Distributed Systems Module

Concurrency Parallelism & Distributed Systems

Carlos Segarra

January 11, 2020

Contents

1 Concepts of Distributed Systems **2**
1.1 Definition of a Distributed System 2
1.2 Challenges of Distributed Systems 2

2 Distributed Algorithms **2**
2.1 Time & Global States 2
2.2 Coordination and Agreement 3

3 Distributed Shared Data **5**
3.1 Distributed Transactions 5
3.2 Consistency & Replication 5

1 Concepts of Distributed Systems

1.1 Definition of a Distributed System

1.2 Challenges of Distributed Systems

2 Distributed Algorithms

2.1 Time & Global States

Time

Physical Clocks

Physical clocks allow to synchronize nodes **within a given bound.** Synchronize at least every $R < \frac{\delta}{2\rho}$ **to limit skew between two clocks to less than** δ. Where:

- R: resyncrhonization interval.
- ρ: maximum clock drift rate.
- δ : maximum allowed clock skew.

Logical Clocks

Implemented to capture the happened-before relation. They satisfy:

1. If a and b are two events in the same process, $a \to b \Rightarrow C(a) < C(b)$
2. If a sends a message to $b => C(a) < C(b)$

Lamport's logical clocks:

- Each process P_i has a counter C_i
- C_i is updated using the following rules:
  1. When an event happens at P_i increment C_i by one.
  2. When P_i sends a message, set $ts(m) = C_i$
  3. When P_i receives a message, set $C_i = \max(C_i, ts(m))$ and then increase C_i by one.

Lamport's clocks do not guarantee that if $C(a) < C(b) \Rightarrow a \to b$.

Vector clocks:

- Each process P_i has an array $VC_i[1, \ldots, n]$
- It is updated as follows:
  1. When P_i sends a message m, it adds 1 to $VC_i[i]$ and sends m with $ts(m) = VC_i$.
  2. When P_j receives a message from P_i, it updates each $VC_j[k]$ to $\max(VC_j[k], ts(m)[k])$ and increments $VC_j[j]$ by one.

Global States

A global state of the system is necessary for:

- Failure Recovery
- Detection of Properties: deadlocks, termination
- Debugging

We define some concepts:

1. The **history** of a process p is the sequence of events ocurred at that process: $h(p) =< p_0, p_1, \cdots >$ (either internal or message sending).

2. The **state** i of process p is p's history until event i: $s_i(p) =< p_0, \ldots, p_i >$.
3. The **global history** is the union of all the individual histories.
4. A **cut** is the global history up to a specific event in each process history.
5. A cut is **consistent** if it contains all the *happened-before* events. A consitent cut corresponds to a **consistent global state**.
6. A **run** is a total ordering of all events in a global history consistent with each local history.
7. A **linearization** or **consistent run** is a run consistent with the *happened-before* relation.
8. We say state S' is reachable from S if there is a linearization such that S preceeds S'.

Distributed Snapshot

Global Predicates

Consistent global states form a lattice with reachability relation between sets. A **global state predicate**, φ is a property that is either true or false for a global state.

- A predicate is **stable** if once it becomes true, it remains true for all reachable states.
- A predicate is **non-stable** if it can become true and then false.
- A predicate φ possibly happened: if it is true for any of the consistent states in the lattice.
- A predicate φ definately happened: if all paths from origin to end contain a consistent global state for which the predicate is true.

2.2 Coordination and Agreement

Mutual Exclusion

Problem: A set of processes in a distributed system want exclusive access to some shared resource. The desired properties are:

1. **Safety:** at most one process may execute at a time.
2. **Liveness:** requests to enter and exit eventually succeed.
3. **Happened-before ordering**

1. Permission-Based Solutions:
   (a) **Centralized Algorithm:** a coordinator grants access to the shared resource, single point of failure.
   (b) **Lin's Voting Algorithm:** decentralized algoritm with N coordinators.
   (c) **Ricart & Agrawala's Algorithm:** multicast with logical clocks, send request to all other processes and decide basing on logical time. Variant receiving votes from a subset of the processes (M subsets of size K being $\sqrt{N}$ the optimum).
2. Token-Based Solutions:
   (a) Organize processes in a logical unidirectional ring.
   (b) A **token** message circulates around the ring.
   (c) Only the process holding the token can enter the critical section.

Election Algorithms

Election algorithms are techniques to pick a **unique** coordinator (leader). Desired properties are:

1. **Safety:** a participant is either non-decided or decided with the non-crashed process with the largest ID.
2. **Liveness:** all processes eventually participate & either decide a coordinator or crash.

Any process can initiate an election (several ones may run concurrently). Some algorithms:

- **Bully Algorithm**
- **Chang and Robert's Ring Algorithm**

Some can tolerate failures, but none of them can deal with network partitions.

