Lecture 15: March 23

Lecturer: Prashant Shenoy *Scribe: Fubao Wu*

15.1 Decentralized Algorithm

Decentralized algorithm uses voting. It assumes n replicas of coordinator threads. Any process needs majority vote $m > n/2$ to acquire the lock. This algorithm works even when some coordinators crash.it has the probability that k coordinators crash that $P(k) = C_k^m P^k (1-p)^{m-k}$

15.1.1 Distributed Algorithm

This algorithm is proposed by Ricart and Agrawala, It needs 2(n-1)messages.it is based on event ordering and timestamps. It grants the lock to the process which requests first. The process is as follows:

When a process k wants to enter critical section, it:

- Generates a new time stamp $TS_k = TS_k + 1$
- Sends $request(k, TS_k)$ to all other processes
- Waits for $reply(j)$ from all other processes
- Enters critical section

When a process j receives $request(k, TS_k)$, it:

- Sends $reply(j)$ if it is not interested
- Queues $request(k, TS_k)$ if it is already in critical section
- If it wants to enter: if $TS_k < TS_j$ sends $reply(j)$, otherwise queues $request(k, TS_k)$

Advantages:
fully decentralized algorithm

Disadvantages:
N points of failure: any failed process can block the system because it cannot send any reply to any request. All processes are involved in all decisions. Any overloaded process can become a bottleneck.

15.1.2 A Token Ring Algorithm

A token ring algorithm arranges the processes as a logical ring and makes them pass a special package named token. Every process has to wait for the token to enter the critical section. After it is done, it passes the

token to its next process on the ring. If it has no interest in the critical section, it can just pass the token immediately. Obviously this algorithm guarantees only one process can enter the critical section at a time since there is only one token.

Detecting token loss is non-trivial. When a process with the token is down, the system should be able to recover the token. LAN protocol, which is very similar to Token Ring Algorithm, solves this by giving every process a fixed amount of time. If a process does not see the token before its time period, it knows the token is lost and regenerates the token.

This method does not work here because the system cannot control the behavior of every process. Each process can hold the lock for an arbitrary period. Therefore the system is not able to tell whether the process has crashed or it is just slow. So basically in all the above algorithms, if a process with the lock crashes , it causes a deadlock.

15.2 Transactions

Transactions provide high level mechanism for atomicity of processing in distributed systems. They have their origins in databases. There is a set of properties, ACID, that guarantees that transactions are processed properly:

- **A**tomicity Atomicity means all or nothing. The entire transaction fails if any part of the transaction fails. It cannot be partially done.
- **C**onsistency Any transaction will bring the system from one consistent state to another. It cannot violate the rules defined in the application.
- **I**solation The effect of an incomplete transaction is invisible to other transactions.
- **D**urability Changes are permanent once a transaction completes.

  A naive approach to guarantee these properties is to apply a global lock. Every time the system executes a transaction it locks the entire database. So only one transaction can be executed at a time, which means no parallelism at all. This does not make sense because most transactions has nothing to do with each other.

  Another option is to apply rollback. All transaction are executed concurrently, but before committing the change, the system checks if the state is consistent. If not, it rollbacks the transactions and execute them from the beginning again. T here are two types of transactions in distributed systems:

Nested Transaction Subtransactions of a nested transaction are executed on different databases. Nested transactions are already done on traditional databases.

Distributed Transaction Subtransactions of a distributed transaction are executed on the same database, but the database itself is distributed on multiple machines.

Two methods are proposed to implement transactions. They both figure out some way to support rollback operation.

Private Workspace Each transaction can get a copy of data and modify the copy. It checks if the change is consistent before it commits the change.

Write-ahead Logs Each transaction makes changes directly to the data, but it writes to log before making changes. If it is aborted finally, system can undo the changes according to the log. The log is usually distributed on different machines. Sometimes the system must rollback multiple transactions together because they impact each other.

15.2.1 Concurrency Control

The goal of concurrency control is to allow system to execute several transactions simultaneously as if they are done serially. It can be implemented in a layered fashion:

Transaction Manager It is the top layer. It is responsible for creating transactions, checking whether a change is consistent, and rollback if necessary.

Scheduler Once a transaction begins, the scheduler deals with the locks. It may get a lock of a specific record, a field, or the entire database if the database supports fine-grained locking.

Data Manager It is at the bottom. It executes the read or write on the database.

In a distributed concurrency control scenario, each machine has its own scheduler and data manager, but there is only one global transaction manager.

15.2.2 Serializability

- Serialiability is used to describe whether the execution of transactions looks the same as if they are executed serially.
- Every transaction consists of one or more operations.
- A schedule of transactions is a list of operations ordered by time. It indicates the order in which the operations are executed. Since transactions are executed concurrently, operations of different transactions may interleave with each other.
- A schedule is serializable if its outcome is the same as if the transactions are executed serially.

15.2.3 Optimal Concurrency Control

- In optimal concurrency control algorithm, the system just executes all transactions directly. Before a transaction commits its change, it checks if the data it modified is already changed by other transactions. If it is, it conflicts with other transactions. So it aborts and tries again. This is based on the insight that conflicts are rare in most cases. Advantages:
- It is deadlock free because no lock is used
- It can achieve the maximum parallelism in the best case

Disadvantages:

- It has to rerun transaction if aborts
- The probability of conflict rises substantially at high loads

  This algorithm is not used widely in commercial systems.

15.2.4 Two-Phase Locking

Two-phase locking is a common technique that guarantees serializability in concurrency control. It defines transactions behavior when acquiring locks:

- When a transaction is in its growing phase, it acquires locks and no locks can be released
- When a transaction is in its shrinking phase, it releases locks and no locks can be acquired Strict two-phase locking ensures that all locks should be released at the same time.

Deadlock can happen in this algorithm. It can be avoided by defining the order in which locks are acquired.

15.2.5 Timestamp-Based Concurrency Control

This algorithm is based on Lamports clocks. It tells whether two transactions conflict by checking the timestamps of transactions and records. Each transaction Ti is a given a timestamp $ts(Ti)$. If Ti intends to do any operation that conflicts with another transaction Tj with timestamp $ts(Tj)$, then ts(Ti) is compared with $ts(Tj)$. If $ts(Ti) < ts(Tj)$, then transaction Ti is aborted, and then later restarted with a larger timestamp.

The idea here is that a transaction with a larger timestamp occurs later in time than a transaction with lower-valued timestamp. Since, transaction with lower-valued timestamp is preceding in time, it is aborted.