14-736: Distributed Systems

Lecture 17 * Spring 2019 * Kesden

Reminder

- *Brittle*: Failure is likely to result in a disproportionately large disability, e.g. one system of many fails or becomes disconnected, and thereby prevents a write-all quorum, disabling all writes.

- *Robust:* Failure generates a less-than-proportional disability, e.g. one server of many fails, but the others are able to maintain the full availability of the service, so the only disability is reduced robustness. Robust systems are often “fail soft”.

Techniques for Building Robust Systems

- *Hardening*: Make individual components less likely to fail.
- *Redundancy*: Make additional resources, and the capability to use them, available, thereby hiding failure.
- *Resilience*: Enable systems to recover quickly from failure
  - *Self-healing*: Automatic retirement of failed resources and reintroduction of fresh resource
  - *Visibility and Control*: Enable humans to see and respond to failure.
- *Gracefulness degradation*: Enable systems to continue to make use of available resources, even aftger some resources have failed or become disconnected.

Our Toolbox So Far

- Redundancy, replication and quorums
- Failing soft, e.g. Coda allowing temporary inconsistency
- Self-healing, e.g. coordinator elections, removing and introducing anew failed participants, etc.
- Human intervention, e.g. appointment of new coodinators, etc.

Today: Logging and Recovery

- Enables self-healing, etc.
- Already discussed in some contexts, as needed, e.g. Write-Ahead Logging (WAL) for transactions
- Today's discussion will provide a comprehensive look, including at those uses.

Checkpointing

- Making a consistent copy of all relevant state to stable storage.
  - Can be used to restore this state
- "Consistent copy" is major challenge
  - Often involves freezing things and letting tings in-flight settle
  - Disrupts availability
- Distributed systems present special challenge
  - Consistency must be global, not just per host.

Logging

- Recording of events as they occur
- Enables playback of events to restore state later
- Recording is a key challenge, because it needs to be to stable storage if it is to be guaranteed available later
- Playback has many challenges:
  - Length of time it takes for long logs
  - Impact upon other systems which receive redundant requests, e.g. peers, databases, etc.
- Insert prior discussion about Write-Ahead Logging (WAL) here.

Checkpointing + Logging

- Common idiom
- Periodically checkpoint to save complete state
- Incrementally log to maintain updates from that state
- Upon checkpointing *prune* the log to include only those events subsequent to the checkpoint
    - Reduces replay time
    - Reduces need to store large log

Still Unanswered

- How to maintain a stable log without introducing unmanageable delay?
- What about resends from recovering server playing back logs?
- What about system availability during checkpointing?
- How to maintain global consistency
- What about “in flight” communication?

Buffering Logs

- Fast, stable storage is one option for maintaining logs
  - Flash ram, etc.
  - But, it may not be fast enough
- Buffer logs into block-sized chunks, then commit to stable storage
  - Amortizes any seek cost or read-merge-write cost
  - Delays commit point
- Logs can only restore the timeline to the most recent point in time to which they contiguously exist

Fast Stable Logging?

- No less latent than available stable storage

Side-Effects of Playbacks

- Naïve playbacks of logs can repeat messages already sent, having undesirable effects on other systems
    - Other participants may act
    - Databases may repeat updates that aren't idempotent
    - Etc
- Options:
    - Squelch sending such communication
    - Use *incarnation numbers* to enable recipients to do the same.
        - Increment number with each "reboot", record in sender and receiver logs. Used to detect duplicates by receiver.

Checkpointing

- Generally involves freezing the system to maintain consistency
  - Checkpointing takes time and can’t change values mid-way
    - Maybe make “fixups”, but dependencies cascade, so not really
  - Freezing means not only local system, but others that might need it
    - Inbound messages need to be frozen
      - Maybe buffered for later
    - What about ACKs and Resends? Can be a complexity
- What about other systems? (See next slide)

Distributed Checkpointing

- Need global consistency
- Easiest solution – free all systems and checkpoint simultaneously
  - Called *Synchronous Checkpointing*
  - But, what about availability?
- More complex and higher risk solution
  - Checkpoint independently, called *Asynchronous Checkpointing*
  - Checkpoints are not at same point in time and unlikely to restore to globally consistent state
    - Patch up with logs? Are they synchronous, i.e. not buffered?

