Highly Available In-Memory Metadata Filesystem using Viewstamped Replication

(https://github.com/pkvijay/metaDR)

Pradeep Kumar Vijay, Pedro Ulises Cuevas Berrueco
Stanford cs244b-Distributed Systems

Abstract

This paper describes the design and implementation of a highly available key-value store based metadata filesystem using Viewstamped Replication [1].

Introduction

Most of the distributed systems have to manage the metadata corresponding to the service that they provide. For example, GFS [3] master node maintains the metadata of the entire filesystem. Zookeeper [2] provides a replicated metadata filesystem which can be used to implement other distributed services like locking, leader election etc. It is inevitable that this metadata need to be highly available as this can be a single point of failure. Viewstamped replication [1] provides the algorithm to achieve consensus in distributed systems. We have implemented a simple in-memory key-value store that is replicated and made highly available using viewstamped replication. We have modeled the key-value store interface to be similar to the one in zookeeper, where the key is in the format of UNIX filesystem path and value is a string. At any point of time, there is a single primary and multiple backups. In order to guarantee strong consistency, all update/delete requests have to be issued to the primary. When the primary fails, one of the backups changes role to primary. In the current implementation, the read requests should also be issued to the primary, but with just a few lines of code change, the backups can respond to read requests as well. This system is designed for low read latency and hence the reads do not go through the replication protocol. If the backups allow reads, there is a greater probability of stale reads. To overcome that, sync mechanism, similar to the one in zookeeper needs to be provided. The system can tolerate 'f' failures where the total number of nodes in the system configuration is 2f + 1. The system is being implemented using C++11 on Linux based platform.

Client Workflow

There are two main components in the system: the client library and the backend server executable. The client library provides the APIs for the KV store. The users of the system are expected to use the client library APIs to store their metadata. The client library uses RPCs to transfer the client requests to the backend server. The backend takes care of processing the client request, modifying the actual key-value store and replicating the update/deletion across all the backups. The main KV store client APIs are create, set, get, list and remove. A shell kind of utility is provided that can be used by the clients to invoke different commands to manage the KV store. The shell can be used if the users do not want to write code using the client library.

If the size of the system configuration is N, then N instances of backend server need to be started. One of them would be elected as primary and the others will be backup. Each instance of the server needs to be started with a configuration file. The configuration file contains the details of the client and the replication ports, where the server has to start the RPC listener threads, and in addition, it contains the IP endpoint details of all the nodes in the system. This is how each server instance would know the IP and port details of the nodes in the system with which it has to

communicate to implement the consensus protocol. The client needs to be aware of the system configuration and all client requests should be sent only to the primary. If a request were sent to a backup, then the client would get an error explaining that the node is not primary. In case of primary failure, the client can broadcast the request to all the nodes and only one of them would accept the request, which is the new primary. From then on, the client can just send the request to the new primary.

Server management client APIs can be provided for reconfiguration and shutdown support. This should allow the clients to dynamically add more server instances to the system and to cleanly shutdown any server instance for maintenance reasons.

Design

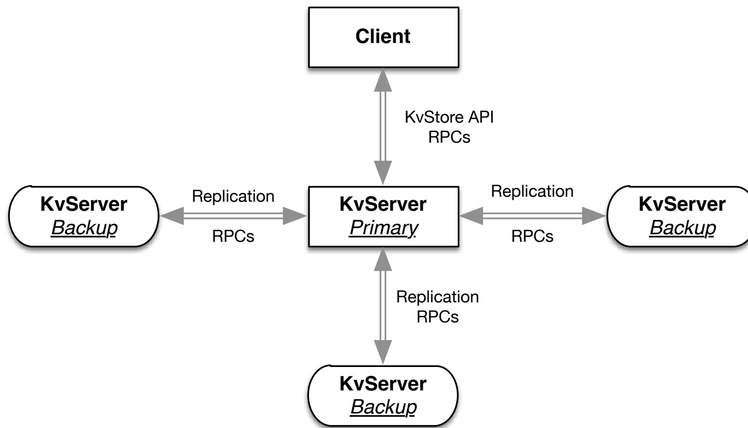

Figure 1 – Architecture

Figure 1 provides the high level architecture of the system, where the client uses RPCs to talk to the primary key-value server, which in turn uses RPCs to replicate the change across the backups.

