Trek: Testable Replicated Key-Value Store

Yen-Ting Liu, Wen-Chien Chen
Stanford University

Abstract

This paper describes the implementation of Trek, a testable, replicated key-value store with ZooKeeper-like semantics. The system adapts the revisited version of Viewstamped Replication to provide availability and fault tolerance, and contains a comprehensive testing and monitoring system for simulating both node crash and network failures, and for monitoring system operation.

1 Introduction

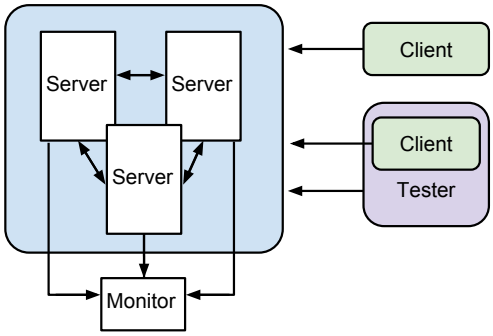

Figure 1: Trek system architecture

With Trek, we aim to build a distributed key-value store with strong consistency guarantees. To do so, we adapted the revisited version of Viewstamped Replication (VR)[1]. To ensure that our implementation provides the same level of guarantees on safety, liveness and fault tolerance as the theoretical guarantees given by VR, we put emphasis on testability in our system. As there are many possible failure scenarios, we worked on creating a system that can automatically test different scenarios and verify the correctness in each case.

As shown in Figure 1, the basic system consists of $2f+1$ replicated servers (blue rectangle) which can tolerate f faulty nodes. In normal operation, clients send key-value store requests to the primary replica in the server cluster, which are then delivered to backups and committed through VR. Each replica logs its internal operations to the monitor, allowing system operations to be observed. For testing, a tester program issues calls to server replicas to simulate node or network failures; it can also issue requests through an internal client to simulate user requests. To verify correctness, the tester program compares the VR commit logs of server replicas to check for correctness and consistency.

2 The Key-Value Store

Trek implements a key-value store with ZooKeeper[2] like semantics. The system exposes the following operations to client applications:

- create(*path*, *value*, *isSequential*): Create a node at path with *value*. If *isSequential* is true, then a sequential ID is appended to the name of the node; the ID is guaranteed to be larger than those of all other existing sequential nodes with the same name.
- delete(*path*, *version*): Delete the node at *path* if the version matches.
- exists(*path*): Return whether a node exists at *path*.
- getData(*path*): Obtain the value stored at *path*.
- setData(*path*, *value*, version): Store *value* at *path* if the version matches.
- getChildren(*path*): Get the children of the node at *path*.

As with ZooKeeper, nodes in the key-value store are versioned. getData and setData both return the version of the node in addition to the stored value. To maintain consistency between the key-value store and different clients, write operations are rejected if the version given does not match the stored version. Unlike ZooKeeper, we do not support ephemeral nodes or watches. Lastly, the key-value store has support for writing a snapshot of its state to disk and restoring its state from a snapshot.

3 Viewstamped Replication

We implemented the VR algorithm to provide consensus among server replicas. While ZooKeeper only guarantees linearizable writes and a FIFO order for client requests, VR imposes a strong consistency model, which is capable of replicating any deterministic state machine. In Trek, the implementation of VR is decoupled from the key-value store, so it is also possible to adapt our implementation to replicate other state machines.

In the revisited version of VR, the primary node is deterministically decided by the view number, which reduces the complexity of view change process. In Trek, all operations start a new transaction. While read operations may be optimized not to require a transaction to increase efficiency, we chose to keep Trek simple at this stage for testability. We implemented the exact same API as described in the paper, which improves code readability.

4 Testing and Correctness

Testability is a focus in Trek. To test the correctness of our system, we verify the theoretical guarantees provided by VR as follows:

- Safety: We compare the commit logs of all live replicas at the end of each test to make sure they are consistent. Since were replicating a deterministic state machine, if each replica has an identical log, all replicas agree on the final state.
- Fault tolerance: Safety should be maintained and the final log should be correct as long as no more than f nodes are failed at any given time.
- Liveness: View changes should eventually succeed so long as there are no more than f failed nodes, and the system should be able to commit client requests whenever there is no ongoing

view change.

