Chapter 4: Global State and Snapshot Recording Algorithms

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

Introduction

- Recording the global state of a distributed system on-the-fly is an important paradigm.
- The lack of globally shared memory, global clock and unpredictable message delays in a distributed system make this problem non-trivial.
- This chapter first defines consistent global states and discusses issues to be addressed to compute consistent distributed snapshots.
- Then several algorithms to determine on-the-fly such snapshots are presented for several types of networks.

System model

- The system consists of a collection of n processes p_1, p_2, ..., p_n that are connected by channels.
- There are no globally shared memory and physical global clock and processes communicate by passing messages through communication channels.
- C_{ij} denotes the channel from process p_i to process p_j and its state is denoted by SC_{ij}.
- The actions performed by a process are modeled as three types of events: Internal events,the message send event and the message receive event.
- For a message m_{ij} that is sent by process p_i to process p_j, let $send(m_{ij})$ and $rec(m_{ij})$ denote its send and receive events.

System model

- At any instant, the state of process p_i, denoted by LS_i, is a result of the sequence of all the events executed by p_i till that instant.
- For an event e and a process state LS_i, $e \in LS_i$ iff e belongs to the sequence of events that have taken process p_i to state LS_i.
- For an event e and a process state LS_i, $e \notin LS_i$ iff e does not belong to the sequence of events that have taken process p_i to state LS_i.
- For a channel C_{ij}, the following set of messages can be defined based on the local states of the processes p_i and p_j

Transit: $transit(LS_i, LS_j) = \{ m_{ij} \mid send(m_{ij}) \in LS_i \bigwedge rec(m_{ij}) \notin LS_j \}$

Models of communication

Recall, there are three models of communication: FIFO, non-FIFO, and Co.

- In FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.
- In non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.
- A system that supports causal delivery of messages satisfies the following property: "For any two messages m_{ij} and m_{kj}, if $send(m_{ij}) \longrightarrow send(m_{kj})$, then $rec(m_{ij}) \longrightarrow rec(m_{kj})$".

Consistent global state

- The global state of a distributed system is a collection of the local states of the processes and the channels.
- Notationally, global state GS is defined as,

$$GS = \{\bigcup_i LS_i, \bigcup_{i,j} SC_{ij}\}$$

- A global state GS is a *consistent global state* iff it satisfies the following two conditions :

C1: $\text{send}(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus \text{rec}(m_{ij}) \in LS_j$. ($\oplus$ is Ex-OR operator.)

C2: $\text{send}(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge \text{rec}(m_{ij}) \notin LS_j$.

Interpretation in terms of cuts

- A cut in a space-time diagram is a line joining an arbitrary point on each process line that slices the space-time diagram into a PAST and a FUTURE.
- A consistent global state corresponds to a cut in which every message received in the PAST of the cut was sent in the PAST of that cut.
- Such a cut is known as a *consistent cut*.
- For example, consider the space-time diagram for the computation illustrated in Figure 4.1.
- Cut C1 is inconsistent because message m1 is flowing from the FUTURE to the PAST.
- Cut C2 is consistent and message m4 must be captured in the state of channel C_{21}.

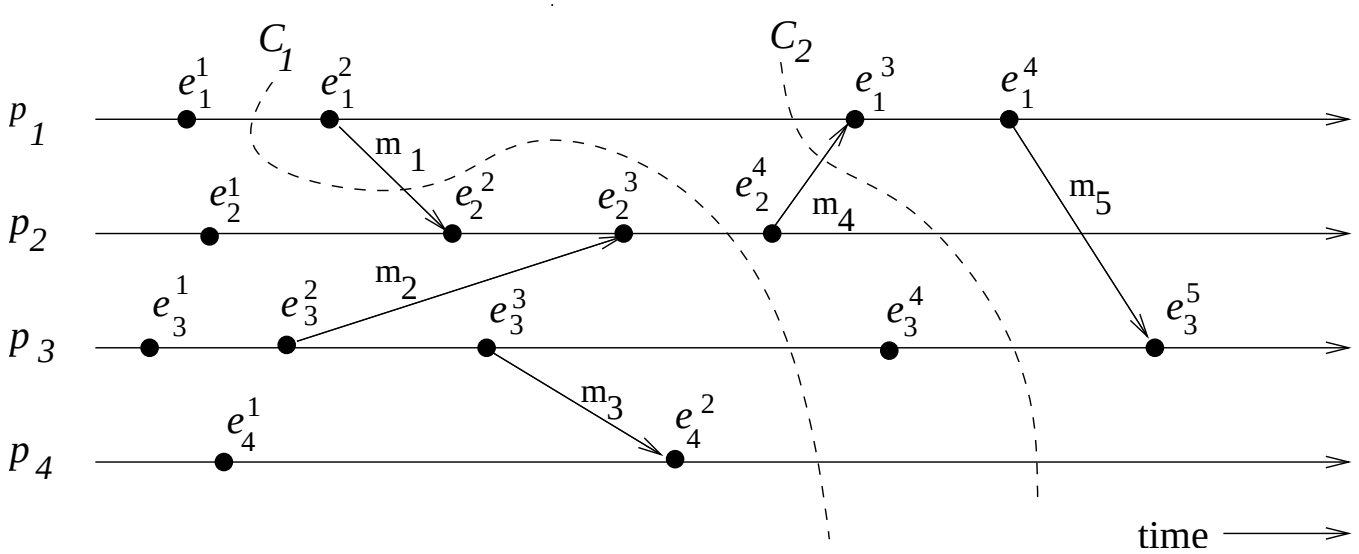

Figure 4.1: An Interpretation in Terms of a Cut.

