CS 425 / ECE 428
Distributed Systems
Fall 2018
Indranil Gupta (Indy)
Lecture 13: Snapshots

All slides © IG
Here’s a Snapshot
• More often, each country’s representative is sitting in their respective capital, and sending messages to each other (say emails).
• How do you calculate a “global snapshot” in that distributed system?
• What does a “global snapshot” even mean?
In a cloud: each application or service is running on multiple servers

Servers handling concurrent events and interacting with each other

The ability to obtain a “global photograph” of the system is important

Some uses of having a global picture of the system

- Checkpointing: can restart distributed application on failure
- Garbage collection of objects: objects at servers that don’t have any other objects (at any servers) with pointers to them
- Deadlock detection: Useful in database transaction systems
- Termination of computation: Useful in batch computing systems like Folding@Home, SETI@Home
What’s a Global Snapshot?

• **Global Snapshot** = **Global State** =
  Individual state of each process in the distributed system
  +
  Individual state of each communication channel in the distributed system

• Capture the *instantaneous* state of each process

• And the instantaneous *state* of each communication channel, i.e., *messages* in transit on the channels
Obvious First Solution

- Synchronize clocks of all processes
- Ask all processes to record their states at known time $t$
- Problems?
  - Time synchronization always has error
    - Your bank might inform you, “We lost the state of our distributed cluster due to a 1 ms clock skew in our snapshot algorithm.”
    - Also, does not record the state of messages in the channels
- Again: synchronization not required – causality is enough!
Example
$\pi_i$ [$701$, 100 iPhones]

$\pi_j$ [$600$, 50 Androids]

$C_{ij}$ [$empty$]

$C_{ji}$ [$299$, Order Android]

[Global Snapshot 1]
Global Snapshot 2

$P_i$

$C_{ij}$

[$701, 100 iPhones$]

[$299, Order Android$]

$P_j$

$C_{ji}$

[$499, Order iPhone$]

[$101, 50 Androids$]

[Global Snapshot 2]
Global Snapshot 3

$1200, 1 iPhone order from Pj, 100 iPhones

$299, Order Android

$101, 50 Androids

[empty]
($299, Order Android),
(1 iPhone)

[$1200, 99 iPhones]
[empty]

[$101, 50 Androids]

[Global Snapshot 4]
\( P_i \) [$1200, 99 iPhones]

\( C_{ij} \) [empty]

\( P_j \) [$400, 1 Android order from \( P_i \), 50 Androids]

[Global Snapshot 5]
Global Snapshot 6

... and so on ...

$P_i$ [$1200, 99$ iPhones]

$C_{ij}$ [empty]

$P_j$ [$400, 1$ Android order from $P_i$, $50$ Androids, $1$ iPhone]

$C_{ji}$ [empty]
Moving from State to State

• Whenever an event happens anywhere in the system, the global state changes
  – Process receives message
  – Process sends message
  – Process takes a step

• State to state movement obeys causality
  – Next: Causal algorithm for Global Snapshot calculation
System Model

- **Problem:** Record a global snapshot (state for each process, and state for each channel)

- **System Model:**
  - $N$ processes in the system
  - There are two uni-directional communication channels between each ordered process pair: $P_j \to P_i$ and $P_i \to P_j$
  - Communication channels are FIFO-ordered
    - First in First out
  - No failure
  - All messages arrive intact, and are not duplicated
    - Other papers later relaxed some of these assumptions
Requirements

• **Snapshot should not interfere with normal application actions, and it should not require application to stop sending messages**

• **Each process is able to record its own state**
  – Process state: Application-defined state or, in the worst case:
  – its heap, registers, program counter, code, etc. (essentially the coredump)

• **Global state is collected in a distributed manner**

• **Any process may initiate the snapshot**
  – We’ll assume just one snapshot run for now
Chandy-Lamport Global Snapshot Algorithm

- First, Initiator $P_i$ records its own state
- Initiator process creates special messages called “Marker” messages
  - Not an application message, does not interfere with application messages
- for $j=1$ to $N$ except $i$
  - $P_i$ sends out a Marker message on outgoing channel $C_{ij}$
  - $(N-1)$ channels
  - Starts recording the incoming messages on each of the incoming channels at $P_i$: $C_{ji}$ (for $j=1$ to $N$ except $i$)
Whenever a process $P_i$ receives a Marker message on an incoming channel $C_{ki}$

