Modern Distributed Computing

Theory and Applications

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 9
Tuesday, May 7, 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

Termination of Distributed Computation

- A distributed algorithm is said to terminate when all processes reach a halting state.
  - Upon reaching such a state no further progress can be achieved.
- If all processes reach a halting state we say that external termination has been achieved.
- There are cases where no further progress can be achieved but some (or all) process are not in a halting state.
  - e.g., each process receives messages but never sends out messages.
  - The distributed algorithm has terminated yet the processes are not aware.
  - We say that internal termination has been achieved.

Termination of Distributed Computation

- Internal termination is also known as communication termination.
  - No further messages are transmitted, eventually communication seizes.
- External termination is also known as process termination.
  - All processes reach a halting state.
- It is easier to design an algorithm that achieves internal termination (e.g., LCR, SyncBFS, . . .).
- Essentially we ignore the final stage of the computation where external termination is achieved.
- In some cases we have to include this final step in our algorithm.
  - e.g., when committing transactions, releasing common resources, . . .

Properties of Termination

- We work using the definition of termination detection, as defined by Dijkstra
- Each process may be in one of the following states:
  1. Active state
  2. Passive state
- Only active processes may send messages (perform an output action).
- Upon receiving a message (input action), a passive process becomes active.
- The reception of a message is the only event that may flip a passive process to become active.
- Each active process may become passive spontaneously, at any time (due to an internal action).

Dijkstra and Scholten Algorithm

- Let $\mathcal{P}_0$ be the coordinating process.
- The algorithm constructs a inverted spanning tree with process $\mathcal{P}_0$ as the root.
- The tree is modified while the higher-level algorithm is executed in a way such:
  - the active processes are located near the root (small height),
  - the passive processes are located on the leaves of the tree (large height).
- These trees are also known as Computation Trees.
- Termination is detected when the root of the tree becomes passive.

Dijkstra and Scholten Algorithm

- We assume that the higher-lever algorithm is centralized, that is, initially only $\mathcal{P}_0$ is active.
- The higher-level algorithm is based on diffusing computations.
- Each process stores a local pointer to the parent process in the tree.
  - If for process $\mathcal{P}_u$, $parent == null$, we say $\mathcal{P}_u$ is free.
- Each process maintains a local counter children storing the number of children in the tree.

Dijkstra and Scholten Algorithm

- Consider process $\mathcal{P}_u \neq \mathcal{P}_0$, a free process, that receives a message from $\mathcal{P}_v$.
  1. It sets $parent = \mathcal{P}_v$ (the edge uv is inserted in the tree),
  2. Informs $\mathcal{P}_v$ (via a control message),
  3. Process $\mathcal{P}_v$ sets $children_v ++$.
- $\mathcal{P}_u$ is not free, and at some points become passive:
  1. Informs $\mathcal{P}_v$ (via a control message),
  2. Process $\mathcal{P}_v$ sets $children_v --$,
  3. $\mathcal{P}_u$ sets $parent = null$ (the edge uv is removed).
- Thus, all "isolated" processes (with no adjacent edges) are passive processes.
- When $\mathcal{P}_0$ becomes passive, the algorithm terminates.

Correctness of Algorithm

Theorem

The Dijkstra–Scholten algorithm correctly detects termination using M control messages, where M is the number of messages exchanged by the higher-level algorithm.

- The algorithm offers a very good balance between control messages and data messages.
- Based on the lower bound M (see next theorem) the algorithm is optimal.

Correctness of Algorithm

Proof:

- Let the computation tree $T = (V_T, E_T)$
- Either T is empty or it is directed to the root $\mathcal{P}_0$.
- The set V_T includes all active processes and all messages in transit.
- The coordinator $\mathcal{P}_0$ invokes the sub-algorithm informing all nodes about termination when $\mathcal{P}_0 \notin V_T$.
  - Since $|V_T| = 0$ the predicate *term* is true.
- Essentially, T expands every time a data message is sent or when a processes becomes active.