Recovery Line

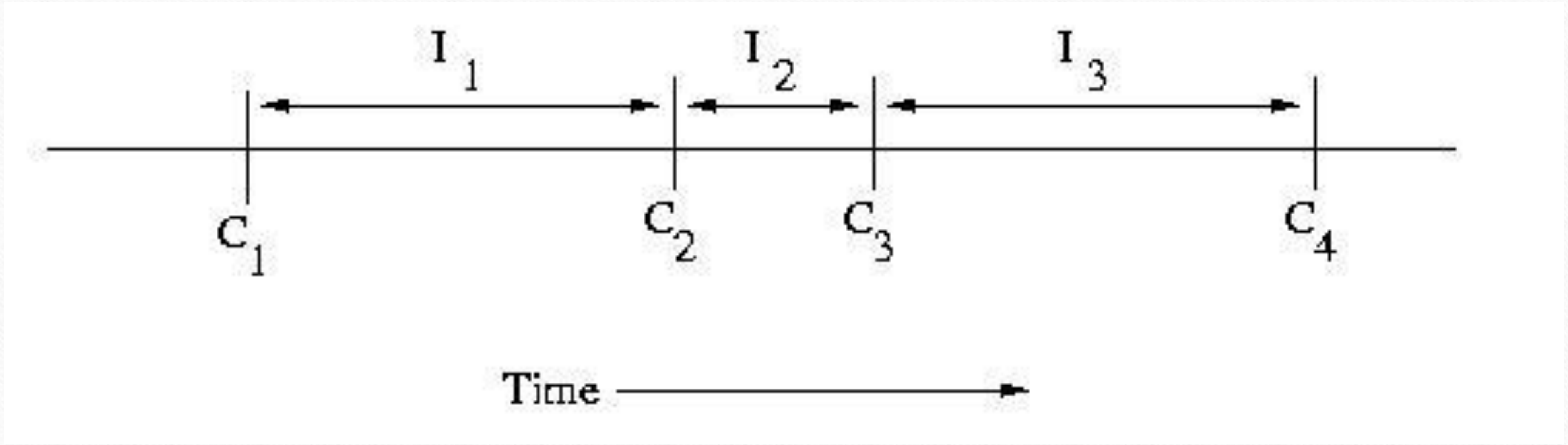
- Restoration might involve:
  - Checkpointing
  - Log playback
  - Updates from peers or a coordinator
- If one of many participants can't be brought up-to-date, it may be necessary for others to fall back to a prior point in time for global consistency. This is called a *Rollback*.
  - If other systems checkpoints are granular, rollback may *Cascade* until a *Consistent Recovery Line* can be found.

Quick Definition

- *Interval, def:*
  - The time between two sequential checkpoints of the same state
  - The time between the commitment of buffered log messages to stable storage

Uncoordinated Checkpointing

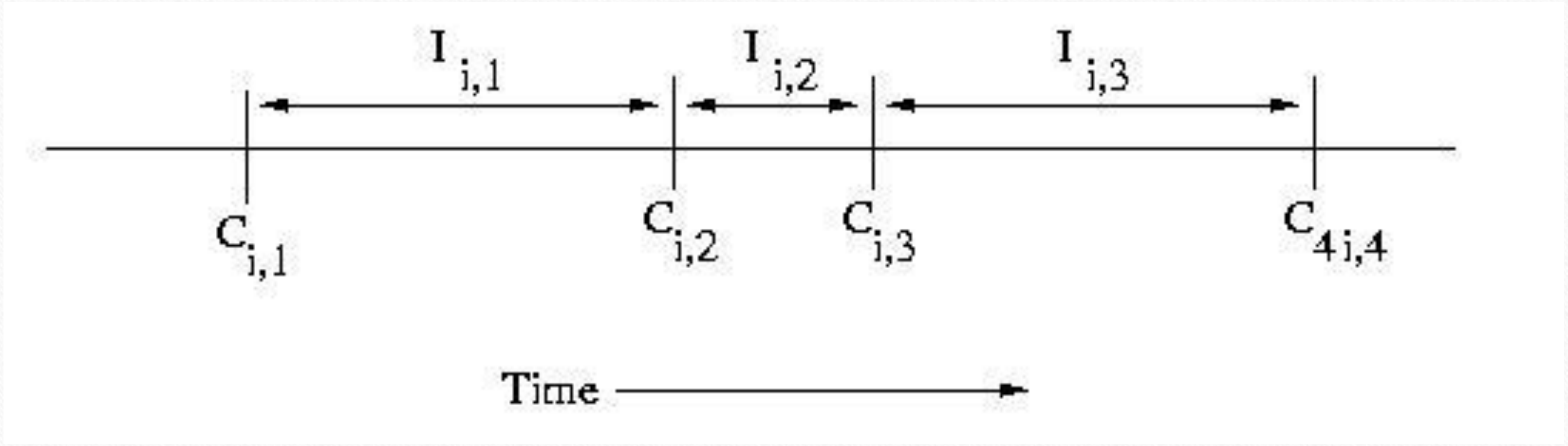
- Each related system checkpoints independently of the others.