Multicast Communications

It is an important service in distributed systems to disseminate data reliably to large number of users. It is also used to implement several distributed algorithms. Different types:

- Multicast: send a message to a process group.
- Reliable Multicast: deliver messages to all or no process in the group.
- Ordered Multicast: deliver messages while fulfilling ordering requirements.
- Atomic Multicast: deliver messages in the same order to all processes and any process can fail. Solution for multicasing in open groups with **faulty** processes. Deliver messages only to **non-faulty** members. A membership service keeps all members updated on who the non-faulty members are (send **view messages** of group membership in total order). View changes when processes join/leave the group. Each message is associated with a group view (multicasts cannot pass across view changes).

Consensus

General form of agreement: some processes must agree on a value in a finite number of steps in the presence of failures. Some of the desired properties are:

- **Termination:** Every non-faulty process must eventually decide.
- **Agreement:** The final decision of every non-faulty process must be identical.
- **Validity:** If all the non-faulty processes proposed the same value, then the final decision must be that value.

With **correct** processes and **reliable** communication, agreement is straightforward. With **unreliable** communication, agreement **cannot be guaranteed**. With **reliable communication**, crash-faulty processes, and synchronous system we use the **Dolev & Strong's Algorithm**. With **byzantine-faulty processes** and reliable communication in a synchronous system we face the **Byzantine Generals Problem**. Byzantine processes may work together maliciously. We have interactive consistency requirements:

IC1: All loyal lieutenants obey the same order (agreement).

IC2: If the commander is loyal, then every loyal lieutenant obeys the order he sends (integrity).

Impossibility result: with three processes, no solution can work with even one traitor. **No solution with fewer than $3m + 1$ generals can cope with m traitors.**

Faulty processes and **reliable** communication in an **asynchronous** system, no algorithm can guarantee to reach consensus.

3 Distributed Shared Data

3.1 Distributed Transactions

Introduction

The goal is to provide atomicity and isolation to a group of operations at a server in the presence of multiple clients and process crashes. The desired properties are:

- **Atomicity:** either all operations are completed or none of them is executed.
- **Consistency:** takes the system from one consistent state to another.
- **Isolation:** updates of one transaction are not visible to other transactions until it commits.
- **Durability:** persistent once completed successfully.

Two transactions are **serially equivalent** if all pairs of conflicting operations are executed in the same order at all the shared objects.

Concurrency Control

Schedule concurrent transactions so that they execute preserving **serial equivalence**.

1. **Strict Two-Phase Locking**: a transaction is not allowed new locks after it has released one to get serial equivalence. Locks must be held until transaction commits or aborts. To increase concurrency, we use R and W locks. However, deadlocks may appear, to prevent them we use wait-for graphs to represent waiting relations. We can also use lock timeouts.
2. **Timestamp Ordering** In order to choose an adequate timeout, we map each transaction with a **unique** timestamp. We then use the following rules:
   - Transaction T performs a **write** operation: valid only if object was last read and written earlier. Create tentative version with the current timestamp.
   - Transaction T performs a **read** operation: valid only if object last written by an earlier transaction.
3. **Optimistic Concurrency Control**: based on the premise that most transactions do not conflict. Three phases:
   (a) **Working Phase:** transaction keeps tentative versions of each object that it updates, together with the R and W sets. Read operations are directed to the tentative version (if exists) write operations to the tentative one.
   (b) **Validation Phase:** check for conflicts with overlapping transactions.
      i. **Backward Validation:** compare the TX read set with the write set of committed transactions.
      ii. **Forward Validation:** compare the TX write set with the read set of active transactions.

Distributed Transactions

A transaction can access objects managed by multiple servers. To ensure atomicity, all the servers accessed by a transaction must agree on the final outcome of the execution (commit/abort). **One-Phase Commit** is not feasible, we need to use a **Two-Phase Commit** protocol (2PC). 2PC is a blocking protocol (safe, but not live).

For distributed concurrency control, the same techniques exposed before apply.

3.2 Consistency & Replication

Introduction

Reasons for replication:

1. Increase **availability**: data is available despite server failures and network partitions.

2. Enhance **reliability**: data is correct on the presence of faults.

3. Improve **performance**: supporting enhanced **scalability**.

Issues with replication:

1. Replication should be **transparent**.

2. Replicated data should be **consistent**.

Data-Centric Consistency Models

Consistency models in a data store:

- Each process has a local replica of the data store.

- Write operations to a local replica need to be propagated to remote replicas.

Strong Consistency Models:

- Strict Consistency: any read on data item x returns the value of the **most recent** write on x (not feasible in distributed systems).

- Sequential Consistency: all processes see **the same interleaving** of operations.

- Linearizability (Strong Consistency): operations receive a timestamp, sequential consistency + $ts(x) < ts(y) \Rightarrow op(x) < op(y)$.

Client-Centric Consistency Models

Consistency Protocols