Correctness of Algorithm

Proof:

- To guarantee progress for the termination detection algorithm, the tree has to "empty" in finite number of steps after the termination of the higher-level algorithm.
- The proof requires that T is a tree and becomes empty only after the higher-level algorithm terminates.
- For each execution γ of the higher-level algorithm we define

$$\begin{aligned} V_T = \quad & \{u : parent_u \neq null\} \\ \cup \quad & \{\text{data messages in transit}\} \\ \cup \quad & \{\text{control messages in transit}\} \end{aligned}$$

Correctness of Algorithm

Proof:

$$\begin{aligned} E_T = \quad & \{(u, parent_u) : parent_u \neq null \wedge parent_u \neq u\} \\ \cup \quad & \{\text{data messages in transit}\} \\ \cup \quad & \{\text{control messages in transit}\} \end{aligned}$$

Safety is based on the following condition P defined as follows:

$$\begin{aligned} P = \quad & state_u == active \Rightarrow u \in V_T && (1) \\ \wedge \quad & (u, v) \in E_T \Rightarrow u \in V_T \wedge v \in V_T \cap P && (2) \\ \wedge \quad & children_u = \#v : (v, u) \in E_T && (3) \\ \wedge \quad & V_T \neq \emptyset \Rightarrow T \text{ tree, rooted on } \mathcal{P}_0 && (4) \\ \wedge \quad & (state_u == passive \wedge children_u == 0) \Rightarrow u \in V_T && (5) \end{aligned}$$

Correctness of Algorithm

Proof:

1. $state_u == active \Rightarrow u \in V_T$
   Graph T includes all active processes.
2. $(u, v) \in E_T \Rightarrow u \in V_T \wedge v \in V_T \cap P$
   T is a tree and all edges are directed towards some process.
3. $children_u = \# v : (v, u) \in E_T$
   Processes properly count their children.
4. $V_T \neq \emptyset \Rightarrow T$ tree, rooted on $\mathcal{P}_0$
   Graph T is a tree, rooted on $\mathcal{P}_0$
5. $(state_u == passive \wedge children_u == 0) \Rightarrow u \in V_T$
   The tree is empty when the higher-level algorithm terminates.

Correctness of Algorithm

Proof:

- The proof of correctness is based on the observation that in condition P it holds that $parent_u == u$ only for $u == \mathcal{P}_0$.

Lemma

Condition P holds for the Dijkstra–Scholten algorithm.

- Let $S = \sum_{u=0}^{n} children_u$ the sum of all children counters.
  - Initially $S = 0$,
  - increases when the next control message is sent,
  - decreases when a control message is received,
  - cannot go negative, due to (3).

Correctness of Algorithm

Proof:

- After the higher-level algorithm terminates, only actions of the termination detection algorithm will be executed.
- Since S decreases after each such action, the termination detection algorithm will also terminate.
- In such a state, V_T does not contain any message in transit.
- Due to (5), V_T will not include any passive process.
- Thus T will have no leaves, and thus become empty.
- The tree will be empty when $\mathcal{P}_0$ will remove itself.
- The liveness requirement is guaranteed.

Correctness of Algorithm

Proof:

- Proving safety is done based on the observation that $\mathcal{P}_0$ will invoke the sub-algorithm informing all nodes about termination before removing itself from V_T.
- Thus, due to (4), T will be empty when this takes place.
- Clearly, the non-interference condition holds.

$\blacksquare$

Synchronous vs Asynchronous Execution

- In Synchronous Systems – we assume synchronized execution.
  - The assumption is too strong and is not very realistic.
  - Based on this assumption we can design efficient algorithmic solutions.
  - Based on this assumption we can evaluate the performance of the system.
- In Asynchronous Systems – we avoid this assumption.
  - It is more realistic.
  - We may assume some upper bounds to study the performance of the system.
  - To achieve synchronized execution we need additional code.

Distributed Data Structures

- Spanning Tree construction – process u, constructs a spanning $T_u(G)$, rooted on u.
- Algorithm – AsynchSpanningTree
  - Message Complexity – $\mathcal{O}(n \cdot m)$
  - Time complexity – $\mathcal{O}\big(\delta(l + d)\big)$
- Algorithm – AsynchBFS
  - Message Complexity – $\mathcal{O}(n \cdot m)$
  - Time complexity – $\mathcal{O}\big(n \cdot \delta(l + d)\big)$

Distributed Data Structures

- Spanning Tree construction – process u, constructs a spanning $T_u(G)$, rooted on u.
- Algorithm – AsynchSpanningTree
  - Message Complexity – $\mathcal{O}(n \cdot m)$
  - Time complexity – $\mathcal{O}\big(\delta(l + d)\big)$
- Algorithm – AsynchBFS / SyncBFS
  - Message Complexity – $\mathcal{O}(n \cdot m)$ / $\mathcal{O}(n \cdot m)$
  - Time complexity – $\mathcal{O}\big(n \cdot \delta(l + d)\big)$ / $\mathcal{O}(\delta)$