- Note the checkpoints and intervals.

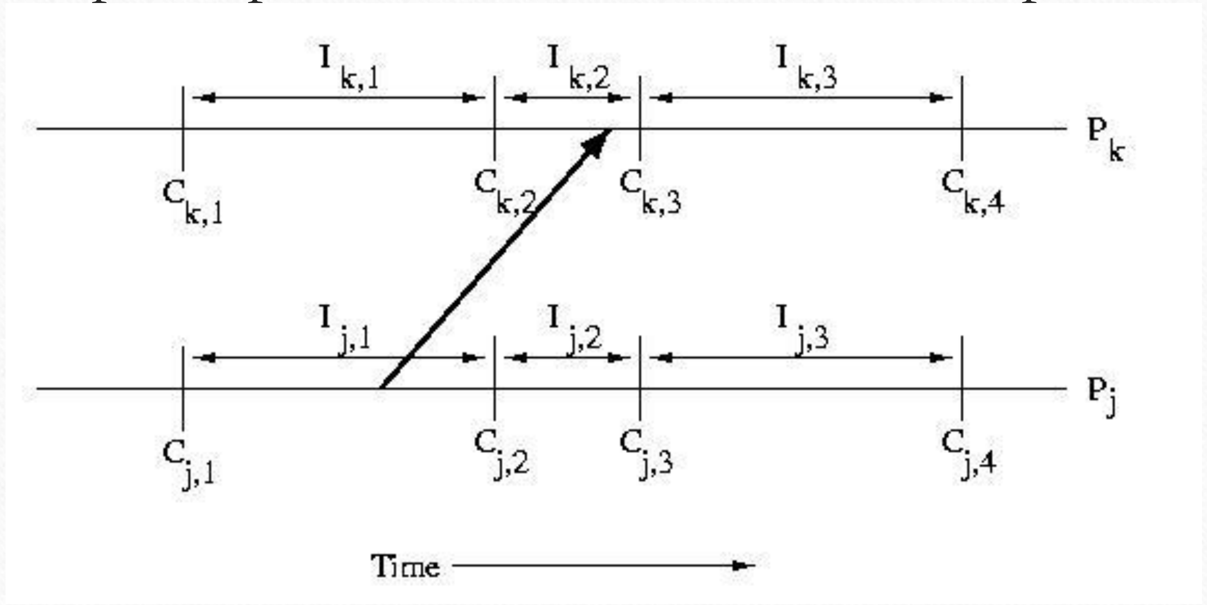
Uncoordinated Checkpointing, *cont.*

- If we have multiple participants we can use subscripts such as $C_{i,c}$ and $I_{i,c}$, where i is the participant number and c is the checkpoint sequence number.



