Distributed Systems (CS60002)

Distributed Coordination Service (Group 06)

TA : Gulab Arora

Suyash Damle 15CS10057
Hardik Tharad 15CS10059
Anubhav Jain 15CS10062
Vivek Mudgal 15EE10055

Feature specifications / APIs

User Features : (Possible)APIs that can be provided to user

1) File System API(assuming multiple file systems are possible in our service), consists of following methods :
    a) CreateFileSystem(params : size, …)
    b) Add/Modify/Read/Delete File(params : fileSystem, dirPath, filename, …)
    c) CreateCheckpoint(returns : checkpointId) , RestoreFromCheckpoint (params : fileSystem, checkpointId)

2) Network Architecture API(supporting any kind of topology), consists of following methods :
    a) Add/Delete Server
    b) Add/Delete Link between two Servers

System Features : (Possible)Features of our service

1) Files distributed over multiple servers, not all servers just contains replicas of same data.
   Pros: more amount of data can be stored
   Cons: complex architecture, more read/write time
   Questions:
   How will data be organized?
   How to find where the file is?
   How to handle failure of servers containing one of the replicas of some files?

   Possible Solutions to Questions: A particular server(along with its replica) contains metadata about file system (how is file system organized and which server contains what data) and it(metadata server) can be queried for all file accesses;
   Problem: Too much load on a single metadata server.

   vs

   All servers just contains replicated data.
   Pros: Less read time, simpler architecture
   Cons: Memory of file system will be constrained to the memory available at each node
   Possible Implementation: A leader is elected and all write are coordinated through the leader(2-phase commit), while all reads happen through all nodes.

2) **Fault tolerance** :
    a) Robust to link failure
        i) **Possible Solutions**: Can have ‘n’ physical paths between any two nodes if n-1 link failure needs to be handled; **OR**

Let TCP itself handle the task of handling link/switch failures and abstract-out those details by assuming that all *active nodes are accessible at all times logically.*

b) Robust to Byzantine failure
   i) **Possible Solution** : We can use OM algorithm for agreement in read and write operations.
   ii) **Problem:** For m Byzantine Failure we require at least 3m+1 servers and data should be replicated in each server. Moreover the message complexity increases. What if the server initiating write operation is itself faulty ?

c) Robust to Crash Faults
   i) **Possible Solution:** To recover from n crash faults, data can be replicated in n+1 servers.

3) **For creating a file system**, the service allots the file system based on memory available in entire system as well as number of node that can store the data(number of nodes important if we assume we need to have some 'n' number of replicas)

4) **Priority/Importance** given to files (can be added as a parameter to addFile() method) : this will be relevant in systems where only some files (and not all) files in the system are replicated, and we need to guarantee the important files are sure-shot replicated.

5) **Load balancing** : Service decides which file to store in which server and where to create replicas (e.g. near the more frequently accessed nodes)
   a) Questions:
   Should we handle replication **dynamically?** What if the load pattern changes over time? Where to create new replicas in case one of the "frequently accessed" servers fail? What to put on a new server, when it joins?
   b) Considerations: Bandwidth, reliability of servers vs access loads of replicas on them. (less accessed files on "cheaper" servers)

6) Handles addition of servers

7) File Locking: to allow a series of updates on a file by a single server. This could be implemented by:
   a) Permission-based : permission from all / from quorum
   b) Token-based : a unique token circulates
   c) Questions:
   In quorum protocol - how to handle failure of servers? (solution: define quorum in terms of **number** of permissions required - the set itself dynamic)
   In token based protocol - how to handle failure of token - owner?

How to know where the token is (solution: directed spanning tree, rooted at owner of the lock)

8) **Consistency models** (Decreasing consistency, increasing performance) :
    a) Linearizability
    b) Sequential consistency
    c) Causal consistency
    d) Eventual consistency (works best in more reads and infrequent writes)

Examples of Solutions (from ZooKeeper or Other Related Systems)

ZooKeeper provides the following **broad API types**:
(This section only describes **broad** level of APIs that ZooKeeper provides, without dealing with specific solutions/algos)

1. File Creation / Modification:
    a. Add a znode (equiv. to creating file/directory)
    b. Delete znode (equiv. to deleting file/directory)
    c. Read from files
    d. Write to files (replication handled internally)
    e. Sync (force propagation)

2. Newer versions of ZooKeeper provide **Dynamic Reconfiguration:**
    a. Addition/Deletion of nodes while the application is still running
    b. It abstracts-out changes in the underlying topology by relying on TCP connections
    c. It does not separately handle partitioning problem - a set of servers not responding are assumed dead

Solutions for the Features that ZooKeeper provides:

1. **File Replication:** ZooKeeper replicates the files over **all** the servers. There are 2 possible ways to handle this:

a. The point of contact with the client takes the onus of replication: In this case, the contacted server itself starts active replication protocol:
   Algorithm:
   i. On write, request sent by local replica to all others using atomic multicast
   ii. On receiving this, replica servers update copy on write and send back acks
   iii. On completion of atomic multicast, success returned to client on write
   Pros: no single point of failure wrt write operations;
   Cons: Since a 2-Ph commit is used to ensure consistency, multiple write requests are handled automatically. However, there is a **possibility of a live lock,** as multiple clients could keep requesting writes repeatedly. Further, the process is quite complicated and is not required for infrequent writes

b. Another option is that the contacted server sends update request to a unique **leader server,** which coordinates and handles the write process(2-Ph commit). **ZooKeeper uses this protocol and we too plan to implement this one.**