- **if** (this is the first Marker $P_i$ is seeing)
  - $P_i$ records its own state first
  - Marks the state of channel $C_{ki}$ as “empty”
  - for $j=1$ to $N$ except $i$
    - $P_i$ sends out a Marker message on outgoing channel $C_{ij}$
    - Starts recording the incoming messages on each of the incoming channels at $P_i$: $C_{ji}$ (for $j=1$ to $N$ except $i$ and $k$)

- **else** // already seen a Marker message
  - Mark the state of channel $C_{ki}$ as all the messages that have arrived on it since recording was turned on for $C_{ki}$
The algorithm terminates when

- All processes have received a Marker
  - To record their own state
- All processes have received a Marker on all the \((N-1)\) incoming channels at each
  - To record the state of all channels

Then, (if needed), a central server collects all these partial state pieces to obtain the full global snapshot
Example

Time

P1 A B C D E
E F G
H I J

P2 E F G

P3 H I J

- Instruction or Step
- Message
P1 is Initiator:
- Record local state S1,
- Send out markers
- Turn on recording on channels $C_{21}, C_{31}$
- First Marker!
- Record own state as S3
- Mark $C_{13}$ state as empty
- Turn on recording on other incoming $C_{23}$
- Send out Markers

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>Time</td>
<td></td>
</tr>
</tbody>
</table>

S1, Record $C_{21}$, $C_{31}$
S1, Record C_{21}, C_{31}

- S3
- C_{13} = \langle \rangle
- Record C_{23}
S1, Record $C_{21} - C_{31}$

State of channel $C_{31} = <>$

Duplicate Marker!

- $S3$
- $C_{13} = <>$
- Record $C_{23}$
• First Marker!
• Record own state as S2
• Mark C_{32} state as empty
• Turn on recording on C_{12}
• Send out Markers

S1, Record C_{21}, C_{31}
C_{31} = <>

C_{13} = <>

Record C_{23}
S1, Record $C_{21}$, $C_{34}$

$C_{31} = <>$

- S3
- $C_{13} = <>$
- Record $C_{23}$

- S2
- $C_{32} = <>$
- Record $C_{12}$
P1       P2       P3

A      B                                  C                   D        E

E             F                          G

H                                I                                          J

S1, Record C_{21}, C_{31}

• S2
• C_{32} = <>
• C_{12} = <>

• Duplicate!
• C_{21} = <message G \rightarrow D >

• C_{31} = <>
• C_{13} = <>
• Record C_{23}

Record C_{12}
Algorithm has Terminated

- $S1 = < >$
- $C_{13} = < >$
- $C_{21} = \langle \text{message G\rightarrow D} \rangle$
- $C_{31} = < >$
- $C_{32} = < >$
- $C_{12} = < >$
- $C_{23} = < >$
Collect the Global Snapshot Pieces

S1

C_{21} = <message G→D >

C_{31} = <>

C_{13} = <>

C_{23} = <>

C_{12} = <>

C_{32} = <>

S2

S3
• Global Snapshot calculated by Chandy-Lamport algorithm is causally correct
  – What?
Cuts

- **Cut** = time frontier at each process and at each channel
- Events at the process/channel that happen before the cut are “in the cut”
  - And happening after the cut are “out of the cut”
Consistent Cuts

**Consistent Cut:** a cut that obeys causality

- A cut $C$ is a consistent cut if and only if:
  - for (each pair of events $e$, $f$ in the system)
    - Such that event $e$ is in the cut $C$, and if $f \rightarrow e$ ($f$ happens-before $e$)
      - Then: Event $f$ is also in the cut $C$
Example

Consistent Cut

Inconsistent Cut

G → D, but only D is in cut
Our Global Snapshot Example ...
... is causally correct

Consistent Cut captured by our Global Snapshot Example
In fact...

- Any run of the Chandy-Lamport Global Snapshot algorithm creates a consistent cut
Chandy-Lamport Global Snapshot algorithm creates a consistent cut

Let’s quickly look at the proof

• Let \( e_i \) and \( e_j \) be events occurring at \( P_i \) and \( P_j \), respectively such that
  
  \[ e_i \rightarrow e_j \]  
  (\( e_i \) happens before \( e_j \))

• The snapshot algorithm ensures that
  
  if \( e_j \) is in the cut then \( e_i \) is also in the cut.