Issues in recording a global state

The following two issues need to be addressed:

I1: How to distinguish between the messages to be recorded in the snapshot from those not to be recorded.

-Any message that is sent by a process before recording its snapshot, must be recorded in the global snapshot (from **C1**).
-Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot (from **C2**).

I2: How to determine the instant when a process takes its snapshot.

-A process p_j must record its snapshot before processing a message m_{ij} that was sent by process p_i after recording its snapshot.

Snapshot algorithms for FIFO channels

Chandy-Lamport algorithm

- The Chandy-Lamport algorithm uses a control message, called a *marker* whose role in a FIFO system is to separate messages in the channels.
- After a site has recorded its snapshot, it sends a *marker*, along all of its outgoing channels before sending out any more messages.
- A marker separates the messages in the channel into those to be included in the snapshot from those not to be recorded in the snapshot.
- A process must record its snapshot no later than when it receives a marker on any of its incoming channels.

Chandy-Lamport algorithm

- The algorithm can be initiated by any process by executing the "Marker Sending Rule" by which it records its local state and sends a marker on each outgoing channel.
- A process executes the "Marker Receiving Rule" on receiving a marker. If the process has not yet recorded its local state, it records the state of the channel on which the marker is received as empty and executes the "Marker Sending Rule" to record its local state.
- The algorithm terminates after each process has received a marker on all of its incoming channels.
- All the local snapshots get disseminated to all other processes and all the processes can determine the global state.

Chandy-Lamport algorithm

Marker Sending Rule for process i

1. Process i records its state.
2. For each outgoing channel C on which a marker has not been sent, i sends a marker along C before i sends further messages along C.

Marker Receiving Rule for process j

On receiving a marker along channel C:

if j has not recorded its state **then**

Record the state of C as the empty set

Follow the "Marker Sending Rule"

else

Record the state of C as the set of messages received along C after j's state was recorded and before j received the marker along C

Correctness and Complexity

Correctness

- Due to FIFO property of channels, it follows that no message sent after the marker on that channel is recorded in the channel state. Thus, condition **C2** is satisfied.
- When a process p_j receives message m_{ij} that precedes the marker on channel C_{ij}, it acts as follows: If process p_j has not taken its snapshot yet, then it includes m_{ij} in its recorded snapshot. Otherwise, it records m_{ij} in the state of the channel C_{ij}. Thus, condition **C1** is satisfied.

Complexity

- The recording part of a single instance of the algorithm requires $O(e)$ messages and $O(d)$ time, where e is the number of edges in the network and d is the diameter of the network.

Properties of the recorded global state

- The recorded global state may not correspond to any of the global states that occurred during the computation.
- This happens because a process can change its state asynchronously before the markers it sent are received by other sites and the other sites record their states.
  - But the system could have passed through the recorded global states in some equivalent executions.
  - The recorded global state is a valid state in an equivalent execution and if a stable property (i.e., a property that persists) holds in the system before the snapshot algorithm begins, it holds in the recorded global snapshot.
  - Therefore, a recorded global state is useful in detecting stable properties.

Spezialetti-Kearns algorithm

There are two phases in obtaining a global snapshot: locally recording the snapshot at every process and distributing the resultant global snapshot to all the initiators.

Efficient snapshot recording

- In the Spezialetti-Kearns algorithm, a markers carries the identifier of the initiator of the algorithm. Each process has a variable *master* to keep track of the initiator of the algorithm.
- A key notion used by the optimizations is that of a *region* in the system. A region encompasses all the processes whose *master* field contains the identifier of the same initiator.
- When the initiator's identifier in a marker received along a channel is different from the value in the *master* variable, the sender of the marker lies in a different region.
- The identifier of the concurrent initiator is recorded in a local variable *id-border-set*.

- The state of the channel is recorded just as in the Chandy-Lamport algorithm (including those that cross a border between regions).
- Snapshot recording at a process is complete after it has received a marker along each of its channels.
- After every process has recorded its snapshot, the system is partitioned into as many regions as the number of concurrent initiations of the algorithm.
- Variable *id-border-set* at a process contains the identifiers of the neighboring regions.

Efficient dissemination of the recorded snapshot

- In the snapshot recording phase, a forest of spanning trees is implicitly created in the system. The initiator of the algorithm is the root of a spanning tree and all processes in its region belong to its spanning tree.

Efficient dissemination of the recorded snapshot

- If p_i receives its first marker from p_j then process p_j is the parent of process p_i in the spanning tree.
- When an intermediate process in a spanning tree has received the recorded states from all its child processes and has recorded the states of all incoming channels, it forwards its locally recorded state and the locally recorded states of all its descendent processes to its parent.
- When the initiator receives the locally recorded states of all its descendents from its children processes, it assembles the snapshot for all the processes in its region and the channels incident on these processes.
- The initiator exchanges the snapshot of its region with the initiators in adjacent regions in rounds.
- The message complexity of snapshot recording is $O(e)$ irrespective of the number of concurrent initiations of the algorithm. The message complexity of assembling and disseminating the snapshot is $O(rn^2)$ where r is the number of concurrent initiations.

Snapshot algorithms for non-FIFO channels

- In a non-FIFO system, a marker cannot be used to delineate messages into those to be recorded in the global state from those not to be recorded in the global state.
- In a non-FIFO system, either some degree of inhibition or piggybacking of control information on computation messages to capture out-of-sequence messages.