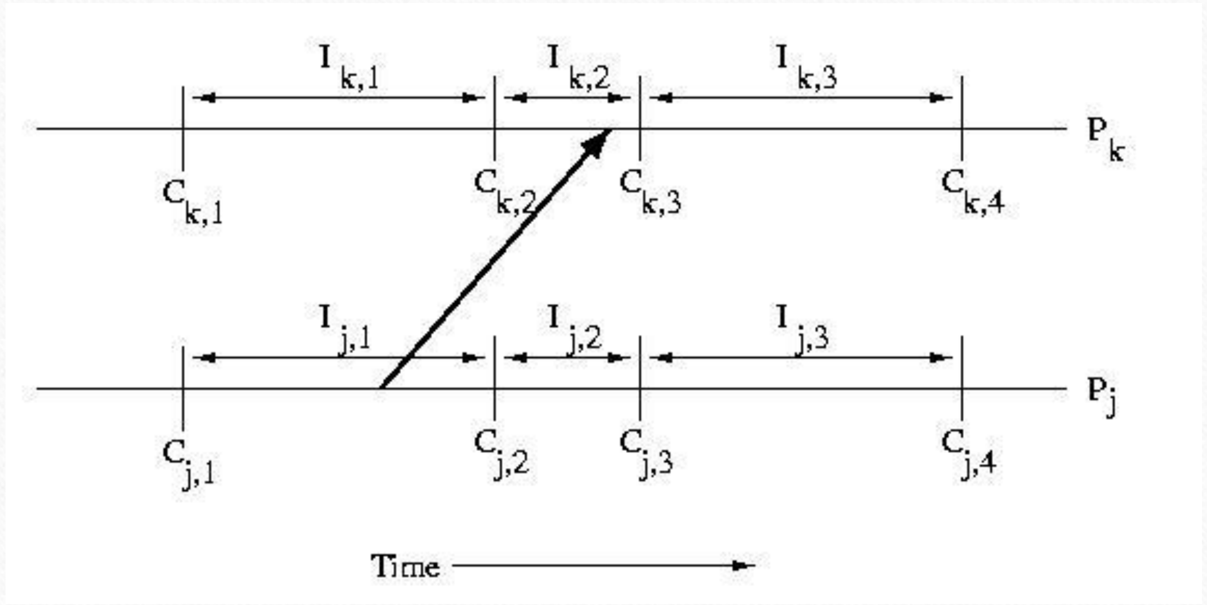
Uncoordinated Checkpointing, *cont.*

- If we have multiple participants we can use subscripts such as $C_{i,c}$ and $I_{i,c}$, where i is the participant number and c is the checkpoint sequence number.

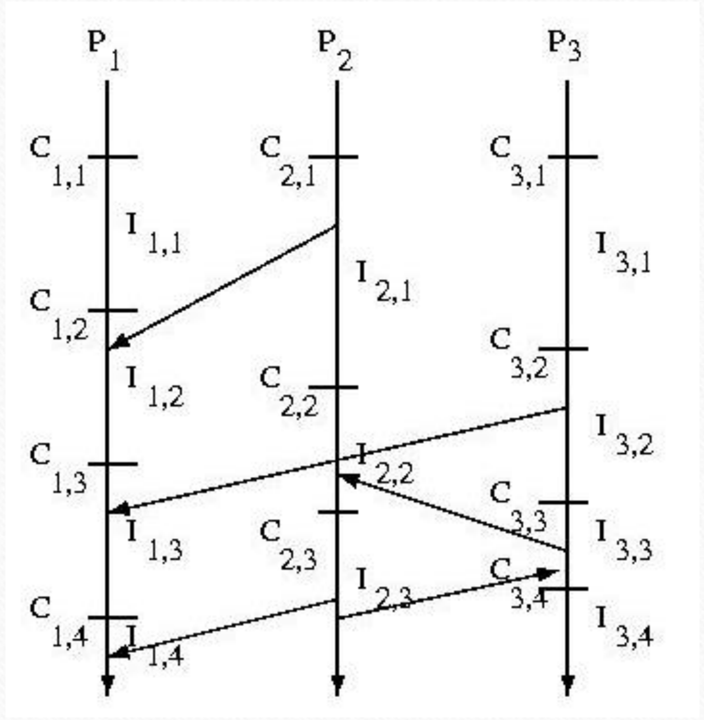


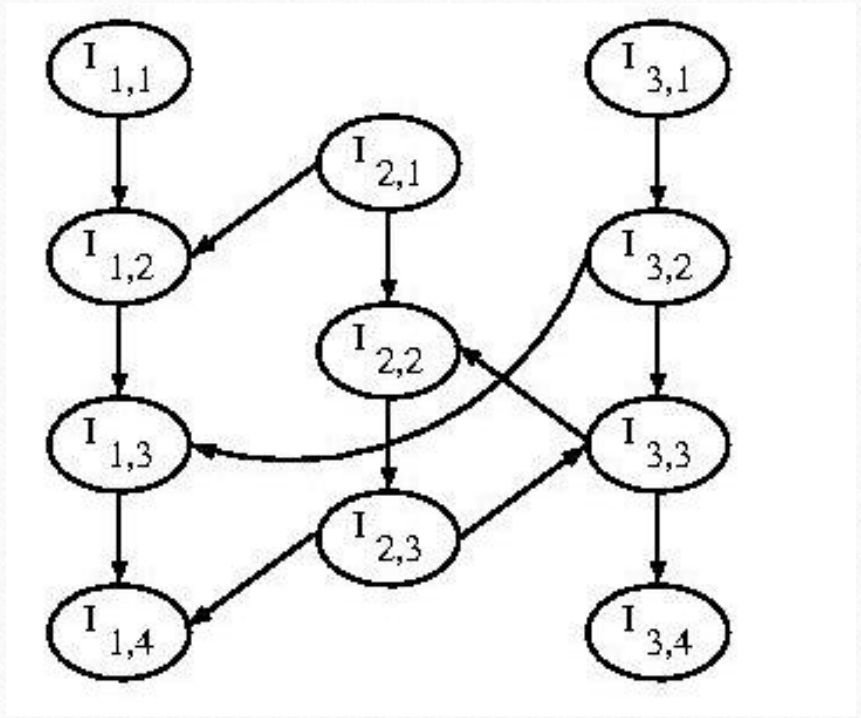
Uncoordinated Checkpointing, *cont.*

- Note that $I_{k,2}$ depends on $I_{j,1}$
- Without $I_{j,1}$ we couldn't have $I_{k,2}$



