Distributed Systems

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Professor Ioannis Liagouris**
for teaching CS351: Distributed Systems
at Boston University [1].

All illustration contain original assets.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

Prerequisites

This text assumes the reader has a basic understanding of computer science and programming. It will also assume they are somewhat familiar with computer architecture and operating systems at a high level. The text will review these concepts briefly for completeness, but it will not try to teach them from scratch or provide a full understanding of these topics.

The main focus will be on distributed systems, and will touch on:

- **Concurrency and Parallelism**
  - Concurrency, Parallelism, Threads
- **Consistency and Fault Tolerance**
  - Consistency, Fault-tolerance, Atomicity
- **Distributed Systems and Coordination**
  - Asynchrony, Coordination, Logical Time, Snapshots
- **Consensus Algorithms**
  - Raft, Paxos, Consensus
- **Replication and Data Management**
  - Replication, Sharding, Cluster
- **Protocols and Computing Models**
  - RPC, 2PC, Broadcast
- **Technologies and Tools**
  - MapReduce, Spanner, Dynamo, GFS, TLA+, Golang

— 1 —

Introduction

2.1 Saving System State: Snapshots

This section discusses saving state. This is useful for fault-tolerance and system migrations.

Definition 1.1: Snapshot

A **snapshot** is a consistent global state of a distributed system at a specific point in time.

Definition 1.2: Consistent vs. Inconsistent Snapshots

To evaluate a snapshot's consistency, we compare events in the system **pre-snapshot** (events before the snapshot) and **post-snapshot** (events after the snapshot). The snapshot itself is instantaneous, like a photograph. Given an event ordering r:

- **Consistent Snapshots**: Respect causal dependencies. Let there be events e_1 and e_2; If $e_1 \rightarrow_r e_2$, then e_1 must be included in the snapshot if e_2 is present.
- **Inconsistent Snapshots**: Violate causal dependencies. If e_2 is included without the causally preceding event e_1, then the snapshot is inconsistent.

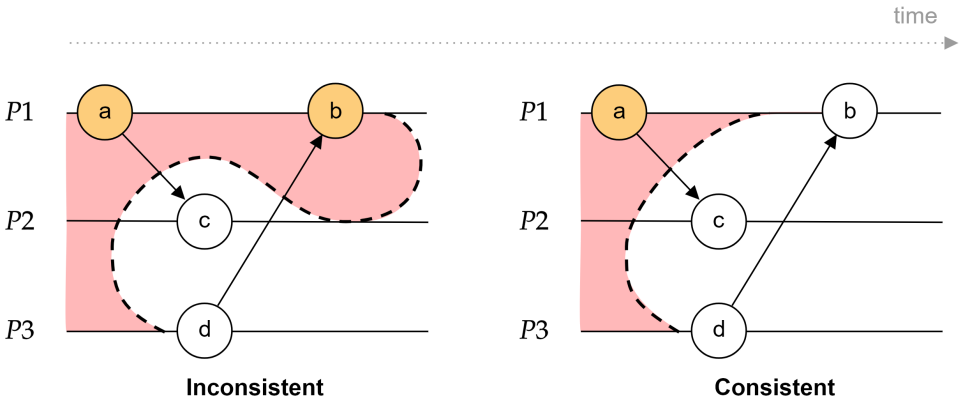


Figure 2.1: Inconsistent vs. Consistent Snapshots (pre-snapshot highlighted in red)

Here, in the inconsistent snapshot, b is included without d, its causally preceding event. In the consistent snapshot, only a is included. In this snapshot c and d could be added without violating causality.

There are many snapshot protocols, but we will focus on the **Chandy-Lamport Algorithm**.

Definition 1.3: Chandy-Lamport Algorithm

The **Chandy-Lamport Algorithm** is a snapshot algorithm that is used in distributed systems for recording a consistent global state of an asynchronous system. In this protocol:

- The snapshot procedure does not disrupt other processes.
- Each process records its local state.
- Any process can initiate the snapshot.