Discussion

- Observe that some algorithms designed for synchronous systems are more efficient in terms of time and message complexity.
  - How can we adjust them for asynchronous systems ?
- The existence of a clock can be used to efficiency solve many problems
  - Synchronization Problem
  - Commit Problem
  - Authorization
  - ... *[B.Liskov, PODC'91]*

Synchronizers

- In synchronous execution, proper design leads to improved efficiency both in terms of time and message complexity.
- In asynchronous execution, we wish to "guarantee" some kind of synchrony by using a synchronizer.
- Then we can combine algorithms for synchronous execution with a synchronizer so that they can be suitable for asynchronous execution.
- In some sense, synchronizers, transform algorithms originally designed for synchronous systems, to execute on asynchronous systems.

Design Issues

Design approach:

1. We set the problem – e.g., spanning tree construction, BFS, mutual exclusion, ...
2. We model the system using an asynchronous model.
3. We design a new algorithm or apply an existing solution.

Alternative approach:

- We intervene an "intermediate level" between the hardware (processor, channel) and the algorithm (processes).
- The "middle layer" makes the underlying system "look like" a synchronous system.

Design Issues

Alternative Design approach:

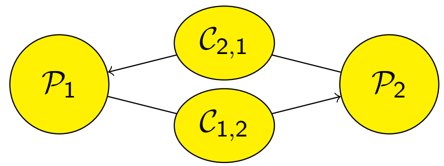
1. We set the problem – e.g., spanning tree construction, BFS, mutual exclusion, ...
2. We model the system using an asynchronous model.
3. We introduce a "middle layer" for synchronization.
4. We design a new algorithm or apply an existing solution for synchronous systems.

In this way we transform synchronous algorithms for asynchronous mode of execution.

Design Issues

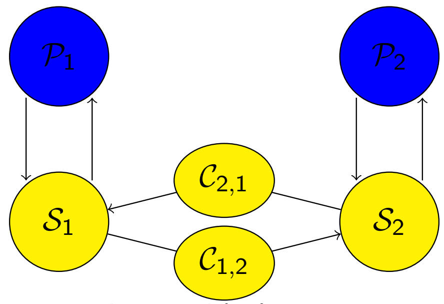
1^{η} approach

```
compile algo-asynch.nc
execute
```



2^{η} approach

```
compile algo-synch.nc
link synchronizer
execute
```



The *basic idea*: at each node u the process communicates with the rest of the system via the synchronizer – the synchronizer "hides" from the synchronous process the asynchronous mode of execution.

Synchronization Problem

We assume that each processor (node) executes 2 processes:

1. Process $\mathcal{P}$ that corresponds to the synchronous protocol.
2. Process $\mathcal{S}$ that corresponds to the asynchronous automaton of the synchronizer.

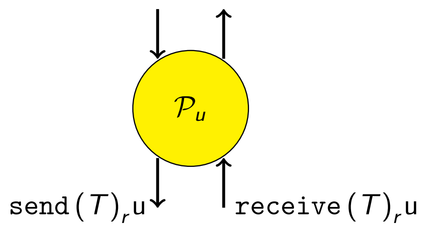
Synchronization Problem

Algorithm $\mathcal{A}$ solves the synchronization problem if it provides an execution environment where process $\mathcal{P}$ cannot distinguish if it is executed in the asynchronous system (in combination with the synchronizer) or if it is executed in a synchronous system.

Specifications for I/O Automaton $\mathcal{P}$

Process $\mathcal{P}_u$

- Let $\mathcal{M}$ a fixed message alphabet used by the algorithm during the execution in the synchronous system.
- For each message m we assign a label v signifying the recipient of the message.



- Output action of $\mathcal{P}_u$ is of type $\texttt{send}(T)_r\texttt{u}$ where
  - T – a set of labeled messages (e.g. $\{\langle m, v\rangle\}$)
  - $r \in \mathcal{N}^+$ – the round of the synchronous system during which the action takes place.