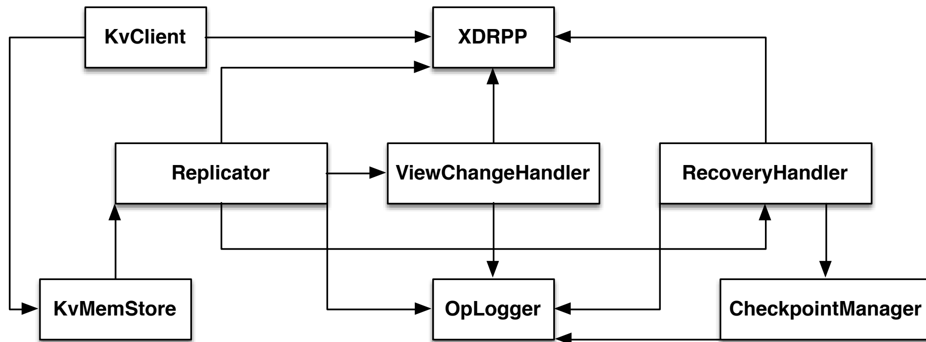

Figure 2 – Module Diagram

Figure 2 provides the high level modules of the system, which are described below.

- **XDRPP** [6] module provides an XDR based RPC [4] implementation. XDR [5] stands for External Data Representation which describes the interface definition language (IDL), which can be used to express data structures, transmitted between machines using RPC. XDRPP comprises of a compiler that facilitates stub generation from XDR based IDLs. It consists of a runtime library that hides the network communication details and facilitates RPC invocations across different machines.
- **KVClient** module provides the client library that allows the clients to link to and invoke the APIs to store their metadata. KVClient uses XDRPP to invoke RPCs to the KVMemStore and returns the results to the clients.
- **KVMemStore** module implements the in-memory key-value data-structure. It implements the KV store APIs and responds to the RPCs invoked by KVClient. It performs the necessary validations before updating the in-memory data-structure. One example of validation is to ensure that the keys are in the proper filesystem format. KVMemStore calls into the Replicator to replicate any changes before committing the actual change into its in-memory structure. If this KVMemStore is running as a backup, then it registers callbacks with the Replicator, so that the backup replicator can make up-calls to KVMemStore to update the committed operations.
- **Replicator** module implements the viewstamped replication [1] protocol. For any update/deletion operation it runs the distributed 2-phase commit kind of protocol as described in the viewstamped replication paper. For any update/deletion operation, replicator commits the operation only if a quorum is achieved for that operation. Replicator uses the OpLogger to log the operations. Replicator runs either in the primary mode or in the backup mode. If it runs in the primary mode, it accepts the replicate requests from KVMemStore and initiates the replication protocol with the backups. If it runs in the backup mode, it accepts requests from the primary replicator, processes them and responds back to the primary. The replicator running in the backup mode also keeps track of the primary health and invokes the failover protocol if it suspects the primary to have died.
- **ViewChangeHandler** module implements the failover protocol as described in viewstamped replication paper. ViewChangeHandler is invoked whenever the backup replicator suspects that the primary has died. This module implements the distributed view-change protocol to select a new primary and the new view. Once a quorum is achieved for the new view, the backups start responding to requests from the new primary. The concept of 'View' is an integral part in viewstamped replication to achieve consensus. Backup replicators accept requests from primary only if they are in the same view. Operation logs are transmitted between the cohorts to implement the view-change protocol. The OpLogger module is used to obtain the latest log.
- **RecoveryHandler** module implements the recovery algorithm as described in viewstamped replication paper. Whenever a node goes in the recovering state, the replicator uses RecoveryHandler to recover the state of the node. The recovery algorithm has to transmit the logs and the OpLogger module is used to obtain the latest log. If the log is being trimmed, then the CheckpointManager is used to construct the complete log or to asynchronously transfer the checkpoint state to the recovering node before the actual recovery algorithm.
- **OpLogger** module facilitates logging the operations. The log is maintained in an in-memory data-structure. It provides the necessary interface to retrieve the log and to check if a particular operation is logged. The log is integral part of viewstamped replication algorithm, which is used to check if a particular operation request should be accepted, or not. It is used to make sure that the operations are committed in the exactly the same order across all the cohorts.

- **CheckpointManager** is used to trim the log and to persist the KV state to disk. The OpLogger notifies this module whenever a certain log size threshold is reached and it asynchronously runs through the log and persists it to disk. It provides the necessary interfaces to obtain the state that has been checkpointed.

Key modules that are missing from the above figure are StateTransfer and Reconfiguration. Refer to 'Status' and 'Future Work' sections for more details on those.