Lai-Yang algorithm

The Lai-Yang algorithm fulfills this role of a marker in a non-FIFO system by using a coloring scheme on computation messages that works as follows:

1. Every process is initially white and turns red while taking a snapshot. The equivalent of the "Marker Sending Rule" is executed when a process turns red.
2. Every message sent by a white (red) process is colored white (red).
3. Thus, a white (red) message is a message that was sent before (after) the sender of that message recorded its local snapshot.
4. Every white process takes its snapshot at its convenience, but no later than the instant it receives a red message.

Lai-Yang algorithm

4. Every white process records a history of all white messages sent or received by it along each channel.
5. When a process turns red, it sends these histories along with its snapshot to the initiator process that collects the global snapshot.
6. The initiator process evaluates $transit(LS_i, LS_j)$ to compute the state of a channel C_{ij} as given below:
   SC_{ij} = white messages sent by p_i on C_{ij} − white messages received by p_j on C_{ij}
   $= \{send(m_{ij}) | send(m_{ij}) \in LS_i \} - \{rec(m_{ij}) | rec(m_{ij}) \in LS_j \}$.

Mattern's algorithm

Mattern's algorithm is based on vector clocks and assumes a single initiator process and works as follows:

1. The initiator "ticks" its local clock and selects a future vector time s at which it would like a global snapshot to be recorded. It then broadcasts this time s and freezes all activity until it receives all acknowledgements of the receipt of this broadcast.
2. When a process receives the broadcast, it remembers the value s and returns an acknowledgement to the initiator.
3. After having received an acknowledgement from every process, the initiator increases its vector clock to s and broadcasts a dummy message to all processes.
4. The receipt of this dummy message forces each recipient to increase its clock to a value $\geq s$ if not already $\geq s$.
5. Each process takes a local snapshot and sends it to the initiator when (just before) its clock increases from a value less than s to a value $\geq s$.
6. The state of C_{ij} is all messages sent along C_{ij}, whose timestamp is smaller than s and which are received by p_j after recording LS_j.

Mattern's algorithm

- A termination detection scheme for non-FIFO channels is required to detect that no white messages are in transit.
- One of the following schemes can be used for termination detection:

First method:

- Each process i keeps a counter $cntr_i$ that indicates the difference between the number of white messages it has sent and received before recording its snapshot.
- It reports this value to the initiator process along with its snapshot and forwards all white messages, it receives henceforth, to the initiator.
- Snapshot collection terminates when the initiator has received $\sum_i cntr_i$ number of forwarded white messages.

Mattern's algorithm

Second method:

- Each red message sent by a process carries a piggybacked value of the number of white messages sent on that channel before the local state recording.
- Each process keeps a counter for the number of white messages received on each channel.
- A process can detect termination of recording the states of incoming channels when it receives as many white messages on each channel as the value piggybacked on red messages received on that channel.

Snapshots in a causal delivery system

- The causal message delivery property **CO** provides a built-in message synchronization to control and computation messages.
- Two global snapshot recording algorithms, namely, Acharya-Badrinath and Alagar-Venkatesan exist that assume that the underlying system supports causal message delivery.
- In both these algorithms recording of process state is identical and proceed as follows :
- An initiator process broadcasts a token, denoted as *token*, to every process including itself.
- Let the copy of the token received by process p_i be denoted $token_i$.
- A process p_i records its local snapshot LS_i when it receives $token_i$ and sends the recorded snapshot to the initiator.
- The algorithm terminates when the initiator receives the snapshot recorded by each process.

Snapshots in a causal delivery system

Correctness

For any two processes p_i and p_j, the following property is satisfied:

$$send(m_{ij}) \notin LS_i \Rightarrow rec(m_{ij}) \notin LS_j.$$

- This is due to the causal ordering property of the underlying system as explained next.
  - Let a message m_{ij} be such that $rec(token_i) \longrightarrow send(m_{ij})$.
  - Then $send(token_j) \longrightarrow send(m_{ij})$ and the underlying causal ordering property ensures that $rec(token_j)$, at which instant process p_j records LS_j, happens before $rec(m_{ij})$.
  - Thus, m_{ij} whose send is not recorded in LS_i, is not recorded as received in LS_j.
- Channel state recording is different in these two algorithms and is discussed next.

Channel state recording in Acharya-Badrinath algorithm

- Each process p_i maintains arrays $SENT_i[1, ...N]$ and $RECD_i[1, ..., N]$.
- $SENT_i[j]$ is the number of messages sent by process p_i to process p_j.
- $RECD_i[j]$ is the number of messages received by process p_i from process p_j.
- Channel states are recorded as follows:
  When a process p_i records its local snapshot LS_i on the receipt of $token_i$, it includes arrays $RECD_i$ and $SENT_i$ in its local state before sending the snapshot to the initiator.

Channel state recording in Acharya-Badrinath algorithm

When the algorithm terminates, the initiator determines the state of channels as follows:

- The state of each channel from the initiator to each process is empty.
- The state of channel from process p_i to process p_j is the set of messages whose sequence numbers are given by $\{RECD_j[i] + 1, \ldots, SENT_i[j]\}$.

Complexity:

- This algorithm requires $2n$ messages and 2 time units for recording and assembling the snapshot, where one time unit is required for the delivery of a message.
- If the contents of messages in channels state are required, the algorithm requires $2n$ messages and 2 time units additionally.

Channel state recording in Alagar-Venkatesan algorithm

- A message is referred to as *old* if the send of the message causally precedes the send of the token.
- Otherwise, the message is referred to as *new*.

