Distributed Systems

Siddharth Bhat

Spring 2020

Contents

Chapter 1

Introduction

Textbooks is ”Distributed Systems: Principles, Algorithms, and Systems: Khsemkalyani and Singhal”. Other books are Gerard Tel, Nancy Lynch.

- Class presentation: 5 marks
- Project: 20 marks
- Assignments: $2 \times 5 = 10$ marks
- Quiz, Mid sem, End sem: $20 + 15 + 30 = 65$

TAs are Additya Popi, Aman Bansal, Avniash Nunna, Devansh Gautam, Pratik Jain, Karandeep Janeja.

1.0.1 Trivia: The CAP Theorem

Consistency — A guarantee that every node returns the same, most recent, successful write. Every client has the same view of the data.

Availability — Every non failing node must be able to respond for all read and write requests in a reasonable amount of time.

Partition Tolerance — The system continues to function in spite of network partitions.

CAP theorem tells us that we can only guarantee two of these three properties.

Chapter 2

Time in distributed systems

We have n processes P_i. A set of channels C_{ij} that connects P_i to P_j. We have three kinds of events: Local event, Message sent, Message received.

We denote an event e that happened causally before j as $e \xrightarrow{f}$.

A logical clock is a function that maps events E to a time domain T, where a time domain is partially ordered. We have a clock function $C : E \to T$. We are looking to create different classes of logical clocks that have different properties:

- Consistent: $e_i \to e_j \implies C(e_i) < C(e_j)$.
- Strongly Consistent: $e_i \to e_j \iff C(e_i) < C(e_j)$.

2.1 Scalar time

This was proposed by Leslie Lamport in 1978. The time domain is a set of nonnegative integers. The logical local clock of a process p_i and its local view of the global time are combined into one integer variable C_i.

- Rule 1: Before executing an event (send, receive, internal), process p_i executes $C_i \leftarrow C_i + d \ (d > 0)$
- Rule 2: Each message bundles the clock value of its sender at the sending time. When a process p_i receives a message with timestamp C_{msg}, it executes the following actions:

$$C_i \leftarrow \max(C_i, C_{msg})$$
$$\text{Run rule 1}$$

The point of this scheme is that a timestamp assigned to an event will be greater than all of the events that this event *could causally depend on*. However, this is not strongly consistent.

Scalar time is monotonic: $e_i \to e_j \implies C(e_i) < C(e_j)$.

We can induce a total ordering using this scheme. We can create a tuple (t_i, p_i) where t_i is the timestamp, p_i is the process id. Order these using lexicographic ordering, and this is now a total order.

This also allows us to perform event counting. If we always increment by 1, then we know that for an event with e with timestamp t, e is dependent on at least $(t-1)$ events before it.

The lack of strong consistency is not achieved. This is because of the bottleneck of using a single clock that has a single local clock and a single global clock. Thus, the causality of events across processors is lost.

Vector time solves this problem, using large data structures.

2.2 Out of order messaging and consistency of vector time

We wish to understand if the messages are out-of-order, we wish to understand what happens to consistency and strong consistency.

2.3 Omega and Butterfly Networks

Unified memory access versus NUMA. Different topologies.

Multistage logarithmic network. cost is $O(n \log n)$, latency is $O(\log n)$.

Chapter 3

Vector clocks

Each process keeps track of knowledge of how all other processes are processing. So each process p_i has a vector v_i, such that $v_{me}[j]$ represents me's view of j's time. $v_{me}[me]$ is updated monotonically, while $v_{me}[j] (j \neq me)$ is updated whenever a message from j is received.

We order as $v \leqslant w \equiv \forall i,\ v[i] \leqslant w[i]$. $v = w \equiv \forall i,\ v[i] = w[i]$. $v < w \equiv (v \leqslant w) \wedge (v \neq w)$.

Vector clocks are strongly consistent: If two events are concurrent, then we will assign incomparable timestamps. The intuition is that if P_1, P_2 have not communicated, then $P_1[1]$ will have progressed while $P_2[1]$ would not have, Similarly, $P_2[2]$ would have progressed while $P_2[1]$ would not have.

Also, summing up all the entries of the vector gives me the number of events that the event is causally dependent on.

strong consistency comes at the expense of storage.

Chapter 4

Lecture 3

4.1 Models of distributed computing

We model the connections and nodes as a directed graph. A distributed application is a connection of processes on a distributed system. A distributed programing is a set of n asynchronous processes $p_1, p_2, \ldots p_n$ that communicate by message pasing over the network. WLOG, we assume that each process runs on a different process. The global state is the messages in transit and the internal state of each processor.

e_i^x denotes the xth event at process p_i. $H_i = (h_i, \rightarrow_i)$. h_i is the set of events procduced by p_i and $\stackrel{i}{\rightarrow}$ expresses causal dependence.

