm
- Input actions$in(\mathcal{P}_u)$
  1. $init(i)_u,\ i \in \mathcal{S}$
  2. $receive(i)_{v,u},$ $i \in \mathcal{S}, 1 \leq v \leq n, v \neq u$
- Output actions $out(\mathcal{P}_u)$
  1. $decide(i)_u,\ i \in \mathcal{S}$
  2. $send(m)_{u,v},\ i \in \mathcal{S}, 1 \leq v \leq n, v \neq u$
- States:
  1. val – vector, indexed by $\{1, \ldots, n\}$ elements of $\mathcal{S} \cup \texttt{null}$, initially `null`

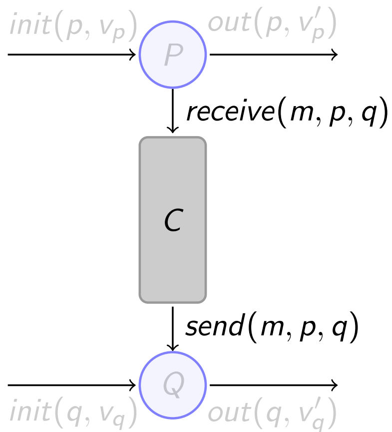
Process $\mathcal{P}_u$



An example of a Process I/O Automaton

- Transitions
  1. $init(i)_u,\ i \in \mathcal{S}$
     - effect – $val(u) = i$
  2. $send(i)_{u,v},\ i \in \mathcal{S}$
     - *precondition* – $val(u) == i$
     - *effect* – none
  3. $receive(i)_{v,u},\ i \in \mathcal{S}$
     - *effect* – $val(v) = i$
  4. $decide(i)_u,\ i \in \mathcal{S}$
     - *precondition* – for each $v, 1 \leq v \leq n :\ val(v) \neq \texttt{null}$ $i = f(val(1), \ldots, val(n))$
     - *effect* – none

Modeling Communication Channels

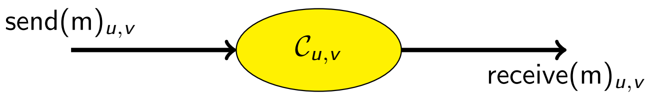


- **I/O automata**
- Defined by a state
  - We assume a queue of messages that need to be delivered
  - e.g., a FIFO priority queue
- Execute actions $receive(m, p, q)$ and $send(m, p, q)$

An example of a Channel I/O Automaton

- Connects processes u, v
- Delivers messages by respecting the order they were received (FIFO)
- Let $\mathcal{M}$ be the message alphabet
- Input actions $in(\mathcal{C}_{u,v})$
  1. $send(m)_{u,v}$, $m \in \mathcal{M}$
- Output actions $out(\mathcal{C}_{u,v})$
  1. $receive(m)_{u,v}$, $m \in \mathcal{M}$

Communication Channel $\mathcal{C}_{u,v}$



An example of a Channel I/O Automaton

- States:
  1. *queue* – a FIFO queue of elements of $\mathcal{M}$, initially empty
- Transitions:
  1. $send(m)_{u,v}$
     - effect – place m in *queue*
  2. $receive(m)_{u,v}$
     - *precondition* – m is in head of *queue*
     - *effect* – remove head of *queue*

Possible executions (the queue state is defined as `[,]`)

`[]`, $send(1)_{u,v}$, `[1]`, $receive(1)_{u,v}$, `[]`, $send(2)_{u,v}$, `[2]`, $receive(2)_{u,v}$, `[]`
`[]`, $send(1)_{u,v}$, `[1]`, $send(2)_{u,v}$, `[12]`, $send(2)_{u,v}$, `[122]`, $receive(1)_{u,v}$, `[22]`, $send(1)_{u,v}$, `[221]`, $receive(2)_{u,v}$, `[21]`, $receive(2)_{u,v}$, `[1]`, ...