Specifications for I/O Automaton $\mathcal{P}$

- Input action for $\mathcal{P}_u$ is of type $\texttt{receive}(T)_r\texttt{u}$ where
  - T – a set of labeled messages (e.g. $\{\langle m, v\rangle\}$)
  - $r \in \mathcal{N}^+$ – the round of the synchronous system during which the action takes place.
- If $\mathcal{P}_u$ does not have any outgoing message during round r, then it executes action $\texttt{send}(null)_r\texttt{u}$

Execution Example for automaton $\mathcal{P}$

Let $n = 3$. The action $\texttt{send}(\{\langle m_1, 1\rangle, \langle m_2, 2\rangle\})_4 3$ implies that during round 4, process $\mathcal{P}_3$ transmits message m_1 to $\mathcal{P}_1$ and m_2 to $\mathcal{P}_2$. Similarly, action $\texttt{receive}(\{\langle m_1, 1\rangle, \langle m_2, 2\rangle\})_4 3$ implies that during round 4, process $\mathcal{P}_3$ receives message m_1 from $\mathcal{P}_1$ and m_2 from $\mathcal{P}_2$.

SimpleSynch Algorithm

For each round r, process $\mathcal{S}_u$ collects $\texttt{send}(T)_r\texttt{u}$ from $\mathcal{P}_u$, and for each $\langle m, v\rangle \in T$ it sends $\langle m, r\rangle$ to $\mathcal{S}_v$. For each message $\langle m, r\rangle$ received from $\mathcal{S}_v$, it inserts $\langle m, r\rangle$ to vector T_r. When a message is received from *each* neighboring $\mathcal{S}_v$ during round r, it delivers $\texttt{receive}(T)_r\texttt{u}$ to $\mathcal{P}_u$.

- If $\mathcal{P}_u$ does not have any messages to transmit during round r to process $\mathcal{P}_v$, we assume that it "fills-in" T with $\langle null, v\rangle$.
- A simple implementation of $\mathcal{S}$.
- SimpleSynch operates at "local level" – processes coordinate to synchronize the rounds of the higher-level algorithms.

SimpleSynch$_u$ Automaton

Actions:

- Input actions *in*(SimpleSynch$_u$)
  1. `send` $(T)_r$u – where T a labeled set of messages, $r \in \mathcal{N}^+$
  2. `net-receive` $(N, r)_{v,u}$ – where N a set of messages, $r \in \mathcal{N}^+$, $v \in nbrs_u$
- Output actions *out*(SimpleSynch$_u$)
  1. `receive` $(T)_r$u – where T a labeled set of messages, $r \in \mathcal{N}^+$
  2. `net-send` $(N, r)_{u,v}$ – where N a set of messages, $r \in \mathcal{N}^+$, $v \in nbrs_u$

SimpleSynch$_u$ Automaton

States:

- `proc-sent`, `proc-rcvd` – boolean vectors, indexed by $\mathcal{N}^+$, initially all elements set to `false`
- `net-sent`, `net-rcvd` –boolean vectors, indexed by $nbrs_u \times \mathcal{N}^+$, initially all elements set to `false`
- `outbox` – an array of message sets, indexed by $nbrs_u \times \mathcal{N}^+$, initially all elements set to `null`
- `inbox` – an array of labelled message sets, $\mathcal{N}^+$, initially all elements set to `null`

SimpleSynch$_u$ Automaton

Transitions:

- `send` $(T)_r$u
  - *effect*:

    `proc-sent(r) = true`

    for each $v \in nbrs_u$, `outbox(v,r)` $= \{m | \langle m, v\rangle \in T\}$
- `net-send` $(N, r)_{u,v}$
  - *precondition:*

    `proc-sent(r) == true`

    `net-sent(v,r) = false`

    `N = outbox(v,r)`
  - *effect*:

    `net-sent(v,r) = true`

SimpleSynch$_u$ Automaton

Transitions:

- `net-receive` $(N, r)_{v,u}$
  - *effect*:

    `inbox(r) = inbox(r)` $\cup \{\langle m, v\rangle | m \in N\}$

    `net-rcvd(v,r) = true`
- `receive` $(T)_r$u
  - *precondition:*

    `proc-sent(r) == true`

    for each $v \in nbrs_u$, `net-rcvd(v,r) == true`

    `T = inbox(r)`

    `proc-rcvd(r) == false`
  - *effect*:

    `proc-rcvd(r) = true`

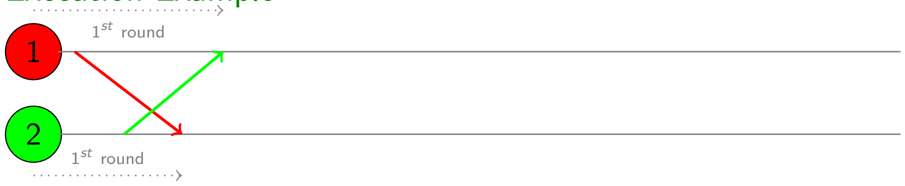
Properties of SimpleSynch

For each simulated round:

- $2m$ messages are exchanged,
- Process $\mathcal{P}_u$ requires time $\mathcal{O}(l)$,
- Process $\mathcal{S}_u$ requires time $\mathcal{O}(l)$.

For simulating r rounds, a total of $r\,(d + \mathcal{O}(l))$ is required.

Execution Example



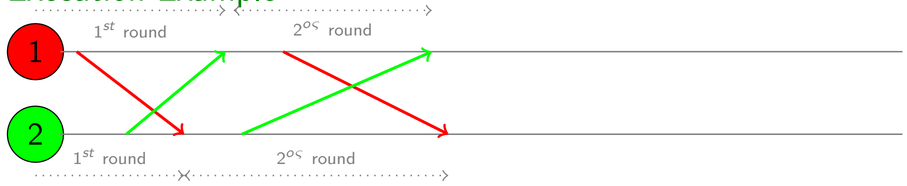
Properties of SimpleSynch

For each simulated round:

- $2m$ messages are exchanged,
- Process $\mathcal{P}_u$ requires time $\mathcal{O}(l)$,
- Process $\mathcal{S}_u$ requires time $\mathcal{O}(l)$.

For simulating r rounds, a total of $r\,(d + \mathcal{O}(l))$ is required.

Execution Example



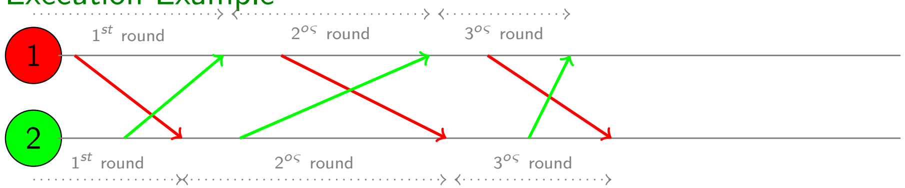
Properties of SimpleSynch

For each simulated round:

- $2m$ messages are exchanged,
- Process $\mathcal{P}_u$ requires time $\mathcal{O}(l)$,
- Process $\mathcal{S}_u$ requires time $\mathcal{O}(l)$.

For simulating r rounds, a total of $r\,(d + \mathcal{O}(l))$ is required.

Execution Example



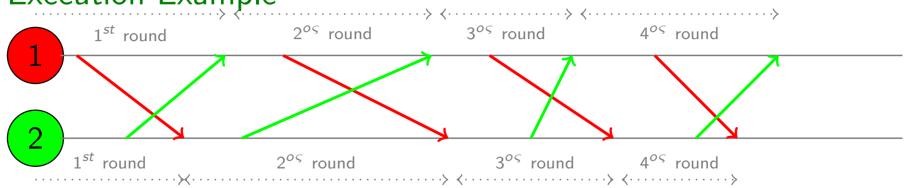
Properties of SimpleSynch

For each simulated round:

- $2m$ messages are exchanged,
- Process $\mathcal{P}_u$ requires time $\mathcal{O}(l)$,
- Process $\mathcal{S}_u$ requires time $\mathcal{O}(l)$.

For simulating r rounds, a total of $r\,(d + \mathcal{O}(l))$ is required.

Execution Example



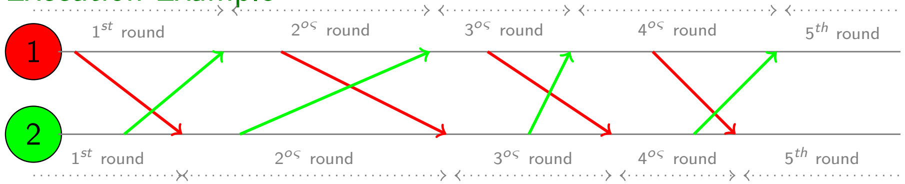
Properties of SimpleSynch

For each simulated round:

- $2m$ messages are exchanged,
- Process $\mathcal{P}_u$ requires time $\mathcal{O}(l)$,
- Process $\mathcal{S}_u$ requires time $\mathcal{O}(l)$.