In Alagar-Venkatesan algorithm channel states are recorded as follows:

1. When a process receives the *token*, it takes its snapshot, initializes the state of all channels to empty, and returns *Done* message to the initiator. Now onwards, a process includes a message received on a channel in the channel state only if it is an old message.
2. After the initiator has received *Done* message from all processes, it broadcasts a *Terminate* message.
3. A process stops the snapshot algorithm after receiving a *Terminate* message.

Algorithms	Features
Chandy-Lamport	Baseline algorithm. Requires FIFO channels. $O(e)$ messages to record snapshot and $O(d)$ time.
Spezialetti-Kearns	supports concurrent initiators, efficient assembly and distribution of a snapshot. Assumes bidirectional channels. $O(e)$ messages to record, $O(rn^2)$ messages to assemble and distribute snapshot.
Lai-Yang	Works for non-FIFO channels. Markers piggybacked on computation messages. Message history required to compute channel states.
Li et al.	Small message history needed as channel states are computed incrementally.
Mattern	No message history required. Termination detection required to compute channel states.
Acharya-Badrinath	Requires causal delivery support, Centralized computation of channel states, Channel message contents need not be known. Requires 2n messages, 2 time units.
Alagar-Venkatesan	Requires causal delivery support, Distributed computation of channel states. Requires $3n$ messages, 3 time units, small messages.

n = # processes, u = # edges on which messages were sent after previous snapshot, e = # channels, d is the diameter of the network, r = # concurrent initiators.

Necessary and sufficient conditions for consistent global snapshots

- Many applications require that local process states are periodically recorded and analyzed during execution or post martem.
- A saved intermediate state of a process during its execution is called a *local checkpoint* of the process.
- A consistent snapshot consists of a set of local states that occurred concurrently or had a potential to occur simultaneously.
- Processes take checkpoints asynchronously. The $i^{th} (i \geq 0)$ checkpoint of process p_p is assigned the sequence number i and is denoted by $C_{p,i}$.
- We assume that each process takes an initial checkpoint before execution begins and takes a *virtual* checkpoint after execution ends.
- The i^{th} *checkpoint interval* of process p_p consists of all the computation performed between its $(i-1)^{th}$ and i^{th} checkpoints (and includes the $(i-1)^{th}$ checkpoint but not i^{th}).

Even if two local checkpoints do not have a causal path between them, they may not belong to the same consistent global snapshot.

An Example:
Consider the execution shown in the Figure 4.2.

- Although neither of the checkpoints $C_{1,1}$ and $C_{3,2}$ happened before the other, they cannot be grouped together with a checkpoint on process p_2 to form a consistent global snapshot.

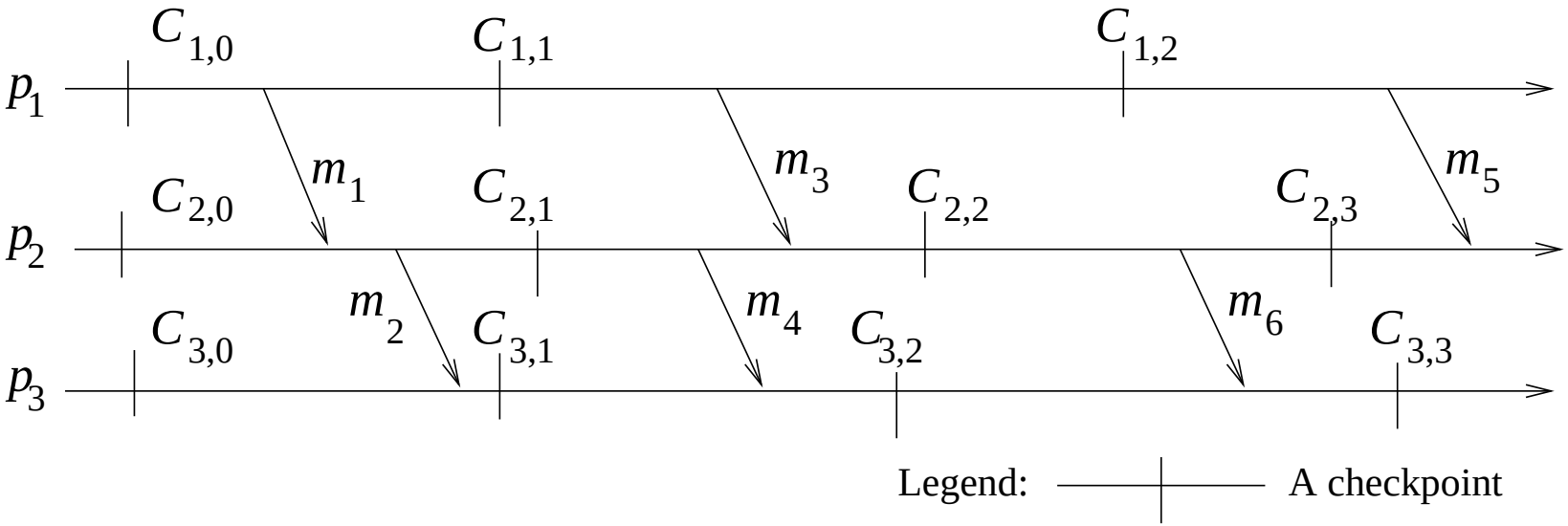

Figure 4.2: An Illustration of zigzag paths.