An example of a Channel I/O Automaton

Other types of communication channels:

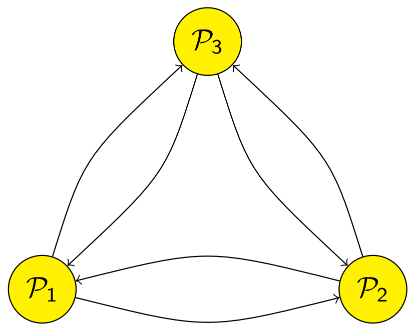
- Reliable, FIFO – deliver all messages by respecting the order they were transmitted (previous example)
- Unreliable, FIFO – Transitions:
  1. $send(m)_{u,v}$ – *effect*: place a finite number of copies of m in *queue*
- Unreliable – States:
  1. `in-transit`– a vector of items of type $\mathcal{M}$, initially empty

  Transitions:
  1. $send(m)_{u,v}$ – *effect*: place a finite number of copies of m in `in-transit`
  2. $receive(m)_{u,v}$ – *precondition*: $m \in$ `in-transit` – *effect*: remove one copy of m from `in-transit`

Composition of I/O Automata

- We model the asynchronous distributed system by composing a set of I/O automata
- We define one automaton

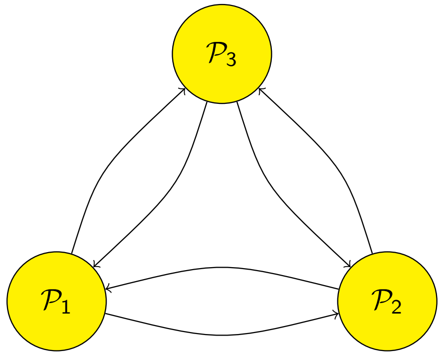
Example of a System



Composition of I/O Automata

Example of a System

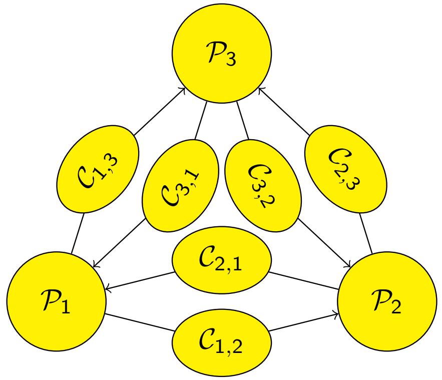
- We model the asynchronous distributed system by composing a set of I/O automata
- We define one automaton
  1. for each process,



Composition of I/O Automata

Example of a System

- We model the asynchronous distributed system by composing a set of I/O automata
- We define one automaton
  1. for each process,
  2. for each communication channel.



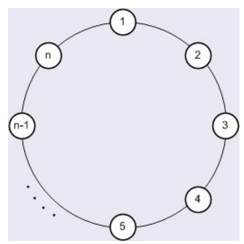
Complexity Measures

- Message complexity
  - We measure the total number of messages transmitted or received.
- Time complexity
  - The undetermined temporal behavior does not allow to measure time complexity in a straight forward way
  - We make the following assumptions when evaluating time complexity
    1. We set an upper bound l for the execution time of every action ϵ at each state κ
    2. We set an upper bound d for the transmission of the oldest message stored in any communication channel

Leader Election

The election of a leader in a network requires the selection of a single, unique, process that will enter a state "leader" (or "elected") while all other processes enter the state "non-leader" (or "non-elected").

Ring network



- We count mod n, allowing 0 to be another name for process n, $n+1$ another name for process 1, ...
- Process with largest ID is u_{max}
- We assume that communication channels are reliable and FIFO

The LCR Algorithm

Algorithm LCR (informal)

Each process sends its identifier around the ring. When a process receives an incoming identifier, it compares that identifier to its own. If the incoming identifier is greater than its own, it keeps passing the identifier; if it is less that its own, it discards the incoming identifier; if it is equal to its own, the process declares itself the leader.

- Originally designed for synchronous systems.
- We can adapt for asynchronous systems,
  - implement an outgoing message queue.