2. **<u>Consistency Guarantee</u>:** ZooKeeper guarantees **sequential consistency.** This could be implemented in the following ways:
   a. As stated above(ZooKeeper's method), reads are provided for by local copies; writes handled by leader. Now, if a write/update is in progress on (say) client1's request. Now, a server N_i is contacted for a read by client2 - it serves from its local copy, which may not be updated, if the client2 has got its final "commit" message. Client1, on the other hand, might have got success, if it got its "commit". Clearly, this violates the Linearizability condition(events across different processes could not be linearly arranged). And, it clearly follows sequential consistency conditions.
      Pros: Simpler to implement; writes on all clients alloy reads from local copies directly - allows high read:write ratios
      Cons: Failure of leader leads to failure of update & unavailability for some time.

   b. Another option to enforce this is to follow a multicast from the contacted server directly (as discussed above), or use a majority voting protocol.
      Pros: No failure of service as a whole on failure of any server.
      Cons: Complex implementation

3. **<u>Fault Tolerance</u>:** ZooKeeper takes care of crash faults only, while relying on TCP to handle other things (Byzantine faults could anyway not be handled). Our system would also take care of crash failures explicitly, while link failures would be assumed to be absent.
   a. A way to this is to keep replication on all servers: If n servers form initial service cluster, upto n-1 crash faults could be handled.

b. Else, we might keep replicas on *certain servers.*

4. **<u>Creating new File-Systems</u>:** It is not clear right now whether ZooKeeper provides multiple distinct file-systems or not. However, this feature could be implemented by:
    a. Using distinct namespaces - allowing multiple FS trees on each node
    b. Segmenting the network servers internally - one partition for each FS - transparent to user

    (We are not implementing this feature)

5. ZooKeeper needs no load balancing - as all files are replicated everywhere

6. **<u>File Locks</u>:** ZooKeeper supports file locking by a unique combination of multi-purpose features:
    a. It allows *ephemeral nodes* which expire when the server stop
    b. Watch flags could be placed on these ephemeral nodes by call to *exists()* function. Each ephemeral node acts as distinct indicator of a waiting process on the file
    c. We plan to avoid using this way of locking to avoid complexity
    d. Instead, we plan to use a token-based system.

<u>Our Model</u>

Model Assumptions

The following assumptions are made to simplify the model:-

1) Fully Connected Topology
2) Asynchronous and reliable system
3) FIFO channel
4) No link failure or Byzantine Fault
5) Not more than n-1 crash faults (n - number of nodes in the network)

System Guarantees

1. **Consistency Model:** Sequential Consistency Guaranteed
   Pros: Ordering of events at different processes may not important as they could have happened in some other order in practice anyway due to different reasons(server speed, message delays); Less overhead and easy to implement when compared to Linearizability
   Cons: User may not always get the most updated value while reading data

2. **Fault model:**
   a. Reliability: System is 100 percent reliable for only read operations. We cannot comment anything for write operations. (Depends on how often the leader crashes)
   b. Availability: System is 100 percent available for read operations. We cannot comment anything for write operations (Depends on how much time it takes to elect new leader and resume operations)

Supported Features

1. **Replication of data in every node for faster read operation** :- We will consider that number of read operations are much larger than number of write operations, therefore we will replicate data in every node of our system.
   Problem: How to replicate data in real time so as to maintain sequential consistency ?
   Possible Solution: To elect a leader and have a 2-phase commit protocol for write operations. Read can happen through any node (local file image)

2. **Tolerant to n-1 crash faults** (n number of nodes in the network) :- Our system can tolerate upto n-1 crash faults.
   Questions:
   What if the leader or any node fails during any stage of two phase commit?
   What are the possible scenarios?
   How to decide time-out value at other servers before they assume that leader has failed?

3. System will be **sequentially consistent** as discussed above

4. System successfully **adds new servers with unique id** - uses a form of 2-Ph commit initiated by the present leader to agree upon the id of the new node. Also, the new

server uses another (sponsor) server to replicate all data on itself before taking client requests

Questions:
How to handle failure of sponsor node before setup of new server is complete?
What to do if the new server has lesser storage space than the total replicated data in the system?
Should we allow multiple sponsor nodes for load balancing? (large chunk of data to be copied to new server)

5. **File Locking - for sequence of updates by one server:** We plan to implement this using token-based mutual exclusion.
    a. On need of a file lock, check if a directed spanning tree exists for this file.
    b. If yes, go to root by following pointers to request token. Else, request current leader for initiating a token for this file.

Proposed Model

Different Protocols that may be used:

1) Leader Election: Required if the leader crashes. Since the topology is fully connected, the node with the least id can be elected as the leader
2) New node addition/deletion: Not implementing the zoopkeeper's protocol to add a new node or delete since arbitrary topology is considered in the implementation.
We propose an adaptation of 2-Ph commit run by the existing leader to agree on the id of the new node. It will simplify things.
3) Adding a file: All write requests are redirected to leader. The leader than can use two phase commit protocol for update propagation.
4) Token-based file locking - directed spanning tree created, rooted at the current possessor of the token. In case of failure of this server, current leader creates a new token.

References

1) AG's slides: http://cse.iitkgp.ac.in/~agupta/distsys/Replication.pdf
2) ZooKeeper intro page: https://zookeeper.apache.org/doc/current/zookeeperOver.html
3) ZooKeeper guide for programmers: https://zookeeper.apache.org/doc/r3.3.3/zookeeperProgrammers.html
4) ZooKeeper Internals: https://zookeeper.apache.org/doc/r3.3.4/zookeeperInternals.html
5) ZooKeeper applications for higher layer: https://zookeeper.apache.org/doc/r3.5.4-beta/recipes.html#sc_recipes_Locks