Interval Dependency Graph (IDG)

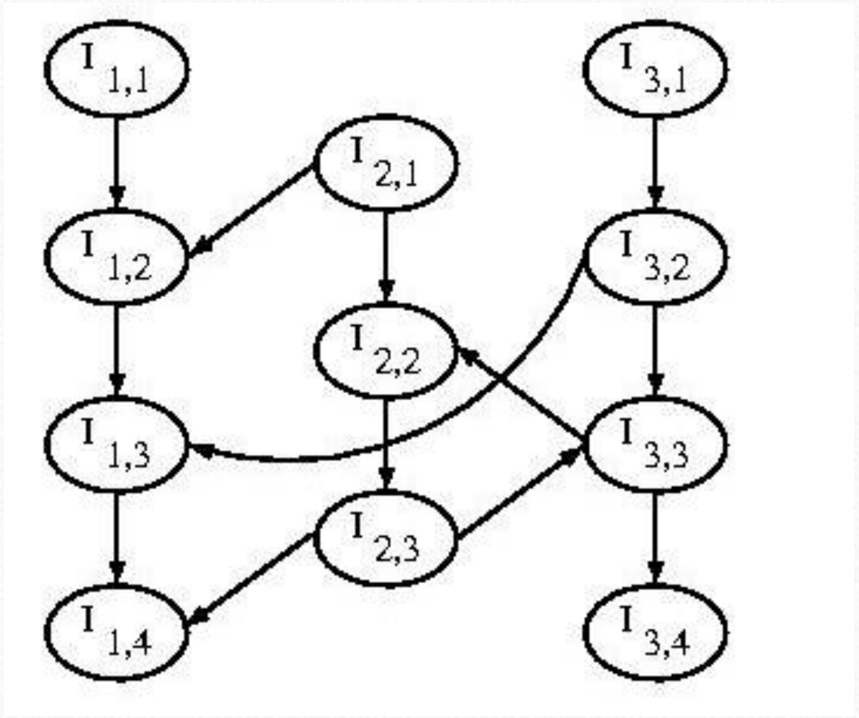




Interval Dependency Graph (IDG), *cont.*

- Two types of dependencies
  - Dependency of a system's present state upon its prior state (downward vertical edges)
  - Dependency of a system's present state upon communication it received (messages from others)
- But, wait! Don't sent messages generate a dependency? They won't get resent!
  - Well, if one cares, they need ACKs or something, which is a message in the other direction, which generates the dependency.
  - In a model where one doesn't need ACKs, well, that is a different model.

Interval Dependency Graph (IDG), *cont.*



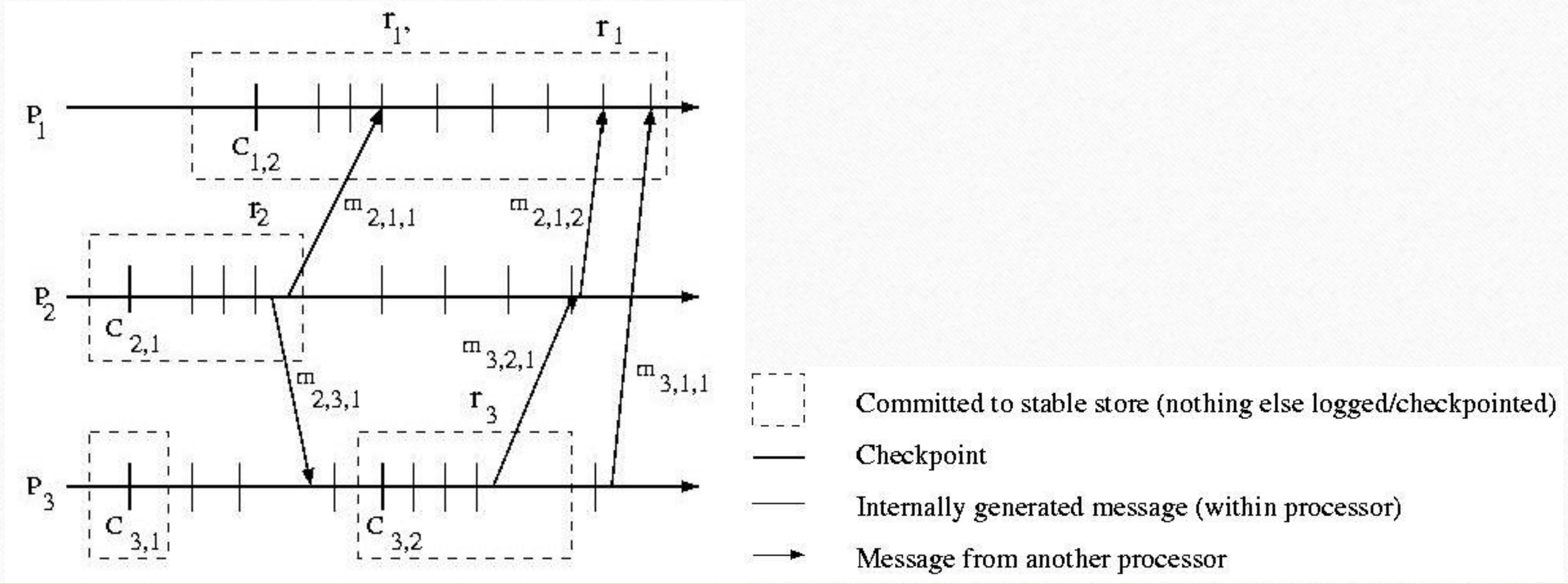
How do we find a recovery line?