I/O Automaton AsynchLCR$_u$

Actions:

- Input action *in*(AsynchLCR$_u$)
  1. *receive*$(\tau)_{u-1,u}$, where τ a UID
- Output actions *out*(AsynchLCR$_u$)
  1. *send*$(\tau)_{u,u+1}$, where τ a UID
  2. *leader*$_u$

Transitions:

- τ – a UID, initially the UID of u
- *send* – a queue (FIFO) with UID, initial contains only the UID of u
- *status* – may be assigned values {`unknown, chosen, reported`}, initially set to `unknown`.

I/O Automaton AsynchLCR$_u$

Transitions:

- *send*$(\tau)_{u,u+1}$
  - *precondition* – τ head of *send*
  - *effect* – remove head of *send*
- *receive*$(\tau)_{u-1,u}$
  - *effect*

  $\tau > u$ – place τ tail of *send*
  $\tau = u$ – *status* $=$ *chosen*
  $\tau < u$ – nothing
- *leader*$_u$
  - *precondition* – *status* $==$ *chosen*
  - *effect* – *status* $=$ *reported*

Properties of AsynchLCR

- Message complexity $\mathcal{O}(n^2)$
- Time complexity
  - The processing of a message may be delayed in some process where (at most) n messages are in queue – given that the delay of each message is (at most) l, the overall delay is $\mathcal{O}(nl)$ or $\mathcal{O}(nd)$ for the communication channels respectively.
  - Since the message will go through all the processes and all channels, the time complexity is $\mathcal{O}\big(n^2(l+d)\big)$
  - In reality, AsynchLCR is faster than that – if we examine the case more carefully we can show that the time complexity is $\mathcal{O}\big(n(l+d)\big)$

Directed spanning tree

A directed spanning tree of a directed graph $G = (V, E)$ is a rooted tree that consists entirely of directed edges in E, all edges directed from parents to children in the tree, and that contains every vertex of G.

- We can modify SynchBFS for the asynchronous message passing model.
- The algorithm constructs a spanning tree,
- it may not hold the Breadth-First property.

AsynchSpanningTree Algorithm

At any point during execution, there is some set of processes that is "marked", initially just i_0. Process i_0 sends out a search message at round 1, to all of its outgoing neighbors. At any round, if an unmarked process receives a search message, it marks itself and chooses one of the processes from which the search has arrived as its parent. At the first round after a process gets marked, it sends a search message to all of its outgoing neighbors.

- Processes are not aware of the total number of processes (n)
- All processes have UIDs.

I/O Automaton AsynchSpanningTree$_u$

Actions:

- Input actions $in(\text{AsynchSpanningTree}_u)$
  1. $receive(\text{"search"})_{u,v}$, where $v \in nbrs$
- Output actions $out(\text{AsynchSpanningTree}_u)$
  1. $send(\text{"search"})_{u,v}$, where $v \in nbrs$
  2. $parent(v)_u$, where $v \in nbrs$

Στατες:

- $parent \in nbrs \cup \{\text{null}\}$ – initially `null`
- $reported$ – type `boolean`, initially `false`.
- for each $v \in nbrs$ – $send(v) \in \{\text{search}, \text{null}\}$ – initially `search` if $u = u_0$, otherwise `null`

I/O Automaton AsynchSpanningTree$_u$

Transitions:

- $send(\text{"search"})_{u,v}$
  - $precondition$ – $send(v) == search$
  - $effect$ – $send(v) = null$
- $receive(\text{"search"})_{u,v}$
  - $effect$ if $u \neq u_0$ and $parent ==$ null then

    $parent = v$

    for each $k \in nbrs - v$ – $send(k) = search$
- $parent(v)_u$
  - $precondition$ – $parent == v$, $reported == false$
  - $effect$ – $reported = true$

Properties of AsynchSpanningTree

- AsynchSpanningTree constructs a directed spanning tree
  - The distance of any process from u_0 may differ in $T(G)$ and in G.
- The communication complexity is $\mathcal{O}(m)$
- Time complexity:
  - If we do not experience message congestion
  - All processes will have selected a parent process within time $\delta(l+d)+l$

Breadth-First directed spanning tree

A directed spanning tree of G with root i is breadth-first provided that each node at distance d from i in G appears at depth d in the tree.