Since there are many failure scenarios, it is infeasible to perform testing and correctness verification manually for all cases. Furthermore, a system for simulating node and network failures is needed in order for failure scenarios to be tested. Below, we describe a scheme we have devised to test some basic scenarios, and the system we have built into Trek to support failure simulation and automated testing.

4.1 Testing Methodology

Consider the following log for a normal transaction (3 nodes, $f = 1$, node 1 is the primary):

1. Node 1 receives request from client
2. $1 \rightarrow 2$: prepare
3. $1 \rightarrow 3$: prepare
4. $2 \rightarrow 1$: prepareOK
5. $3 \rightarrow 1$: prepareOK
6. $1 \rightarrow 2$: commit
7. $1 \rightarrow 3$: commit

To test failure scenarios for this transaction, for each of the nodes, we simulate crash-stop node failure and network partition (where the node in question is partitioned from the two other nodes) at each step. The list of scenarios to test begins as follows

1. Normal operation
2. Node 1 fails before client request is received
3. Node 2 fails before client request is received
4. Node 3 fails before client request is received
5. Node 1 fails after sending 1 RPC call
6. Node 2 fails after sending 1 RPC call
7. ...

In addition to testing failures, we also test node recovery from failure. In the case of primary node failure, we need to test three cases: if the node recovers before a view change is started by a backup node, if it recovers during the view change, and if it recovers after the start of a new view.

Clearly, even in such a small cluster, the number of scenarios to test is already quite high. Therefore, to ensure correctness, we need a way to automate testing. To facilitate the simulation of network and node failures and to allow for automated testing, we have built a testing system into Trek, described below:

4.2 Testing System

The testing system consists of an API implemented by server replicas which simulate failures at the replica, and a tester program which interfaces with the API. To facilitate the simulation of network failures, each replica keeps a *partition state*, which stores whether the replica is able to reach any given replica. The testing API exposes the following operations:

- setParitioned(*values*, *rpcCount*): After *rpcCount* outgoing RPC requests, set the partition state at the replica to that specified by *values*.

- kill(*rpcCount*): Simulate crash-stop failure after *rpcCount* outgoing requests.
- recover(): Simulate bringing a crashed replica back up.
- isAlive(): Return whether the replica has crashed.
- getCommitLog(): Get the VR commit log for this replica.

The tester program in turn uses the API to provide the following operations:

- partition(*serverList*): Create a network partition between the replicas in *serverList* and the other replicas.
- group(*serverList*): Remove any network partitions between the replicas in *serverList*.
- reset(*serverList*): Remove all network partitions involving any replicas in *serverList*. If *serverList* is empty, all network partitions are removed.
- kill(*server*): Simulate crash-stop failure at *server*.
- recover(*server*): Simulate recovery at *server*.
- health(): Return the state of all replicas.

For automated testing, the tester program contains several test routines which simulate client operations interleaved with failures at specific points. Our testing system allows us to control when simulated failures occur with respect to client and VR operations, enabling comprehensive testing of different failure scenarios.

To verify the correctness of our system, we run the test routines and compare the commit logs of different server replicas. If the commit log at every server that has not crashed or become partitioned at the end of the test is consistent with the expected log, the test is considered to have completed successfully.

To monitor the operation of our system, we implemented a monitor daemon to which each replica logs its internal operations. The monitor collects the logs and emits them to standard output.

5 Implementation

We implemented Trek in Java. This has allowed us to utilize various libraries, including those for RPC and multi-threading. The whole system, including the testing components, is around 2000 lines of code, which is a surprisingly small number considering the complexity of a distributed system. In addition, the use of Java makes operating heterogeneous server clusters possible.

