FaRM

Overview

- Another version of getting strongly consistent / available distributed transactions
- FaRM is a main-memory distributed computing platform
    - Provides distributed ACID transactions with high availability, high throughput, low latency
- All in one data center, uses configuration manager (i.e. ZooKeeper) to choose primaries / backups
- Sharded with primary / backup replication
    - Can recover as long as each shard has at least one replica alive
- Exploits performance potential of RDMA NICs
    - NIC - Network Interface Card (connects computer to the network)
    - RDMA - feature on some modern NICs that looks for special command packets over the network and executes those commands itself instead of giving them to the CPU
        - The NIC can also look into its own CPU's memory and write to it
            - This allows us to bypass the kernel and driver code and just write stuff to memory we want the CPU to send and the NIC can then send this out
            - The NIC can also just write incoming messages to the CPU and we can just read it
        - Gives hardware ACKs to application code
        - One-sided RDMA are when one computer can use RDMA NICs to read/write memory in another computer without that computer's CPU
    - Normally the CPU handling network things is a huge performance bottleneck
        - Has to handle socket buffers, system calls, driver code, interrupts
        - RPCs norm can't deliver more than 100k per second
- Keeps everything in the RAM (so no disk reads)
    - However they use non-volatile RAM (NVRAM) to ensure it stays intact after crash
    - Normally RAM loses content in power failure or crash
        - We can't just write to RAM of all machines and call it a day because power failure is not independent and can cause all machines to go down
    - We have batteries in every rack which can power machines for a few min after failure
        - After failure, this halts all transaction processing, writes the RAM to SSD, and then shuts down
    - Crashes prevent writing to SSD, but this is why we have the replication
        - We assume that crashes are independent if they're not from power failure
- All of this for millions of distributed transactions per second
    - Much faster than Spanner
    - We also only get a single datacenter
- However, we need some form of concurrency control

Concurrency Control

- Pessimistic concurrency control (two-phase locking) waits for lock on first use of object and holds them until commit / abort
    - Conflicts cause delays
- Instead we'll discuss Optimistic Concurrency Control
    - We read objects without locking but don't install writes until we commit
    - When we try to commit, we validate to see if any other transactions conflicted
        - If not, then we commit, otherwise abort
- If we want the performance increase of FaRM using one-sided RDMA reads, then we can't have the foreign server actively participate in reads through something like locking
    - Therefore, we have to use OCC
- This works best if few conflicts

Transaction Protocol Description

- Primary / Backups replicate the primary log
    - So when things are being added to the primary's log, they are replicated across everything
- Execute Phase
    - Client reads objects it needs from servers (including records that it will write) without locking
        - Always reads from primaries
    - Remembers the version numbers of what it reads
    - Buffers the writes locally
- Commit Phase
    - LOCK
        - Transaction coordinator (TC) sends a new log entry using RDMA to the primary of each written object
            - TC is the client
            - LOCK record contains memory location, version number, new value
        - On receipt of a LOCK (by polling the log entries), we:
            - Check if the object is locked or if the version number is out of date
                - If so, then we respond "no"
            - Otherwise, we reply "yes" and set the object's lock
            - We don't modify the data
            - All of these actions are atomic using CPU locks
            - TC waits for all LOCK reply messages and if any fail, then we abort
                - Append ABORT to primary logs so they release locks and then returns "no"
    - VALIDATE
        - For objects that are read only, we do a one-sided RDMA to check to make sure that things we read still have the same version numbers and are not locked
    - COMMIT BACKUP
        - Tells the backups the new value(s)
        - TC waits to send COMMIT-PRIMARY until all LOCKs and COMMIT-BACKUPS are complete
        - This is just put in the NVRAM logs, but since NVRAM is fault-tolerant, then just putting it there is enough
    - COMMIT PRIMARY
        - TC appends COMMIT-PRIMARY to primaries' logs
            - TC just waits for RDMA hardware to acknowledge that this went through (doesn't wait for primary to process log entry)
            - TC then returns yes to say the commit succeeded
        - When the primary processes COMMIT-PRIMARY, it copies new value, increments version, and clears lock
        - The commit point is when the first COMMIT-PRIMARY is written since at that point the transactions results are all good
            - This means that even if the transaction requires multiple shards, we can still return the commit after a single COMMIT-PRIMARY ACK
            - We need to wait for one ACK because if the TC crashes as well as all backups, we need to some evidence that our transaction actually went through, which can be found in a primary
- This provides serializability because with locks we check if transactions are going to be bad / conflicting because of the different version numbers or the locks
    - Read/write transactions are serializable by point where all write locks are acquired
    - Read-only transactions are serializable at point of last read
- Reads are guaranteed to be serializable:
    - Two concerns about reads:
        - If we have a large object with multiple blocks of data, then we might have the object modified while we're reading
            - Each block of data has a version number and reading it is atomic (norm a block is a single cache line)

            - Therefore we read the whole object and if the version numbers don't match for the whole thing, then we abandon transaction
        - A concurrent writing transaction might cause interleaved data
            - The validate step (which occurs after the execute phase) makes sure that all of our reads all have the same version number and weren't modified between reading some of it and reading all of it
- A pure read-only FaRM transaction uses only one-sided RDMA reads, so very fast

Fault Tolerance

- The Primary / Backup replication means that everything sent to the primary is actually sent to all of the backups
- A client only returns success when COMMIT-PRIMARY is added to at least one primary's log, which means that the shard must have all replicas up to date
- As a result, all servers will have up to date information
- The configuration manager (CM) uses very short leases to keep track of which machines are alive
    - When one dies, then a reelection process causes a new primary to be elected