Lecture 4: Synchronizers

CS 539 / ECE 526

Distributed Algorithms
Outline

• Lockstep rounds too strong assumption

• How to enforce lockstep rounds?
  – In synchrony: clock synchronization
  – Today: In asynchrony: synchronizers
Synchronizers

- Enforce lockstep rounds in asynchrony
- Message passing
- Generic graph
- No failure
Outline

• A simple local synchronizer

• Awerbuch’s framework
  – An alternative local synchronizer
  – A global synchronizer
  – Hybrid local/global synchronizer

• Fault tolerance of synchronizers

• Correctness of local synchronizers
A Simple Synchronizer

• Idea: a process can send round-(r+1) msgs once it receives all round-r msgs
  (all msgs are marked with round number)

  – Having received round-(r+1) msgs before that?

    • Simply delay processing those

    • Similarly, could be too earlier for other processes, but others can also just buffer round-(r+1) msg
A Simple Synchronizer

• Idea: a process can send round-(r+1) msgs once it receives all round-r msgs (all msgs are marked with round number)

• Send “NoMsg, r” if there is nothing to send
  – Do this separately for every link

• Move to round r+1 upon receiving round-r msgs (or NoMsg) from ALL neighbors
A Simple Synchronizer

• This synchronizer is **local**

• Nearby nodes are off by 1 round at most
  – Node i is waiting for round-\(r\) msgs
  – Node i has not sent its round-\((r+1)\) msg or NoMsg
  – Node i’s neighbors cannot start round \(r+2\)

• Far-apart nodes may be off by many rounds
A Simple Synchronizer

• Far-apart nodes may be off by many rounds

A -------- B -------- C -------- D -------- E
Synchronizer Correctness

• Far-apart nodes may be off by many rounds
• Is this really equivalent to lockstep rounds?
• For external observers, no!
  – Also for lockstep using clock synchronization
• For the nodes themselves?
  – Feels like it, but how do we formally prove it? Not trivial, will come back to it
A Simple Synchronizer: Efficiency

• Transforms a lockstep algo into an async one

• Efficiency: measured by blowup

• Round blowup: 1x (i.e., none)

• Message blowup
  – $M \to R^*|E|$ where $R$ is the lockstep round complexity

• Good for rounds, potentially bad for comm
  – When is communication blowup small?
Outline

• A simple local synchronizer

• Awerbuch’s framework
  – An alternative local synchronizer
  – A global synchronizer
  – Hybrid local/global synchronizer

• Fault tolerance of synchronizers

• Correctness of local synchronizers
Awerbuch’s Synchronizers

• A general class of synchronizers

• Do not send NoMsg. ACK every msg.

• A node is “done sending in round r” if all its round-r msgs have been ack’ed

• If ALL neighbors are “done sending in round r”, a node has received all round-r msgs
  – Hence, can send round r+1 msgs
  – Question left: how to communicate “done sending”
Awerbuch’s Synchronizers

• Only question left: how to communicate “done sending in round r”

• Option 1: simply send to all neighbors
  – Called Alpha Synchronizer by Awerbuch

• This gives an alternative local synchronizer
  – Round and communication blowup?
  – No advantage over the simpler one, but helpful for reasoning about more complex synchronizers
Awerbuch’s Synchronizers

- Only question left: how to communicate “done sending in round r”

- Option 1 (alpha): simply send to all neighbors

- Option 2 (beta): via a leader and spanning tree
  - Convergecast “done sending r” to root / leader
  - Leader broadcasts “start round r+1”
Awerbuch’s Beta Synchronizer

• A global synchronizer

• No process sends round-\((r+1)\) msg until ALL round-\(r\) msgs (from/to all procs) are received

• Correctness straightforward / by definition
Beta Synchronizer Efficiency

• Round blowup
  – $R$ to $R^*(2+2D)$ where $D$ is the depth of spanning tree
  – But $D$ could be $|V|$ in async if unlucky

• Message blowup
  – $M$ to $2M + 2*R*|V|$ 
  – $2M$ from acks, rest are convergecast & broadcast
Awerbuch’s Synchronizers

• Only question left: how to communicate “done sending in round r”

• Option 1 (alpha): simply send to all neighbors

• Option 2 (beta): via a leader and spanning tree

• Option 3 (gamma): tradeoff between 1 and 2
Awerbuch's Gamma Synchronizer

• A spanning forest (multiple spanning trees)
  – E.g., $b \rightarrow a/d$, $e \rightarrow f \rightarrow c$, $g \rightarrow h$

• First, beta synchronizer within each tree

• Then, alpha synchronizer among roots
  – Root: “done r” (for my tree)
  – Go to round $r+1$ if my tree and all neighboring trees send “done r”
Awerbuch’s Gamma Synchronizer

- Which trees are neighboring trees?
  - If and only if any of their members are in contact

- Is it OK to have no link between b and g?
  - OK in this example
  - Not OK if d --- g (or a --- h)
Awerbuch’s Gamma Synchronizer