Necessary and sufficient conditions for consistent global snapshots

- To describe the necessary and sufficient conditions for a consistent snapshot, Netzer and Xu defined a generalization of the Lamport's happen before relation, called a zigzag path.
- A zigzag path between two checkpoints is a causal path, however, a zigzag path allows a message to be sent before the previous one in the path is received.
- In the Figure 4.2 although a causal path does not exist from $C_{1,1}$ to $C_{3,2}$, a zigzag path does exist from $C_{1,1}$ to $C_{3,2}$.
- This zigzag path means that no consistent snapshot exists in this execution that contains both $C_{1,1}$ and $C_{3,2}$.

Zigzag paths and consistent global snapshots

Definition

A *zigzag path* exists from a checkpoint $C_{x,i}$ to a checkpoint $C_{y,j}$ iff there exists messages m_1, m_2, ...m_n (n$\geq$1) such that

1. m_1 is sent by process p_x after $C_{x,i}$.
2. If m_k (1$\leq$k$\leq$n) is received by process p_z, then m_{k+1} is sent by p_z in the same or a later checkpoint interval (although m_{k+1} may be sent before or after m_k is received), and
3. m_n is received by process p_y before $C_{y,j}$.

Example:
In the Figure 4.2 a zigzag path exists from $C_{1,1}$ to $C_{3,2}$ due to messages m_3 and m_4.

Zigzag cycle

Definition

A checkpoint C is involved in a *zigzag cycle* iff there is a zigzag path from C to itself.

In Figure 4.3, $C_{2,1}$ is on a zigzag cycle formed by messages m_1 and m_2.

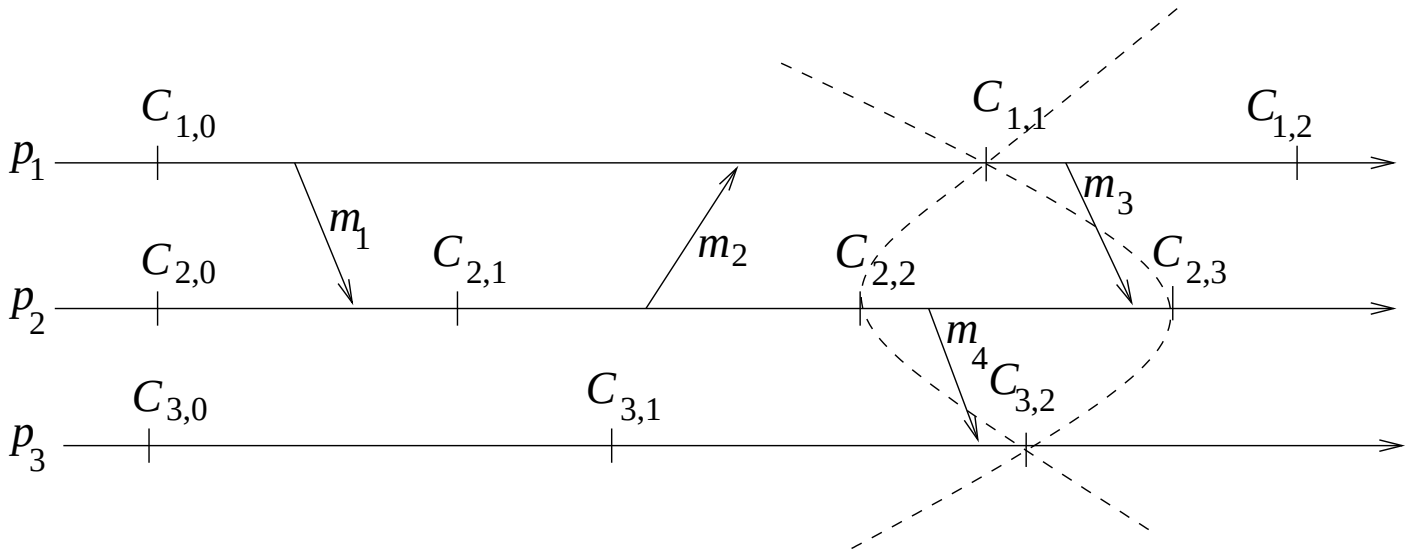

Figure 4.3: A zigzag cycle, inconsistent snapshot, and consistent snapshot.

Difference between a zigzag path and a causal path

- A causal path exists from a checkpoint A to another checkpoint B iff there is chain of messages starting after A and ending before B such that each message is sent after the previous one in the chain is received.
- A zigzag path consists of such a message chain, however, a message in the chain can be sent before the previous one in the chain is received, as long as the send and receive are in the same checkpoint interval.
- Thus a causal path is always a zigzag path, but a zigzag path need not be a causal path.
- Another difference between a zigzag path and a causal path is that a zigzag path can form a cycle but a causal path never forms a cycle.

Consistent global snapshots

Netzer and Xu proved that if no zigzag path (or cycle) exists between any two checkpoints from a set S of checkpoints, then a consistent snapshot can be formed that includes the set S of checkpoints and vice versa.

- The absence of a causal path between checkpoints in a snapshot corresponds to the necessary condition for a consistent snapshot, and the absence of a zigzag path between checkpoints in a snapshot corresponds to the necessary and sufficient conditions for a consistent snapshot.
- A set of checkpoints S can be extended to a consistent snapshot if and only if no checkpoint in S has a zigzag path to any other checkpoint in S.
- A checkpoint can be a part of a consistent snapshot if and only if it is not invloved in a Z-cycle.

Finding consistent global snapshots in a distributed computation

- Discuss how individual local checkpoints can be combined with those from other processes to form global snapshots that are consistent.
- Manivannan-Netzer-Singhal analyzed the set of *all* consistent snapshots that can be built from a set of checkpoints S.