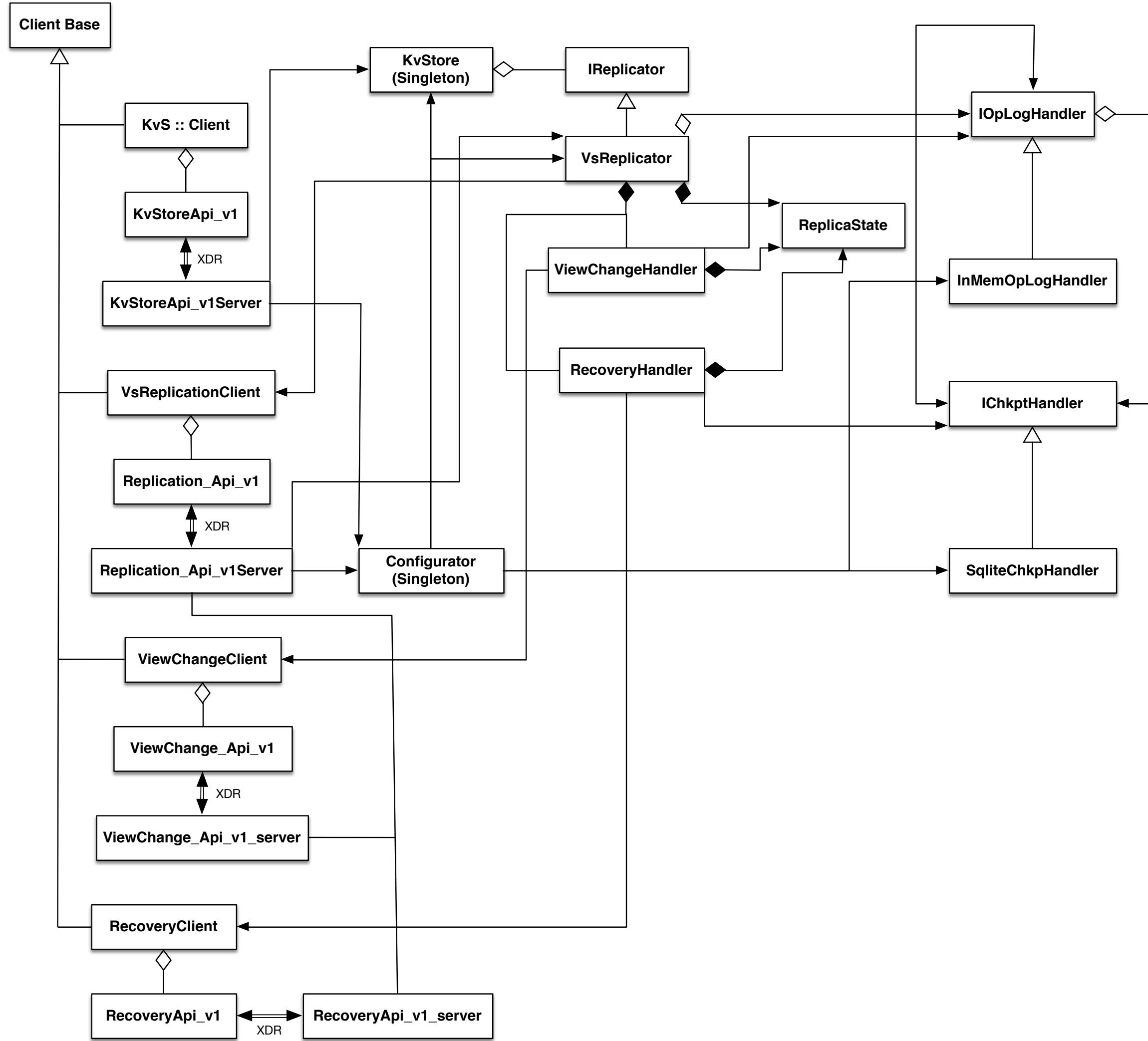

Figure 3 – Class Diagram

Figure 3 provides the class diagram of the system. From the class diagram, it is evident that the system is designed to be extensible and maintainable. The classes only use the required abstract interfaces instead of the actual implementing class. For example, the **KvStore** singleton class just composes of an instance of **IReplicator** and invokes the methods through the abstract instance. This kind of design facilitates changing the underlying implementation with minimal breakage to the dependent modules. For example, instead of binding a **VsReplicator** instance to the IReplicator in KvStore, tomorrow we can implement a new replicator, say RAFT [7], deriving