• That is: if \( e_j \rightarrow <P_j \text{ records its state}> \), then
  
  – it must be true that \( e_i \rightarrow <P_i \text{ records its state}> \).
Chandy-Lamport Global Snapshot algorithm creates a consistent cut

- if \( e_j \rightarrow <P_j \text{ records its state}> \), then it must be true that \( e_i \rightarrow <P_i \text{ records its state}> \).
  - By contradiction, suppose \( e_j \rightarrow <P_j \text{ records its state}> \) and \( <P_i \text{ records its state}> \rightarrow e_i \)
  - Consider the path of app messages (through other processes) that go from \( e_i \rightarrow e_j \)
  - Due to FIFO ordering, markers on each link in above path will precede regular app messages
  - Thus, since \( <P_i \text{ records its state}> \rightarrow e_i \), it must be true that \( P_j \text{ received a marker before } e_j \)
  - Thus \( e_j \) is not in the cut \( => \) contradiction
What is the Chandy-Lamport algorithm used for?
“Correctness” in Distributed Systems

- Can be seen in two ways
- Liveness and Safety
- Often confused – it’s important to distinguish from each other
Liveness

- **Liveness** = guarantee that something *good* will happen, *eventually*
  - Eventually == does not imply a time bound, but if you let the system run long enough, then …
Liveness: Examples

- **Liveness** = guarantee that something **good** will happen, **eventually**
  - Eventually == does not imply a time bound, but if you let the system run long enough, then …

- **Examples in Real World**
  - Guarantee that “at least one of the athletes in the 100m final will win gold” is liveness
  - A criminal will eventually be jailed

- **Examples in a Distributed System**
  - Distributed computation: Guarantee that it will terminate
  - “Completeness” in failure detectors: every failure is eventually detected by some non-faulty process
  - In Consensus: All processes eventually decide on a value
Safety

• Safety = guarantee that something bad will never happen
Safety: Examples

• **Safety** = guarantee that something **bad** will **never** happen

• **Examples in Real World**
  – A peace treaty between two nations provides safety
    • War will never happen
  – An innocent person will never be jailed

• **Examples in a Distributed System**
  – There is no deadlock in a distributed transaction system
  – No object is orphaned in a distributed object system
  – “Accuracy” in failure detectors
  – In Consensus: No two processes decide on different values
Can’t we Guarantee both?

- Can be difficult to satisfy both liveness and safety in an asynchronous distributed system!
  - Failure Detector: Completeness (Liveness) and Accuracy (Safety) cannot both be guaranteed by a failure detector in an asynchronous distributed system
  - Consensus: Decisions (Liveness) and correct decisions (Safety) cannot both be guaranteed by any consensus protocol in an asynchronous distributed system
  - Very difficult for legal systems (anywhere in the world) to guarantee that all criminals are jailed (Liveness) and no innocents are jailed (Safety)
In the language of Global States

- Recall that a distributed system moves from one global state to another global state, via causal steps.
- **Liveness w.r.t. a property Pr in a given state S means**
  - S satisfies Pr, or there is some causal path of global states from S to S’ where S’ satisfies Pr.
- **Safety w.r.t. a property Pr in a given state S means**
  - S satisfies Pr, and all global states S’ reachable from S also satisfy Pr.
Using Global Snapshot Algorithm

• Chandy-Lamport algorithm can be used to detect global properties that are **stable**
  – Stable = once true, stays true forever afterwards

• **Stable Liveness examples**
  – Computation has terminated

• **Stable Non-Safety examples**
  – There is a deadlock
  – An object is orphaned (no pointers point to it)

• **All stable global properties can be detected using the Chandy-Lamport algorithm**
  • Due to its causal correctness
The ability to calculate global snapshots in a distributed system is very important.

But don’t want to interrupt running distributed application.

Chandy-Lamport algorithm calculates global snapshot.

Obeys causality (creates a consistent cut).

Can be used to detect stable global properties.

Safety vs. Liveness.
Announcements

• Midterm next Tuesday (10/16)
• Locations:
  – DCL 1320: if your last name starts with A-L
  – 1GH-100: if your last name starts with M-Z
    • 100 Gregory Hall (810 S. Wright St., Urbana)
• Material through lecture 12 (Time and Ordering)
Announcements (2)

• No lecture this Thursday 10/11
• BUT
  – View Four lecture videos on website (included in syllabus, though not midterm)
  – Solve rest of Practice Midterm