Definition

Let A, B be individual checkpoints and R, S be sets of checkpoints. Let $\rightsquigarrow$ be a relation defined over checkpoints and sets of checkpoints such that

1. $A \rightsquigarrow B$ iff a Z-path exists from A to B,
2. $A \rightsquigarrow S$ iff a Z-path exists from A to *some* member of S,
3. $S \rightsquigarrow A$ iff a Z-path exists from *some* member of S to A, and
4. $R \rightsquigarrow S$ iff a Z-path exists from *some* member of R to *some* member of S.

$S \not\rightsquigarrow S$ defines that no Z-path (including Z-cycle) exists from any member of S to any other member of S and implies that checkpoints in S are all from different processes.

In this notations, the results of Netzer and Xu can be expressed as follows:

Theorem 4.1:

A set of checkpoints S can be extended to a consistent global snapshot if and only if $S \not\rightsquigarrow S$.

Corollary 4.1

A checkpoint C can be part of a consistent global snapshot if and only if it is not involved in a Z-cycle.

Corollary 4.2

A set of checkpoints S is a consistent global snapshot if and only if $S \not\rightsquigarrow S$ and $|S| = N$, where N is the number of processes.

Finding consistent global snapshots

Given a set S of checkpoints such that $S \not\rightsquigarrow S$, we first discuss what checkpoints from other processes can be combined with S to build a consistent global snapshot.

First Observation:

- None of the checkpoints that have a Z-path to or from any of the checkpoints in S can be used.
- Only those checkpoints that have no Z-paths to or from any of the checkpoints in S are candidates for inclusion in the consistent snapshot.
- We call the set of all such candidates the *Z-cone* of S and all checkpoints that have no causal path to or from any checkpoint in S the *C-cone* of S.
- A causal path is always Z-path, the Z-cone of S is a subset of the C-cone of S for an arbitrary S as shown in the Figure 4.4

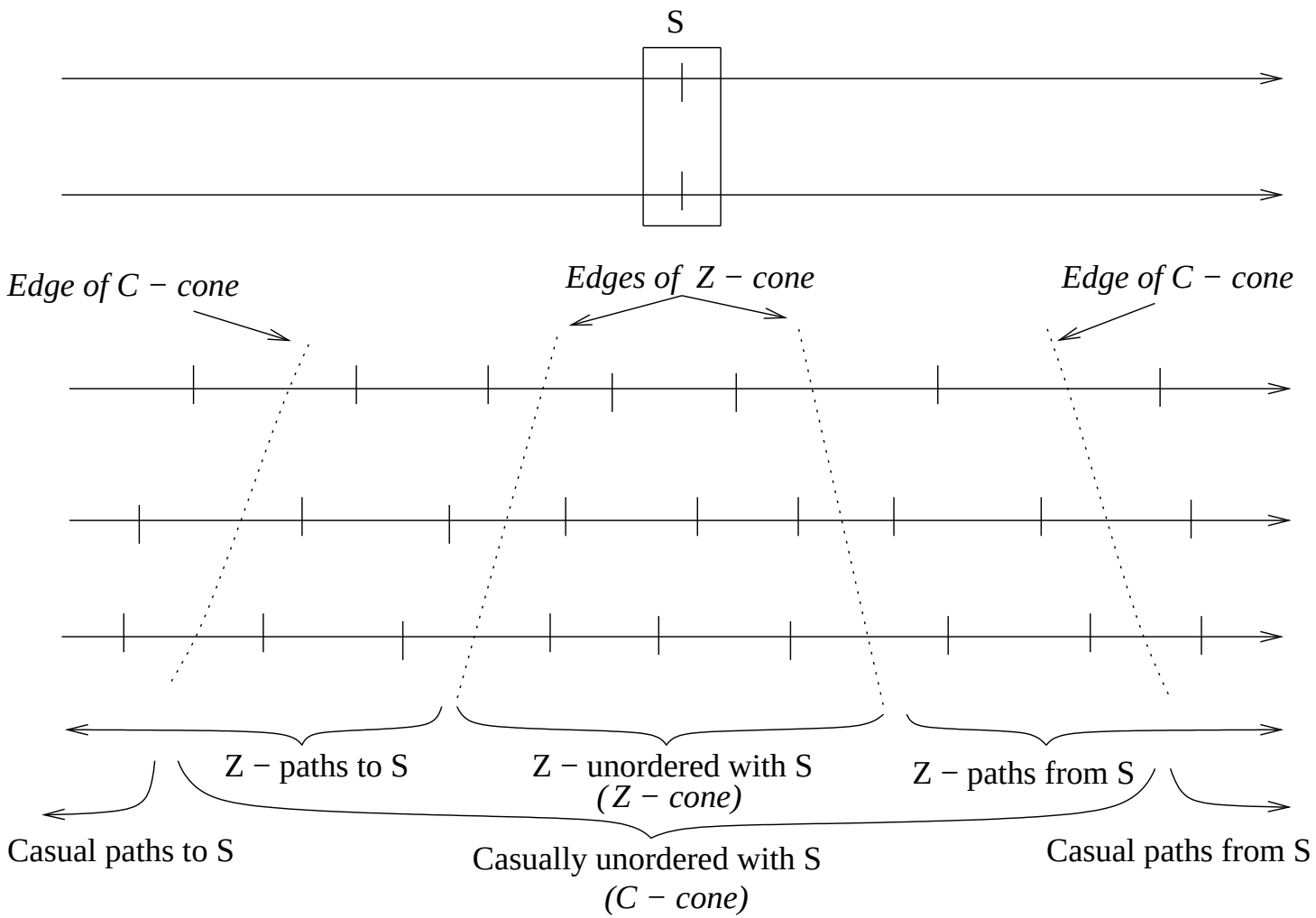

Figure 4.4: The *Z-cone* and the *C-cone* associated with a set of checkpoints S.