This model requires:

- **No Failures**: No failures during the snapshot.
- **First in, First out Channels (FIFO)**: no lost or duplicated messages.
- **Strongly Connected Network**: All processes can reach every other process.
- **Single Initiator**: Only one process can initiate the snapshot.

The initiator then:

- Sends a **Marker** message to all outgoing channels.
- Records local and incoming channel data.

Recipients of the marker message:

- Designates the channel which the marker arrived as **empty**.
- Records their local state and all other incoming channels except the empty one.
- Sends the marker to all outgoing channels.

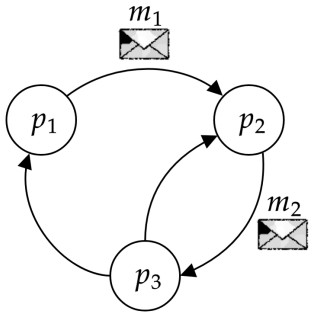
Completion: When all processes have received and sent a marker, the snapshot is complete. This means every processes incoming channel is empty, hence the conclusion of the snapshot.

Tip: Leslie Lamport recalls the Chandy-Lamport Algorithm's creation as follows:
"The distributed snapshot algorithm described here came about when I visited Chandy, who was then at the University of Texas in Austin. He posed the problem to me over dinner, but we had both had too much wine to think about it right then. The next morning, in the shower, I came up with the solution. When I arrived at Chandy's office, he was waiting for me with the same solution."

- https://lamport.azurewebsites.net/pubs/chandy.pdf

Observe the following illustration of the Chandy-Lamport Algorithm given processes p_1, p_2, p_3:

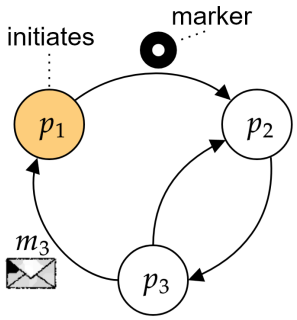
(1)



Snapshot

p_1	p_2	p_3

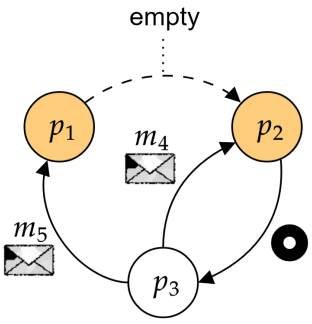
(2)



Snapshot

p_1	p_2	p_3

(3)



Snapshot

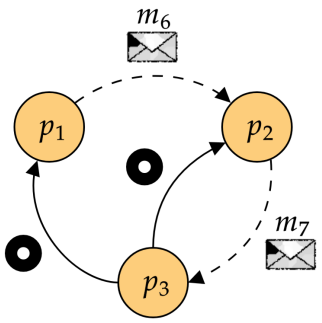
p_1	p_2	p_3
m_3		

Examining these first three steps: (1) The system before the snapshot. (2) p_1 initiates the snapshot sending a marker to p_2 and recording its local and incoming channel's state. (3) p_2 receives the marker, designating the incoming channel from p_1 as empty, sends the marker on outgoing channels, and begins recording. We also see, p_1 has recorded an incoming message m_3 from (2).

We continue on the next page.

We continue the snapshot process with the following steps:

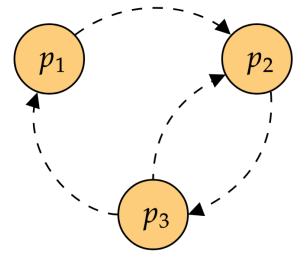
(4)



Snapshot

p_1	p_2	p_3
m_3 m_5	m_4	

(5) Complete



Snapshot

p_1	p_2	p_3
m_3 m_5	m_4	

(4) p_3 received the marker from p_2, and designates that incoming channel as empty. It begins recording state, sending out the marker to p_1 and p_2. Take notice that messages m_4 and m_5 have been recorded from (3). (5) All channels are now empty, concluding the snapshot. Take note that m_6 and m_7 were not recorded as they were sent on empty channels.

Bibliography

[1] Ioannis Liagouris. Cs351: Distributed systems. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.