CS 5412 - Cloud Computing

Recitation 02/19: Replication

Sagar Jha

Note: Requests, commands, operations, updates, etc. are used interchangeably throughout. Also, "node" is used sometimes instead of replica.

Replication is for fault-tolerance: it helps with availability and consistency.

1 Benefits of replication

1. **Fault tolerance**: Replicas store the same state. This helps with availability and consistency. If some replicas fail, others can be used to access the state (availability) and the state is not lost (consistency).
2. **Performance**: Since replicas stores the same state, they can answer queries in parallel, independently. Updates can also benefit from parallelism, although it is complicated since each replica receiving updates from clients simultaneously is at odds with the necessity to replicate these updates.

2 Replication is non-trivial

Replication is challenging because we must guarantee that replicas continue to store the same state through a series of updates.
To appreciate some of the challenges, we consider a naïve approach where replicas respond to requests separately and send the updates to each other.
Consider an airline ticketing system that supports the operation of booking a ticket specified by a ticket id and a customer id, $book(tid, cid)$. Suppose that the system has two servers r_1 and r_2, and two requests $s_1 = book(t, c_1)$ and $s_2 = book(t, c_2)$ simultaneously arrive at r_1, r_2, respectively. If ticket t is available for booking, then s_1 runs successfully at r_1 and s_2 runs successfully at r_2, but when they are sent to the other replica, each returns an error because t is already booked. This is a case of diverging replica states as replica r_1 thinks that t is booked by c_1, while replica r_2 thinks that it is booked by c_2.
What fundamentally goes wrong in this example? The replicas executed the requests in different order that led to diverging states. Thus, we desire the following property:

⋆ Replicas run the operations in the same order

3 State machine replication

Suppose S_0 is the initial state every replica starts with. If each one of them runs the commands $c_1, c_2, \ldots$ in the same order and every command in every state leads to a unique next state, then each replica will transition through the same sequence of states.
This is the state machine replication (SMR) approach.
The state may be maintained in memory or on disk at the replicas. Alternatively, a log of all the operations (commands) executed by the system can be stored as an ordered list. Then the log can be exposed to the clients that can manage the state themselves.

4 Challenges for SMR

1. What is the replica set?

   The replica set is not static because of failures. Failures also force us to replenish the number of replicas by adding more of them. The information of which nodes belong in the replica set is the *group membership problem*.

2. How to decide on the same order of commands?

   This is the *distributed consensus* problem. We need consensus on the order of the commands, or equivalently, on the next command that should be run. Thus SMR can be seen as solving a sequence of consensus problems, each one deciding on the next command every replica should execute.

   A distributed consensus problem, more generally, for a set of processes is one in which every process starts with an input value, and has to decide on an output value such that the following is satisfied:

   - **Safety/Agreement** Every correct (non-failed) process must decide on the same output value.
   - **Validity/Integrity** The output value must be the input value to some process. Equivalently, if every process receives the same input value, the output should be that value.
   - **Progress/Termination** Every correct (non-failed) process must eventually output a value (in other words, processes must decide on the output value eventually).

   Consensus requires *coordination* - replicas need to coordinate with each other to communicate the requests they are receiving from the clients.

3. *State transfer*: How to synchronize state for a new joining replica?

We established that failures force us to add new members. We need to synchronize state at the joining replicas before they can support user operations.

SMR is a general approach to replication. Actual systems can implement it in different ways. Some systems might decide on the commands to execute in "batches", meaning that instead of replicas deciding on the single next command to execute, they decide on a sequence of commands instead. In some systems, there is a designated leader replica, which decides on the sequence of updates to be committed.
At any single point in time, replicas might differ in the actual state they have, because some of them have not yet processed the next few commands. Real-world processes and network work asynchronously - nodes can execute instructions at different rates, the network may have unbounded delays etc. This begs the question: When is it safe for a replica to "commit" (or apply) an update and change the state? One sample answer is: when a replica knows that the next command has been received by everyone. This guarantees that even if a replica fails immediately after it changes its state, the same state change will be executed by every other replica.

5 Chain replication

Chain replication is an implementation of SMR. It is still not a complete system, because it does not solve the membership problem.
In chain replication, the leader replica (or the head of the chain) receives all the update commands. Each update propagates in a chain from the leader replica all the way to the terminal replica which applies (or commits) the update. The acknowledgements flow back in the chain and the update is applied in reverse order. Note that it is not safe to apply an update in the forward pass, because replicas down the chain are not aware of the update.
Queries are processed by the terminal replica. This guarantees that queries are also well-ordered with respect to updates.
Chain replication is a very simple scheme to solve SMR. It forms bottlenecks at the leader replica which initiates all updates and at the terminal replica which answers all queries. Moreover, performance of chain replication is very sensitive to slow links between any two replicas in the chain.

6 Sharding for scalability

Replication provides limited scalability. Increasing the number of replicas to scale query processing slows down the updates, as the coordination costs multiply with the number of replicas. For example, suppose you are designing your system to respond to 10^{12} queries per second. If each server is capable of processing 10^9 queries per second only, you might be tempted to have 1000 replicas. In practice, the replication factor is very small, typically < 5.

A better approach is sharding your data. If you have 250 shards of 4 replicas each, each replica of every shard will receive 10^9 queries per second on expectation. Even if the shards were to be co-located (meaning that they have nodes in common), updates in a shard will nowhere be as expensive as for the 1000 replicas case.