CS 5412 - Cloud Computing

Recitation 02/14: Replication

Sagar Jha

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

Replication is for fault-tolerance: it helps with availability and consistency.
The main task for replicas is to store the same state. This same state must be maintained while the system responds to requests (reads/writes).

1 Replication is non-trivial

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

2 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.

3 Challenges for SMR

1. What is the replica set?

   The replica set is not static because of failures. Failures also force us to replinish the number of replicas by adding more of them. The information of which nodes constitute the replicas is the *group membership problem*. To be able to implement SMR, it may be crucial that every node knows the membership of the system.

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.

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.

4 Paxos

Paxos is a family of protocols that solves SMR. The state is kept in memory or persisted on disk and transferred to joining members. There's another variant of Paxos which replicates the commands executed in order and stores them in a durable log. A joining replica then simply copies this log to its storage and replays it.

4.1 Classical Paxos

Classical Paxos was originally developed by Leslie Lamport. Its mechanisms are outside the scope of this course, but we will still look at some of its features.

- Fluid membership model: There's no strong notion of a membership in this. Every active node proposes the commands it receives and the protocol decides on which command to execute next. Thus, a node can be arbitrarily slow, or can fail and restart, but the protocol steps are the same. Classical Paxos will progress (run commands) as long as a majority of the nodes are correct.

- Failure assumptions are ingrained in the common run of the protocol. Thus, to the actual failure events, classical Paxos is quite oblivious.

4.2 Virtual Synchrony

Virtual Synchrony was developed by Ken at Cornell. Unlike classical Paxos, it has a strong notion of membership.

- It works with the abstraction of a group (which is like the replica set). A group is an ordered set of its members.
- The group supports the operation of atomic multicast. Each member can multicast messages to the entire group and each member "delivers" the messages (from all members) in the same order. This all-or-nothing property of message delivery is the reason this is called atomic multicast. Take delivery to mean the same as running the command in SMR. Delivery of a message is not the same as receive of a message, because coordination is required to make sure that every node delivers the messages in the same order.
- Any failures or join, leaves etc. of the members lead to a reconfiguration of the membership. The group momentarily pauses the delivery of messages while it is reconfiguring. A run of the system with a given membership is called a "view". A reconfiguration of the membership is called "view change". Thus, the system moves through a series of epochs (views) where each transition is marked by a view change.
- Therefore, in contrast to classical Paxos, there are no failure assumptions in the common case and explicit failure handling in the form of view changes. This gives great performance as failures are quite rare in these kind of systems.
- The group membership decided during a view change effectively solves the consensus problem of the order of delivery. During the view that follows, the members deliver messages in round robin order i.e. the first message of member 0 is delivered first, followed by the first message of member 1 and so on, up to the last member. Then, the second message of each member is delivered in the same order and so on. Thus effectively, we are moving the consensus problem to the view change where consensus on the membership is obtained. The membership then determines the message ordering.

Derecho is based on virtual synchrony.

5 Asides

1. **Strong vs. Weak Consistency**: Demanding that the replicas always maintain the same state is asking for strong consistency. Where does weak consistency lie in this space? Weak consistency allows for a (controlled) divergence of the replica states in a way that still doesn't violate system correctness (for example, selling the same ticket to two different customers is still not desirable). Many systems serve read-only queries using possibly stale data and run a background task to continuously synchronize the states. Think about how this works out for Facebook's TAO.

2. **Split-brain problem**: Suppose that there is a network outage that creates a partition of a group. Each partition might consider all the nodes of the other as failed and form a group of its own. This way, we have two groups going their own way. We don't want this to happen as this will create a scenario where clients talking to different groups will see different system states, violating consistency. This is known as the split-brain problem. To avoid this, we generally enforce that a group must have a majority of the members from the previous configuration. Only one of the partitions can have a majority which guarantees safety, but the system can no longer handle the simultaneous failure of a majority of its members (which is extrememly rare, anyway).