For simulating r rounds, a total of $r\,(d + \mathcal{O}(l))$ is required.

Execution Example



Tel — Leeuwen Synchronizer

- The synchronizer uses the following assumptions:
  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
- Asynchronous systems that adhere to the above assumptions are known as *Asynchronous Bounded-Delay Networks*
- Under these assumptions it is fairly easy to implement a synchronizer.
- The only design issue is to guarantee that all messages sent during round r have been properly received before round $r+1$ is about to start.

Tel — Leeuwen Synchronizer

- We assume that all nodes are equipped with a local clock.
- We assume that clocks are synchronized.
- There is an upper bound $\mu \geq l + d$ that is known to all processes.
- No need for $\mathcal{P}_u$ to transmit `null` messages during a round r where no actual messages are transmitted.

ABD Algorithm

During each round r, process $\mathcal{S}_u$ after receiving all $\texttt{send}\,(T)_r\texttt{u}$ messages from $\mathcal{P}_u$, for each $\langle m, v\rangle \in T$ sends a message $\langle m, r\rangle$ to $\mathcal{S}_v$. For each message $\langle m, r\rangle$ received from $\mathcal{S}_v$, it inserts $\langle m, r\rangle$ in vector T_r. When the local *clock* reaches $r \cdot 2 \cdot \mu$, it delivers the final $\texttt{receive}\,(T)_r\texttt{u}$ to $\mathcal{P}_u$.

ABD_u Automaton

Actions:

- Input actions $in(\text{ABD}_u)$
  1. $\texttt{send}\,(T)_r\texttt{u}$ – where T a labeled set of messages, $r \in \mathcal{N}^+$
  2. $\texttt{net-receive}\,(N, r)_{v,u}$ – where N a set of messages, $r \in \mathcal{N}^+$, $v \in nbrs_u$
- Output actions $out(\text{ABD}_u)$
  1. $\texttt{receive}\,(T)_r\texttt{u}$ – where T a labeled set of messages, $r \in \mathcal{N}^+$
  2. $\texttt{net-send}\,(N, r)_{u,v}$ – where N a set of messages, $r \in \mathcal{N}^+$, $v \in nbrs_u$

ABD_u Automaton

States:

- `clock`$_u$ – a local clock
- `round` – integer variable, initially set to 1
- `pulse` – time variable
- `proc-sent`, `proc-rcvd` – boolean vectors indexed by $\mathcal{N}^+$, initially all elements set to `false`
- `outbox` – an array of message sets, indexed by $nbrs_u \times \mathcal{N}^+$ initially all rows are `null`
- `inbox` – an array of labeled message sets, indexed by $\mathcal{N}^+$, initially all rows are `null`

ABD_u Automaton

Transitions:

- `send` $(T)_r$`u`
  - *effect*:
    `proc-sent(r) = true`
    for each $v \in nbrs_u$, `outbox(v,r)` $= \{m | \langle m, v\rangle \in T\}$
- `net-send` $(N, r)_{u,v}$
  - *precondition:*
    `proc-sent(r) == true`
    `N = outbox(v,r)`

ABD_u Automaton

Transitions:

- `net-receive` $(N, r)_{v,u}$
  - *effect*:
    `inbox(r) = inbox(r)` $\cup \{\langle m, v\rangle | m \in N\}$
- `receive` $(T)_r$`u`
  - *precondition:*
    `clock`$_u$ `- pulse ==` $2 \cdot$ `round` $\cdot \mu$
    `T = inbox(r)`
    `proc-rcvd(r) == false`
  - *effect*:
    `proc-rcvd(r) = true`

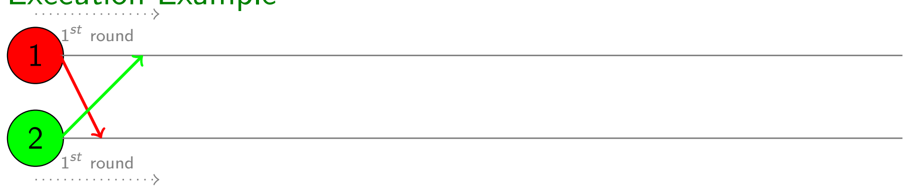
Properties of ABD Algorithm

For each round:

- no message exchanged by process $\mathcal{S}_u$.
- Process $\mathcal{P}_u$ requires $\mathcal{O}(l)$ time.
- Process $\mathcal{S}_u$ requires $\mathcal{O}(l)$ time.