- We can modify AsynchSpanningTree in order to fix the wrong selected parents.
- If a process receives a search message from a parent that is closer to the root than the existing one, we allow the process to change its parent.
- We need to add a counter in the search messages so that we can measure the distance of each process from the root.

Algorithm AsynchBFS

Each process u holds a variable d_u with its current distance from u_0 (initially if $u \neq u_0$, $d_u = \infty$ otherwise if $u = u_0$, $d_u = 0$). Process u_0 starts the execution by transmitting d_{u_0} to all its neighbors. During each turn, if a process receives a message m from v where $m+1 < d_u$, it sets $d_u = m+1$, and the variable **parent** to the UID of v from which it received the message.

- Let $d(u)$ the distance of u_0 from u in G
- During each execution, for any neighboring u, v either $d_v < d_u + 1$ or d_u is transmitted from u to v

I/O Automaton AsynchBFS$_u$

Actions:

- Input action $in(\text{AsynchSpanningTree}_u)$
  1. $receive(m)_{u,v}$, where $m \in \mathcal{N}, v \in nbrs$
- Output action $out(\text{AsynchSpanningTree}_u)$
  1. $send(m)_{u,v}$, where $m \in \mathcal{N}, v \in nbrs$

States:

- $d_u \in \mathcal{N} \cup \{\infty\}$ – initially 0 if $u = u_0$ otherwise ∞
- $parent \in nbrs \cup \{\text{null}\}$ – initially `null`
- for each $v \in nbrs$ – $send(v)$ – a queue (FIFO) containing elements of $\mathcal{N}$, initially contains 0, if $u = u_0$, otherwise empty.

I/O Automaton AsynchBFS$_u$

Transitions:

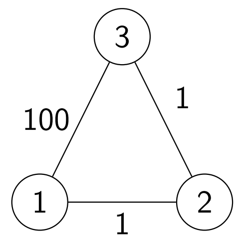
- $send(m)_{u,v}$
  - *precondition* – m head of $send(v)$
  - *effect* – remove head of $send(v)$
- $receive(m)_{u,v}$
  - *effect* – if $m + 1 < d_u$ then

    $parent = v$
    for each $k \in nbrs - v$ – add d_u tail of $send(k)$

Properties of AsynchBFS

- In each time instance of the execution where a d_u is not set to ∞, the value of d_u will be the length of some path connecting u_0 with u
  - $d(u) \leq d_u < n$
  - variable d_u will change value at most n times
- Message complexity is $\mathcal{O}(nm)$

Example



Properties of AsynchBFS

Lemma

For each u within time $d(u)n\big((l) + (d)\big)$ it holds that $d_u = d(u)$.

- For $d(u) = 0$ it is trivial.
- Let assume that it holds for every v where $d(v) \leq k$
- Let process u with $d(u) = k + 1$ and process v (neighboring of u) with $d(v) = k$
- Within time $kn\big((l) + (d)\big)$, process v has set $d(v) = k$ and has decided to send k to process u
- Within additional time $n(l)$, process v will send k to $\mathcal{C}_{vu}$
- Within additional time $v(d)$, process u will receive it, set $d_u = k + 1$ and choose v as parent.

Properties of AsynchBFS

Theorem

The execution of AsynchBFS converges to a configuration where the processes have constructed a breadth-first spanning tree $T(G)$ such that the distance of each vertex from u_0 is the same in G and in $T(G)$ and this is completed within time $\mathcal{O}\Big(\delta n\big((l) + (d)\big)\Big)$

- The convergence technique is common for asynchronous distributed systems.