Second Observation:
Although candidates for building a consistent snapshot from S must lie in the Z-cone of S, not all checkpoints in the Z-cone can form a consistent snapshot with S. If a checkpoint in the Z-cone is involved in a Z-cycle, then it cannot be part of a consistent snapshot.

Definition

Let S be a set of checkpoints such that $S \not\rightsquigarrow S$. Then, for each process p_q, the set S^q_{useful} is defined as

$$S^q_{useful} = \{C_{q,i} \mid (S \not\rightsquigarrow C_{q,i}) \wedge (C_{q,i} \not\rightsquigarrow S) \wedge (C_{q,i} \not\rightsquigarrow C_{q,i})\}.$$

In addition, we define

$$S_{useful} = \bigcup_q S^q_{useful}.$$

Lemma 4.1

"Let S be a set of checkpoints such that $S \not\rightsquigarrow S$. Let $C_{q,i}$ be any checkpoint of process p_q such that $C_{q,i} \notin S$. Then $S \cup \{C_{q,i}\}$ can be extended to a consistent snapshot if and only if $C_{q,i} \in S_{useful}$."

Lemma 4.1 states that if we are given a set S such that $S \not\rightsquigarrow S$, we are guaranteed that any *single* checkpoint from S_{useful} can belong to a consistent global snapshot that also contains S.

Third observation

- Although none of the checkpoints in S_{useful} has a Z-path to or from any checkpoint in S, Z-paths may exist between members of S_{useful}.
- One final constraint is placed on the set T we choose from S_{useful} to build a consistent snapshot from S: checkpoints in T must have no Z-paths between them. Furthermore, since $S \not\rightsquigarrow S$, from Theorem 4.1 at least one such T must exist.

Theorem 4.2:
Let S be a set of checkpoints such that $S \not\rightsquigarrow S$ and let T be any set of checkpoints such that $S \cap T = \emptyset$. Then, $S \cup T$ is a consistent global snapshot if and only if

1. $T \subseteq S_{useful}$,
2. $T \not\rightsquigarrow T$, and
3. $|S \cup T| = N$.

Manivannan-Netzer-Singhal algorithm for enumerating consistent snapshots

An algorithm due to Manivannan-Netzer-Singhal that explicitly computes all consistent snapshots that include a given set a set of checkpoints S.

```
1:    ComputeAllCgs(S) {
2:        let G = ∅
3:        if S ↛ S then
4:            let AllProcs be the set of all processes not represented in S
5:            ComputeAllCgsFrom(S, AllProcs)
6:        return G
7:    }
8:    ComputeAllCgsFrom(T, ProcSet) {
9:        if (ProcSet = ∅) then
10:           G = G ∪ {T}
11:       else
12:           let p_q be any process in ProcSet
13:           for each checkpoint C ∈ T^q_useful do
14:               ComputeAllCgsFrom(T ∪ {C}, ProcSet \ {p_q})
15:   }
```

Manivannan-Netzer-Singhal algorithm

The algorithm restricts its selection of checkpoints to those within the Z-cone of S and it checks for the presence of Z-cycles within the Z-cone.

Correctness:

The following theorem argues the correctness of the algorithm.

Theorem 4.3:

Let S be a set of checkpoints and G be the set returned by $ComputeAllCgs(S)$. If $S \not\rightsquigarrow S$, then $T \in G$ if and only if T is a consistent snapshot containing S. That is, G contains exactly the consistent snapshots that contain S.

Finding Z-paths in a distributed computation

Discuss a method for determining the existence of Z-paths between checkpoints in a distributed computation that has terminated or has stopped execution, using the rollback-dependency graph (R-graph).

Definition

The rollback-dependency graph of a distributed computation is a directed graph $G = (V, E)$, where the vertices V are the checkpoints of the distributed computation and an edge $(C_{p,i}, C_{q,j})$ from checkpoint $C_{p,i}$ to checkpoint $C_{q,j}$ belongs to E if

1. $p = q$ and $j = i + 1$, or
2. $p \neq q$ and a message m sent from the i^{th} checkpoint interval of p_p is received by p_q in its j^{th} checkpoint interval $(i, j > 0)$.

Example of an R-graph

- Figure 4.6 shows shows the R-graph of the computation shown in Figure 4.5.
- In Figure 4.6, $C_{1,3}$, $C_{2,3}$, and $C_{3,3}$ represent the volatile checkpoints, the checkpoints representing the last state the process attained before terminating.
- We denote the fact that there is a path from C to D in the R-graph by $C \overset{\text{rd}}{\rightsquigarrow} D$. It only denotes the existence of a path; it does not specify any particular path.