Lamport's happens before: $e_i \rightarrow e_j$ (direct / transitive dependence). $e_i \not\rightarrow e_j$ Event e_j is unaware of e_i.

$e_i || e_j \equiv e_i \not\rightarrow e_j \wedge e_j \not\rightarrow_i e_i$.

Models of communication: FIFO: message ordering is preserved.

Non FIFO: Channel acts like a set, sender adds messages, reciever process removes messages.

Causal ordering model: If we have two messages that are causlly related, m_{ij}, m_{kj} if $send(m_{ij}) \rightarrow m_{kj})$, then $rec(m_{ij}) < rec(m_k j)$.

Note that $CO \subseteq FIFO \subseteq Non - FIFO$

LS_i^x is the state of process p_i after which event i has happened, event $i + 1$ has not.

$SC_{i,j}^{x,y}$ is all messages p_i has sent up to the even e_i^x, which process p_j has not received upto event e_j^y.

Models of process communication: Synchronous and asynchronous. In Synchronous models, the sender process blocks until the message has been received. In the Asynchronous model, the sender process has no.

Chapter 5

Lecture 4

5.0.1 Global snapshots

We want to understand how to capture global snapshots, based on the kind of message passing that is allowed. This is useful if we want to, say, perform a rollback. The lack of a global clock, and the lack of common memory makes this difficult. Can get out-of-thin-air style situations, if we simply take a snapshot at any point in time.

If there is any message passing, then we cannot do this.

If we have a message that we sent, which has neither been received nor in transit, then we cannot take a valid global snapshot.

$$
\begin{aligned}
LS_i^t &\equiv \text{local state of process } i \equiv \text{All events executed by process } i \text{ till time } t \\
SC_{ij} &\equiv \text{state of channel } i \to j \equiv \{m_{ij} \ : \ send(m_{ij}) \in LS_i \wedge rec(m_{ij}) \not\in LS_j\}
\end{aligned}
$$

To formalize, let LS_i^x denote the local state of process P_i after the occurrence of all events until the event e_i^x. For example, LS_i^0 is the initial state of P_i.

The global state of a distributed system is a collection of the local states of the processes and the channels. This is defined as $GS \equiv \{\cup_i LS_i\} \cup \{\cup_{i,j} SC_{i,j}\}$.

A global state GS is a consistent global state iff:

$$
\begin{aligned}
&send(m_{ij}) \in LS_i \implies m_{ij} \in SC_{ij} \oplus rec(m_{ij}) \in LS_i \qquad (\oplus \text{ is XOR}) \\
&send(m_{ij}) \not\in LS_i \text{TODO}
\end{aligned}
$$

We can interpret these in terms of cuts. A cut slices the space-time diagram into past and future. A consistent global state is a cut where every message that was received in the *PAST* of the cut was sent in the *PAST* of the cut.

5.0.2 Chandy-Lamport algorithm for global snapshots

We assume FIFO queues.

We use special marker messages. One process acts as an initiator, starting the state collection by following the marker sending rule below.

One process acts as the initiator, which starts the state collection by following the marker rule below. The initiator P records its own state. For every outgoing channel C, if a marker has not already been dispatched, P dispatches a marker. (Then, P continues regular communication. Check this, seems dodgy)

If Q has not received a marker, then we record the state, and we note the channel C_{marker} along which the marker came as *empty*. Then, Q follows the marker sending rule.

If Q has already received a marker, it records the state of C_{marker} as the sequence of messages received along C_{marker} after Q's state was recorded, before Q received the marker along C.

Once process P_i has received a message from every other process P_{-i}, P_i can write down its local state and its channel state to the initiator.

Clearly, we create a global snapshot: we will discuss its consistency next. Every process P_i wrote down its own state and all outgoing channel states.

5.0.3 Marker analysis

Once a process P receives a marker for the first time, P records its state. Now look at the kinds of messages P can recieve:

from the initiator I: a message $I \xrightarrow{msg} P$ that was sent by I before the marker $I \xrightarrow{m} P$

I dislike this way of viewing things. We should only state stuff from the perspective of P — but since we have FIFO order, since msg was sent before m, msg will be *received* after m.

The initiator sent this before marking its global state. The message reached P after P marked its state. Hence, this message is a transit message.

from the initiator: the marker

The initiator sent this before marking its global state. The message reached P after P marked its state. Hence, this message is a transit message.