- Remove the failed state
- Remove every state that depends upon it
- Wash. Rinse. Repeat.

- Can't this keep falling back?
  - Yep! That's a *cascading rollback*.

- Can't this cascade back forever?
  - Nope! The beginning of time is a checkpoint ☹

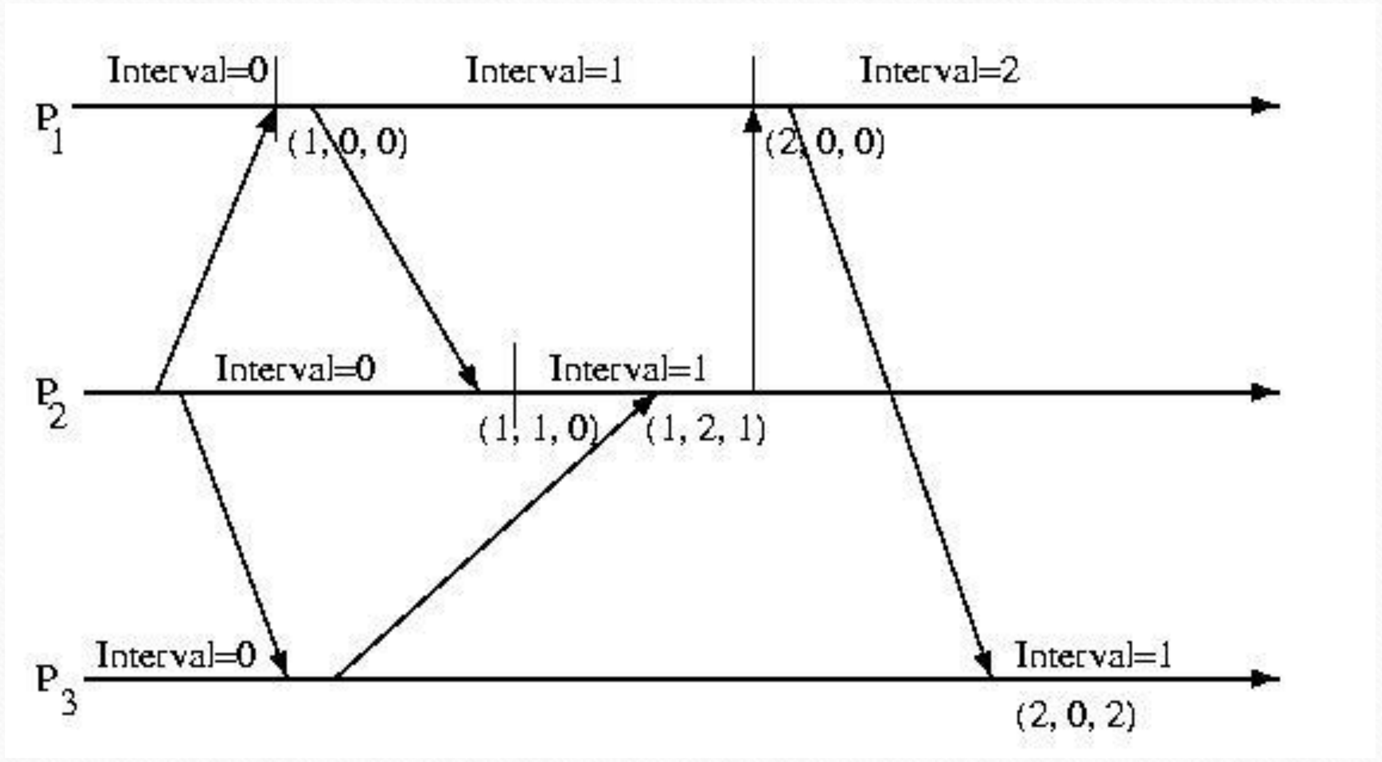
Asynchronous Logging



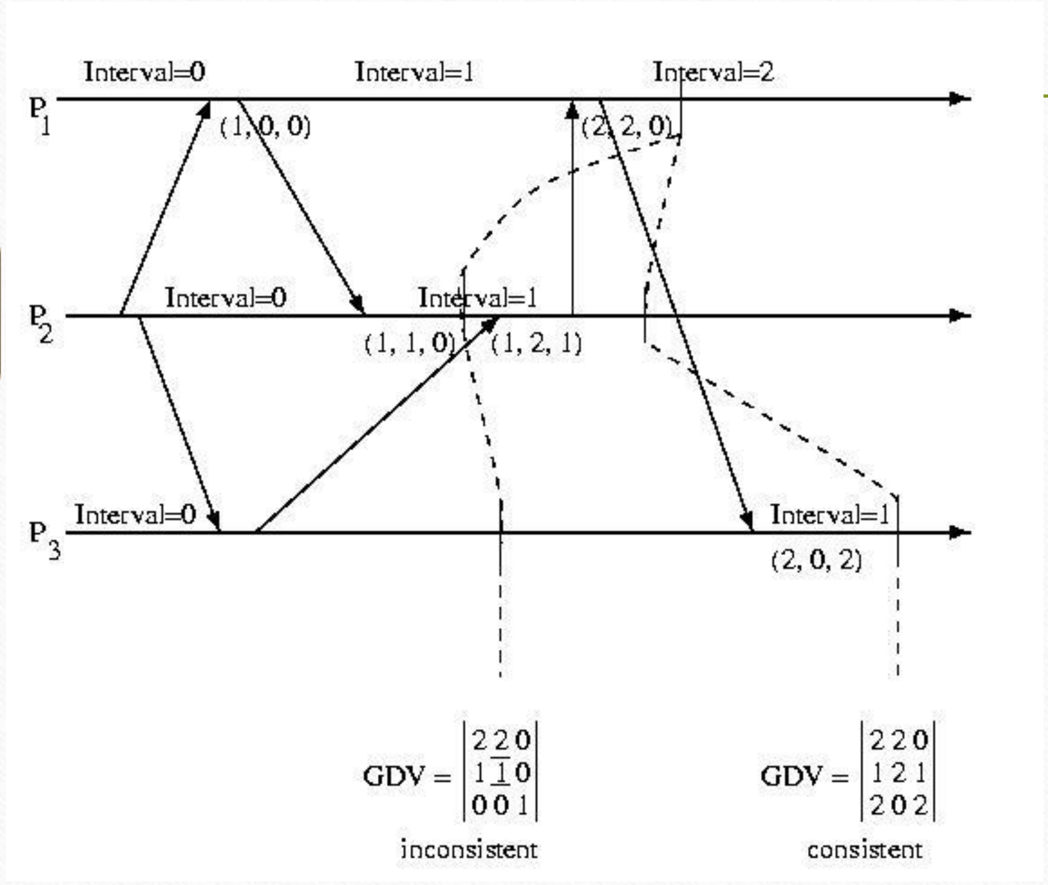
Direct Dependency Vectors (DDVs)

- Each sending participant includes its interval number with each message.
- Each receiving processor forms a DDV contains the processor's understanding of the interval on all processors.
- One processor is only informed about another processor's interval, if it directly receives a message from that processor. No gossip. Not inherited.
- A message from processor X to processor Y only contains X's interval, not the DDV present on X.

Direct Dependency Vectors, *cont.*



Global Dependency Vector (GDV)



- We can check to see if a GDV represents a consistent state, by comparing the DDVs pair-wise.
- For each entry in a consistent GDV, the interval shown for another processor must be less than or equal to the interval that that processor's DDV within the GDV contains for itself.
- GDV(p)[q] <= GDV(q)[q], 1<=p, q<=M
- In other words, no processor can have received a message from another processor originating in an interval in advance of that processor's current interval.

Where Is the GDV Stored?

- It can be stored in a distributed fashion with each host keep its own edges
- DDVs can be sent periodically or upon update to a coordinator
  - The coordinator can keep track of the recovery line
  - The coordinator can advise participants which checkpoints may be deleted, e.g. are behind the recovery line. (Or maybe the current and immediate prior recovery line, etc).
  - This can also be done in a distributed way – but requires communication to all hosts, not just one coordinator.
    - Having said that, there is then no need to separately communicate the recovery line or unneeded checkpoints, as they are locally computed.