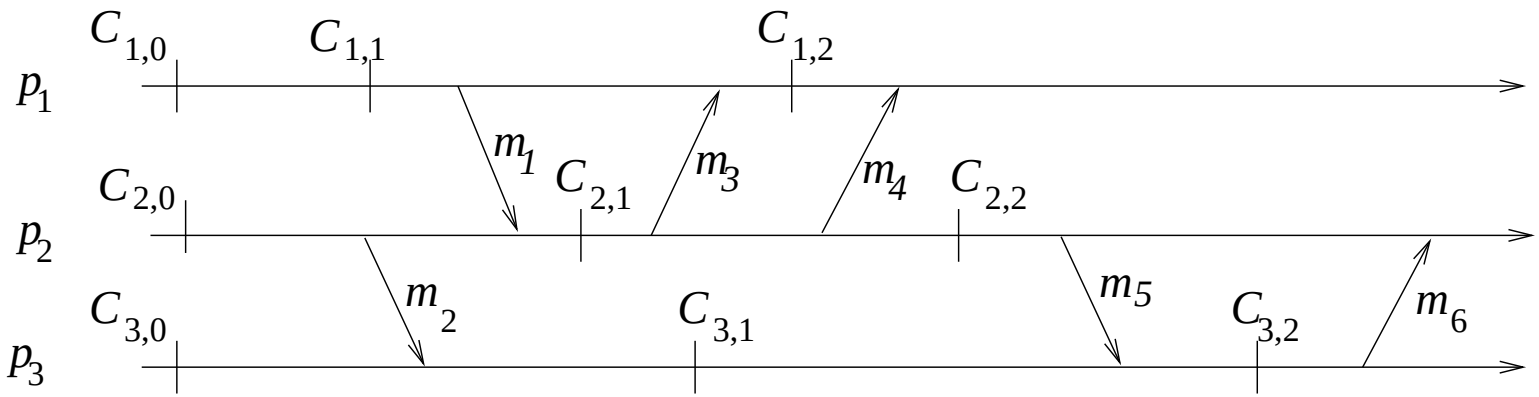

Figure 4.5: A distributed computation.

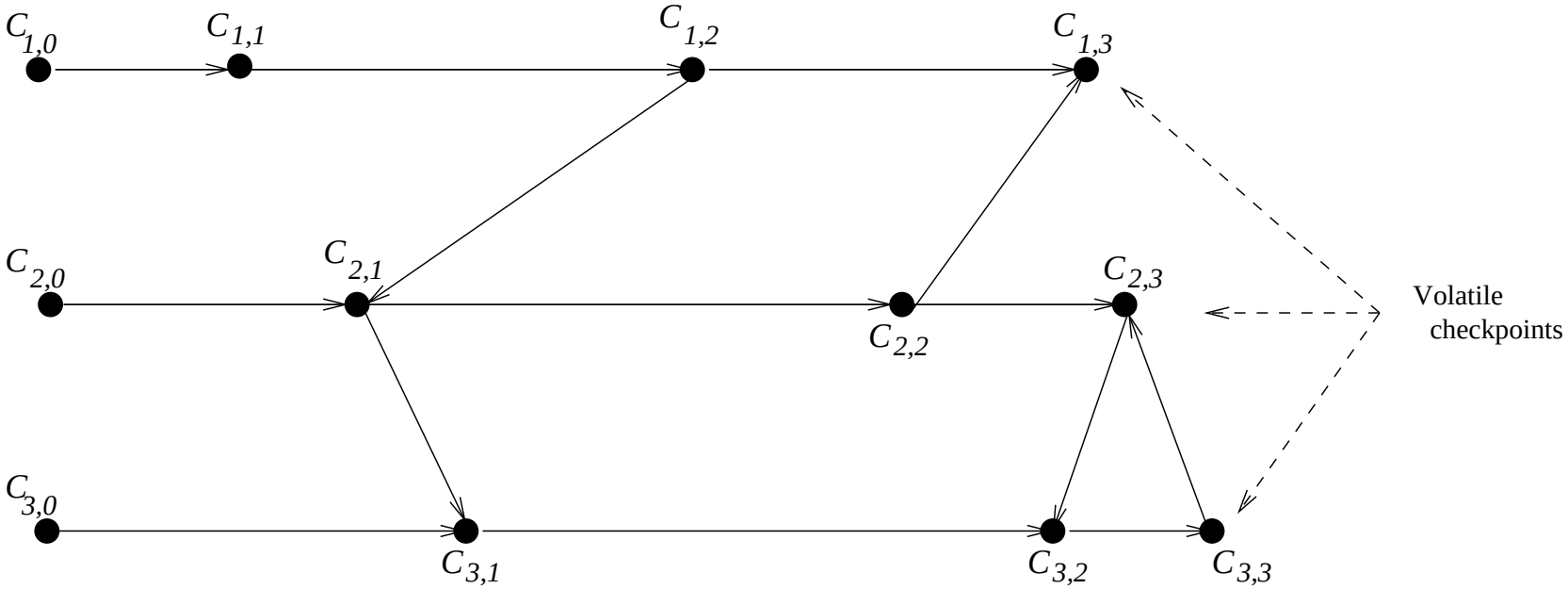

Figure 4.6: The R-graph of the computation in Figure 4.5.

Example of an R-graph

The following theorem establishes the correspondence between the paths in the R-graph and the Z-paths between checkpoints.

Theorem:

Let $G = (V, E)$ be the R-graph of a distributed computation. Then, for any two checkpoints $C_{p,i}$ and $C_{q,j}$, $C_{p,i} \rightsquigarrow C_{q,j}$ if and only if

1. $p = q$ and $i < j$, or
2. $C_{p,i+1} \overset{\text{rd}}{\rightsquigarrow} C_{q,j}$ in G (note that in this case p could still be equal to q).

Examples:

- In Figure 4.5, a zigzag path exists from $C_{1,1}$ to $C_{3,1}$ because in the corresponding R-graph, shown in Figure 4.6, $C_{1,2} \overset{\text{rd}}{\rightsquigarrow} C_{3,1}$.
- Likewise, $C_{2,1}$ is on a Z-cycle because in the corresponding R-graph, shown in Figure 4.6, $C_{2,2} \overset{\text{rd}}{\rightsquigarrow} C_{2,1}$.