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:

$$C_i \leftarrow \max(C_i, C_{msg})$$
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.