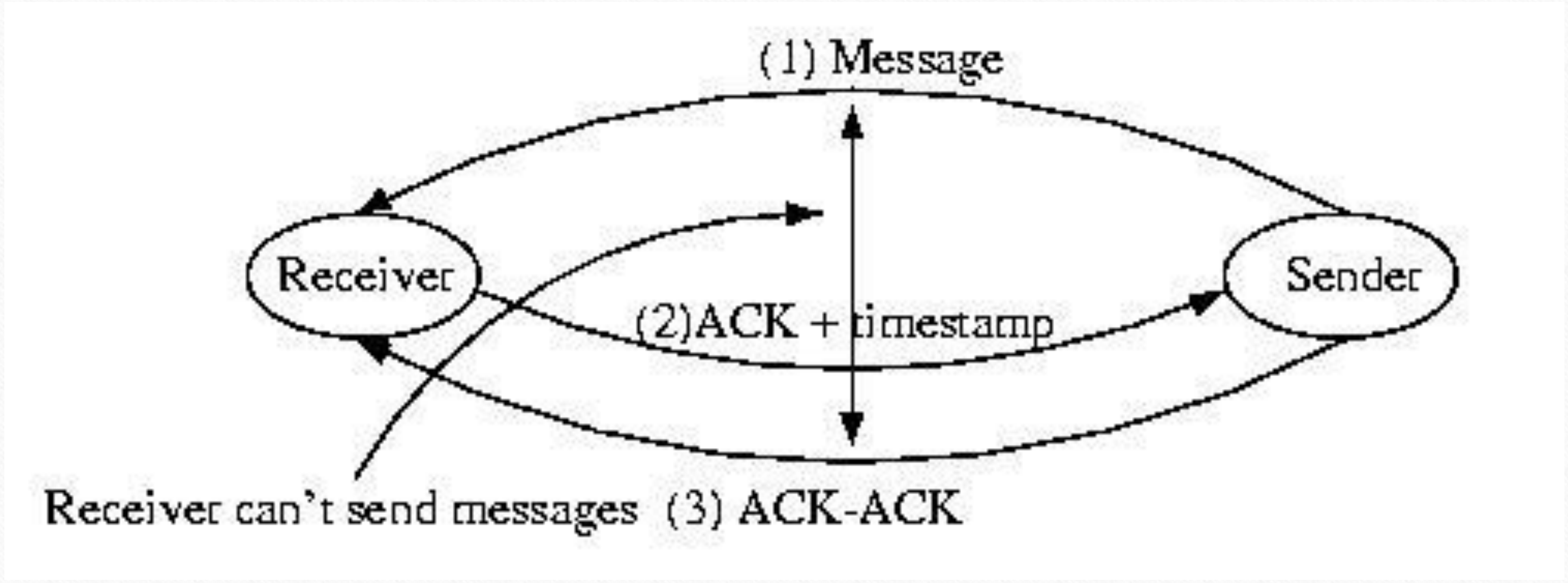
Adaptive Logging

- One might observe that it isn't necessary for a processor to log every message.
    - It only needs to log those messages that have originated from processors that have taken checkpoints more recently than it has.
    - If processors with less recent checkpoints were to fail, they would be forced to roll back prior to the message's sending, anyway.
- One optimization is then for each processor to give a sequence number to its checkpoints (as we have done before), and to keep a vector containing its best understanding of the most recent checkpoints on all other processors.
- If each time a message is sent, the current recovery line (CRL) is sent with it, a processor can determine if it is ahead of, or behind, the sender with respect to making checkpoints by comparing its checkpoint sequence number, to the sequence number of the sender in CRL received with the message.
- If, and only if, the sender is ahead, the receiver will log the message. If not, it won't worry about it.

Sender-Based Logging

- Logging normally occurs on the recipient, because only the recipient knows the order in which it has received the messages.
  - Sender only know the order in which they sent the messages, not how their receipt might have been interleaved with those from other senders on the recipient
- Unreliable or light-weight clients might not be able to maintain stable storage for checkpointing, so may need to rely upon sender logging
  - To achieve this, they can communicate the serial number back to the sender indicating the order in which they received the message. The sender can then annotate this in its log.
  - Upon playback, the recipient just buffers the messages as necessary to applies them in order, and if the buffer becomes fully, relies upon resends.

Sender-Based Logging, *cont*


(1) Message
Receiver
Sender
(2)ACK + timestamp
Receiver can't send messages
(3) ACK-ACK