5.1 Communications

All communications in Trek are implemented as RPCs using jsonrpc4j[3], with method calls serialized into JSON format and passed to callees in HTTP POST requests. Each server replica has handlers for three RPC interfaces, mounted at different URLs: *DStoreService* for handling client requests, *DStoreInternal* for VR, and *DStoreTesting* for the testing system. Client requests are encoded as *StoreAction* objects and sent to the primary replica, which returns *StoreResponse* objects containing return values and any errors. Messages between replicas are implemented as RPCs, with message contents being passed as arguments. For the RPC server, we chose to use Jetty[4] to host RPC over HTTP requests.

To simulate node crashes, we remove the handlers for client requests and VR; to simulate network

partitions, we leverage jsonrpc4j to return local proxy objects with no-op service stubs from the RPC client when trying to communicate with a partitioned node.

The RPC server at each replica uses multi-threading to support concurrent RPC requests. Common VR operations such as view change and log commit requires waiting for $f + 1$ receipts. To support this, we utilize semaphores initialized as $-f$ permits. Whenever there is an incoming receipt, we release a permit. A waiting thread acquires the permit when one becomes available and continues executing the VR protocol.

5.2 Data Storage

The key-value store is stored in memory as a tree map data structure in which the keys are the path to a node and the values are data structures containing the node value and version information. A tree map is used to allow for efficient querying of child nodes and calculation of the ID to assign to a sequential node. When a snapshot is taken, the entire tree map is serialized and written to disk.

6 Discussion

Trek is a distributed key-value store that provides strong consistency guarantees. There are some other projects providing similar interfaces; however, most of them are optimized for other applications.

Memcached[5] and Redis[6] are the two popular distributed memory object caching systems. While they support all the operations of Trek, there is no guarantee of safety in case of node or network failure. However, since the only possible issue is missing cache data, it can be resolved like a cache miss.

ZooKeeper is another system that provides key-value storage. As discussed above, ZooKeeper has a relaxed consistency constraint and relies on node version numbers for conflict resolution. Trek differs from it by providing safety with identical commit logs replicating a deterministic state machine. This makes Trek suitable for applications where consistency is critical, such as configuration management and managing shared resources.

Trek is designed and implemented with testability in mind. As a result, we built the system as separate testable modules, which have the added benefit of being more easily reusable. This has led to a more structured codebase, which would also adapt better to test driven development (TDD).

7 Future Work

In the current implementation of Trek, server cluster configuration is manually passed to each server and client as command line arguments. We hope to eventually replace this with a system where it is read from a configuration file. Furthermore, there is currently no support for cluster reconfiguration. Implementing a reconfiguration protocol would make Trek clusters more flexible, making it unnecessary to restart the cluster to add or remove servers.

The implementation of VR in Trek currently performs node recovery by requesting all committed operations from the primary replica. We hope to instead make use of snapshots of the key-value store to reduce the amount of data transfer required.

We would also like to improve the efficiency of Trek. Currently, the bottleneck in normal

operation is the cost of RPCs. In addition to the optimization of not using transactions for reads mentioned earlier, we can potentially increase system throughput by using a more sophisticated RPC library and more compact object serialization (e.g. by using Protocol Buffers[7]).

8 Conclusion

In this paper, we have described the design and implementation of Trek, our testable replicated key-value store. We discussed our methodology for testing different failure scenarios, and we also discussed the design of the testing system, which allows us to simulate failures, observe server behaviour, and automate testing of failure scenarios. Finally, we compared Trek to other key-value stores and proposed several improvements that can be made to improve the usefulness and efficiency of Trek.

References

[1] Barbara Liskov and James Cowling. *Viewstamped Replication Revisited*. Tech. rep. MIT-CSAIL-TR-2012-021. MIT, July 2012.

[2] Patrick Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. Boston, MA: USENIX Association, 2010.

[3] `https://github.com/briandilley/jsonrpc4j`.

[4] `http://eclipse.org/jetty/`.

[5] Brad Fitzpatrick. "Distributed Caching with Memcached". In: *Linux J.* 2004.124 (Aug. 2004), pp. 5–. ISSN: 1075-3583.

[6] `http://redis.io/`.

[7] `https://developers.google.com/protocol-buffers/`.