6 Derecho

Replication has been traditionally costly because of overheads of coordination (passing messages etc.). This motivated weakly consistent systems for the cloud. However, two recent trends have challenged this notion:

- Increasing network speeds: Network speeds have increased to 200 Gbps and will soon reach 1 Tbps. On the other hand, processor speeds have largely stagnated.
- Data plane/control plane separation: The fast networking layer moving data should not be stalled by the control layer that involves host CPUs. Data messages consist of actual application data that needs to be replicated, while control messages consist of acknowledgements of data messages, failure suspicions etc. For high performance applications, we want high throughput for data movement and low latency for the control messages. This is best achieved by taking the coordination out of the critical path of data transfer.

Derecho, built at Cornell, achieves fast replication. It manages application structure into its components (called subgroups) and provides SMR for each component. Its data plane is the system RDMC (RDMA multicast) and the control plane is SST (Shared State Table). Both make use of RDMA networks, which provide two important network primitives:

1. Send/Receive: This is like the traditional TCP functionality. The sender sends a message and the receiver decides where to store it. Unlike TCP, it does not involve storing data in kernel buffers and then copying to user space. But, it still requires the receiver node to know about the transfer (and use its CPU cycles to allocate buffer space for an incoming message).
2. Read/Write: In this mode, the receiver provides the sender permission beforehand (in the connection step) to write to or read from a portion of its memory. RDMA writes and reads are completely oblivious to the receiver i.e. the receiver is not aware of any ongoing transfer.

What advantage does RDMA have over TCP-based communication? TCP suffers from the following drawbacks:

1. Context switch required for packet processing: The kernel strips the incoming packet of its headers to determine which application to hand it off to. In RDMA, the NIC (Network Interface Card) does the packet processing, so the host CPU is not involved at all.

2. Memory copy overhead: The kernel receives the TCP packets in sockets open by the user program. The user program has to call the read/write socket API to transfer to its memory. This involves a memory copy which is slow. In RDMA, the NIC directly DMAs the incoming data to the user memory asynchronously in the read/write case and to the host-specified address at receive time in the send/receive case.

Additionally, TCP assumes that messages can be dropped and arbitrarily reordered which adds additional protocol overheads (of acknowledgements etc.). RDMA packets are not dropped or reordered unless there's a failure.
Now, we will look at both systems in detail.

6.1 RDMC

RDMC provides the ability to multicast a message to a group of nodes. It constructs a multicast scheme based on point-to-point unicasts. It employs the binomial pipeline algorithm to achieve high performance.
To motivate the algorithm, first consider a naïve scheme where the sender sequentially sends the message to each receiver (called sequential send). Why is this inefficient? By transferring the same piece of data around, we are wasting network bandwidth. It takes linear time in the number of receivers to multicast a message. Additionally, it only utilizes the sender-receiver links and not the receiver-receiver links.
Instead, consider an enhanced scheme where the sender sends the message to one of the receivers first (first step). Now, two nodes have the message and both act as senders. Each sends the message to a different receiver in parallel (second step). Continuing this way, it takes logarithmic time in the number of receivers to multicast a message. It also utilizes many receiver-receiver links. This scheme is called binomial send.
But, this is still not optimal. Half of the receivers receive the message in the last step and therefore, do not participate in sending. Similarly, one-fourth of the receivers participate only in the second last step. The binomial pipeline algorithm improves on this by dividing the message into blocks and sending the blocks in different binomial sends utilizing many more links.

6.2 SST

SST provides each node a row in a distributed table with columns for data they care about. For example, a column received_num[i] will keep track of how many

messages have been received from node i. Each node maintains a copy of the entire table in memory. A node updates its own row by modifying its local table first and then using RDMA writes to update the row at the remote nodes.

In the next recitation, we will look more closely at Derecho's API and how RDMC and SST coordinate to provide high performance.