from the IReplicator and bind that to the KvStore. With minimal change, we can completely switch to a new replication implementation. Similarly **VsReplicator** just composes of an **IOpLogHandler** instance. So it is easy to switch to some other log handler instead of **InMemOpLogHandler**. Similar case holds for IOpLogHandler and **IChkptHandler**. VsReplicator composes of a **ViewChangeHandler** and **RecoveryHandler** instance. **ReplicaState** represents the replicator state that is shared between VsReplicator, ViewChangeHandler and RecoveryHandler. The **Configurator** is the entry point for the system. It is a singleton that parses the input configuration and maintains the configuration information. It does the bootstrap process, where it starts of the **KvStore** singleton class and registers a **VsReplicator** instance with it. The replicator is registered with an instance of **InMemOpLogHandler** and the logger is registered with an instance of **SqliteChkptHandler**. The **kvs::Client** class composes of the XDRPP generated **KvStoreApi_v1** class and provides the wrappers to the client APIs to mange the KV store. Similarly **VsReplicationClient** wraps **ReplicationApi_v1**, **ViewChangeClient** wraps **ViewChangeApi_v1** and **RecoveryClient** wraps **RecoveryApi_v1**. The corresponding backend server classes are **KvStoreApi_v1Server**, **VsReplicationApi_v1Server**, **ViewChangeApi_v1Server** and **RecoveryApi_v1Server**, which upon receiving the RPC requests, calls the corresponding implementing functions in other backend modules. All the *Api_v1 and *Api_v1Server classes are auto-generated by XDRPP using the XDR IDLs.

Implementation

This system is implemented using C++11 on Linux based platform. Source code can be accessed at https://github.com/pkvijay/metaDR. Client to server communication and all communications between the replicas are performed using XDRPP [6]. IDL in kvStore.x provides the client facing APIs. The backend IDLs are vsReplication.x, viewChange.x, recovery.x and vrTypes.x. IDL in vrTypes.x declare the common types that are shared between all the backend modules. Each replica server runs two socket listener threads, one to process kvStore RPCs and the other to process the replication RPCs. NOTE that the replications RPCs include the viewChange and the recovery RPCs. The VsReplicator runs a heartbeat thread, which runs either in the primary mode or in the backup mode. In the primary mode, it sends the COMMIT message to the backups with the primary's latest commit number. As mentioned in the viewstamped replication paper, commit operation is piggybacked to send the heartbeats. In the backup mode, the heartbeat thread monitors for the primary health status. If the last commit timestamp exceeded a certain threshold, it invokes the view-change protocol. Whenever the PREPARE request is received by a backup, it pushes the prepare request to a concurrent/blocking queue. Each backup runs a dedicated thread to process the prepare requests. This thread gets woken up if the concurrent queue is not empty after which it processes the prepare request and responds back to the primary if it accepts the prepare request. KV store is thread-safe, which means that multiple clients can issue simultaneous requests to the KV store and the backend takes care of synchronizing these requests. For the complete implementation detail, please refer to the source code link.

Status

Source code for the system is at https://github.com/pkvijay/metaDR. The replication, view-change and recovery modules are complete. We wanted to complete the checkpoint module, but realized that checkpointing is strongly linked to state transfer. Once the checkpointing module trims the log, only the suffix of the log will be transferred during view-change and recovery

protocols. If any replica is lagging behind and needs the earlier log entries, it needs to be able to do a state transfer to catch up. So state-transfer and checkpointing modules have a strong dependency and both have to be implemented in order to ensure correctness of the view-change and recovery protocols.

Future Work

Even after this course completion, we are hoping to continue working on this system. There is still lot of work to be done to make this a more practical system. The main bottleneck with viewstamped replication is that the protocol involves transferring the entire log and it is a costly affair for a long running system. To address this issue, all the techniques mentioned in section 5 of the viewstamped replication [1] paper need to be implemented, the most important being the application state-transfer mechanism using Merkle tree. The reconfiguration protocol mentioned in section 7 of the paper has to be implemented to allow dynamic configuration changes without bringing down the whole system. Clean shutdown mechanisms need to be implemented to facilitate bringing down server instances for maintenance reasons. The optimization techniques mentioned in section 6 of the paper can be implemented as well. Some of them are really simple to implement, for example, reads can be allowed at the backups at the cost of consistency. The KV store is maintained as a simple hash table, where the key and value are strings. The scale of the metadata that can be stored in this system is limited by this in-memory data-structure. Techniques like prefix compression can be used to improve the KV store memory utilization and scale. The keys/nodes can also be decorated with additional properties to provide a data model similar to the one in zookeeper.

References

[1] Viewstamped Replication Revisited (http://www.scs.stanford.edu/14aucs244b/sched/readings/vr-revisited.pdf)

[2] Zookeeper (http://www.scs.stanford.edu/14au-cs244b/sched/readings/zookeeper.pdf)

[3] The Google File System (http://www.scs.stanford.edu/14au-cs244b/sched/readings/gfs.pdf)

[4] RPC Specification version 2 (https://tools.ietf.org/html/rfc5531)

[5] XDR: External Data Representation Standard (http://tools.ietf.org/html/rfc4506)

[6] XDRPP (http://www.scs.stanford.edu/14au-cs244b/labs/xdrpp-doc/)

[7] RAFT (http://www.scs.stanford.edu/14au-cs244b/sched/readings/raft.pdf)