For simulating r synchronous rounds we need $r \cdot \mathcal{O}(d + l)$ rounds.

Execution Example



Properties of ABD Algorithm

For each round:

- no message exchanged by process $\mathcal{S}_u$.
- Process $\mathcal{P}_u$ requires $\mathcal{O}(l)$ time.
- Process $\mathcal{S}_u$ requires $\mathcal{O}(l)$ time.

For simulating r synchronous rounds we need $r \cdot \mathcal{O}(d + l)$ rounds.

Execution Example

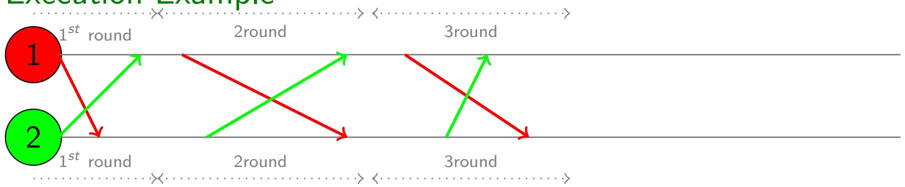
Properties of ABD Algorithm

For each round:

- no message exchanged by process $\mathcal{S}_u$.
- Process $\mathcal{P}_u$ requires $\mathcal{O}(l)$ time.
- Process $\mathcal{S}_u$ requires $\mathcal{O}(l)$ time.

For simulating r synchronous rounds we need $r \cdot \mathcal{O}(d + l)$ rounds.

Execution Example



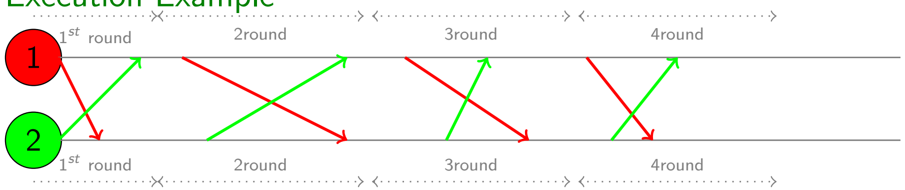
Properties of ABD Algorithm

For each round:

- no message exchanged by process $\mathcal{S}_u$.
- Process $\mathcal{P}_u$ requires $\mathcal{O}(l)$ time.
- Process $\mathcal{S}_u$ requires $\mathcal{O}(l)$ time.

For simulating r synchronous rounds we need $r \cdot \mathcal{O}(d + l)$ rounds.

Execution Example



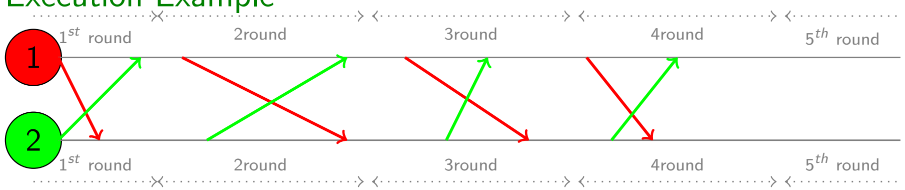
Properties of ABD Algorithm

For each round:

- no message exchanged by process $\mathcal{S}_u$.
- Process $\mathcal{P}_u$ requires $\mathcal{O}(l)$ time.
- Process $\mathcal{S}_u$ requires $\mathcal{O}(l)$ time.

For simulating r synchronous rounds we need $r \cdot \mathcal{O}(d + l)$ rounds.

Execution Example



Discussion

algorithm	AsynchBFS	SynchBFS SimpleSync	SynchBFS ABD
time	$\mathcal{O}(\delta \cdot n(d+l))$	$\mathcal{O}(\delta(d+l))$	$\mathcal{O}(\delta \cdot \mu)$
messages	$\mathcal{O}(n \cdot m)$	$\mathcal{O}(n \cdot m^2)$	$\mathcal{O}(n \cdot m)$

- The time complexity of ABD is an upper bound for SimpleSync.