• Correctness
  – All my neighbors are in same or neighboring trees
  – My root broadcasts “start round r+1” if it receives “done r” from our entire tree (via convergecast) AND all neighboring roots

  • Former takes care of my neighbors in same tree
  • Latter takes care of my neighbors in neighboring trees
Awerbuch’s Gamma Synchronizer

• Efficiency depends on forest structure

• Example: k trees of size n/k, roots form clique
  
  – Round blowup: depth of tree, so $O(n/k)$
  
  – Msg blowup: $M$ to $2M + R(2k*n/k + (n/k)^2)$
  
  – Tune $k$ for a trade-off between round and msg (between alpha and beta), e.g., $k = \sqrt{n}$ is typical
Outline

• A simple local synchronizer

• Awerbuch’s framework
  – An alternative local synchronizer
  – A global synchronizer
  – Hybrid local/global synchronizer

• Fault tolerance of synchronizers

• Correctness of local synchronizers
Fault Tolerance

• None of the synchronizers today tolerates even a single crash fault
  – Fault tolerant synchronizer impossible!

• Clock synchronization using a reference also does not tolerate a single crash
  – Fault tolerant clock synchronization is possible (in synchrony)
Fault Tolerance

• Fault tolerant synchronizer impossible!

• Proof sketch:
  – If no one hears from node x, what do we do?
  – Must move on eventually (liveness)
    • Cannot wait forever, x may have crashed
  – But x could be just slow due to asynchrony
    • Moving on violates correctness (safety)
Safety and Liveness

• Desired property: “good” things happen
• Common and helpful to break it down
• Safety: nothing “bad” happens
• Liveness: something happens
Outline

• A simple local synchronizer

• Awerbuch’s framework
  – An alternative local synchronizer
  – A global synchronizer
  – Hybrid local/global synchronizer

• Fault tolerance of synchronizers

• Correctness of local synchronizers
Correctness of Synchronizers

• Desired property: equivalence to lockstep

• Straightforward for global synchronizers

• Want to show other synchronizers are equivalent to global synchronizer

• How do we define equivalence?
  – Intuitively today, rigorously next lecture
Equivalence of Executions

• We have seen one example
• Again, not equivalent for external observers
• In asynchrony, process cannot rely on time
  – Unlike in synchrony

Proc 1’

Proc 2’

Proc 1

Proc 2
Equivalence of Executions

• We have seen one example

• Again, not equivalent for external observers

• In asynchrony, process cannot rely on time
  – Unlike in synchrony
• Recall guarantee: a process sends round-\((r+1)\) msgs once it receives all round-\(r\) msgs

  – A process reads round-\(r\) msgs (from others) only after it finishes sending round-\(r\) msgs

• So the local view at one process looks like

```plaintext
Proc 1

Round 1  Round 2  Round 3  Round 4

=  =  ++ =  ++  +++  +++  +++  +++  +++  +++  +++

= send
+ receive
```
Correctness of Synchronizers

- An execution that results from a local/hybrid synchronizer may look “unsynchronized”

- But it is equivalent to ...

\[
\begin{align*}
= & \text{ send} \\
+ & \text{ receive}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Round 1</th>
<th>Round 2</th>
<th>Round 3</th>
<th>Round 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proc 1</td>
<td>=</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>Proc 2</td>
<td>==++</td>
<td>==</td>
<td>=+</td>
</tr>
</tbody>
</table>
Correctness of Synchronizers

• A globally synchronized execution
  – Events ordered by rounds
  – Within a round, send events before receive events

= send
+ receive

<table>
<thead>
<tr>
<th>Round</th>
<th>Proc 1</th>
<th>Proc 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>==</td>
<td>==</td>
</tr>
<tr>
<td></td>
<td>+++==</td>
<td>+++==</td>
</tr>
<tr>
<td></td>
<td>++++==</td>
<td>++++==</td>
</tr>
<tr>
<td>2</td>
<td>+++++=</td>
<td>+++++=</td>
</tr>
<tr>
<td>3</td>
<td>+++++=</td>
<td>+++++=</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>++++++</td>
</tr>
<tr>
<td></td>
<td>++</td>
<td>+++=++</td>
</tr>
</tbody>
</table>
Correctness of Synchronizers

- Why not the following? Is it also equivalent?
- How do we define equivalence formally?
- Topics for next lecture, exercise for now!

= send
+ receive

<table>
<thead>
<tr>
<th>Round 1</th>
<th>Round 2</th>
<th>Round 3</th>
<th>Round 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proc 1</td>
<td>=</td>
<td>=+++</td>
<td>===</td>
</tr>
<tr>
<td>Proc 2</td>
<td>=++</td>
<td>+</td>
<td>=++++=+</td>
</tr>
</tbody>
</table>

(send)
Summary

• Synchronizers: ensure lockstep in async

• Local, global, and hybrid
  – Good for rounds, communication, or a trade-off
  – Correctness of global synchronizers is clear
  – Local/hybrid produce equivalent executions

• Fault tolerant synchronizers impossible