FAULT-TOLERANT REMOTE EXECUTION OPERATORS FOR THE REACTIVE EXTENSIONS LIBRARY

Master Thesis

Submitted to the Delft University of Technology in accordance with the requirements of the *MASTER OF SCIENCE* degree in the Faculty of Electrical Engineering, Mathematics and Computer Science.

by

Mircea-Raul VODA

Supervisor: Prof.dr. H.J.M. Meijer, TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science
DELFT UNIVERSITY OF TECHNOLOGY

January 2018

ABSTRACT

The continuous shift of various industries towards internet-based services have caused an exponential growth in the amount of data produced over the past few years. On top of this, the increasing need for real-time analytics and the increase in data velocity have made asynchronous, event-driven applications the norm. In this context, the reactive programming paradigm has gained much traction as it focuses on the propagation of change and composing/transforming streams of data. The industry standard reactive programming library for the JVM, .NET and Javascript ecosystems is the Reactive Extensions (Rx) library. However, despite being well equipped to deal with asynchronous data, it does not offer any way of scaling the computation on multiple machines. In this thesis, we attempt to lay the groundwork for a scalable Rx library by implementing infrastructure and operators for remote execution of Rx streams.

Contents

1

INTRODUCTION

1

1.1. INTRODUCTION

The growth of the Internet since the turn of the millenium and the increasing number of applications and services that have millions or even billions of users have given birth to a new domain area dedicated to managing and processing "big data".

As a term, big data has been used since the 90s[1], with a number of research papers[2–4] attempting to provide a formal definition and identify challenges and opportunities. In a nutshell, the term refers to data sets that cannot be handled by traditional software tools in a tolerable amount of time, due to their size, complexity or velocity. Common challenges in this area include effective storage, processing, querying and visualization.

Thus, modern applications typically have to be able to process varying amounts of data that arrive from different sources (often times in differing formats: structured or not) and with different velocities. In his 2012 paper "Your mouse is a database"[5], E.Meijer argues that along each of these 3 dimensions of volume, velocity and variety we find a well-known database technology.

For example, traditional RDBMS systems handle relatively small sets of well-structured data, which is consumed in a pull manner. Systems based on MapReduce[6] such as HBase[7] also handle well-structured data in a synchronous/pull fashion, but this time the system is able to handle large amounts of data by distributing it on a cluster of machines.

On the other hand, the increase in data velocity and the rising need of real-time analytics have brought push-based, event-driven systems into the limelight. This led to the rise in popularity of the reactive programming paradigm which revolves around asynchronous data streams and the propagation of change and is well-suited for the development of event-driven applications.

One such example is the reactive extensions (Rx) library which enables users to coordinate and orchestrate asynchronous push-based computations. Currently, there are a number of implementations of Rx in different programming languages including C#, Java, Scala, Clojure, Javascript and Kotlin, among others.

Rx offers a large number of operators that allow developers to process streams of asynchronous events. Additionally, many implementations of Rx use Schedulers to control an Observable's transition between threads in a multi-threaded environment. This gives developers the option of running multiple Rx streams in parallel. However, parallelizing an application by executing it on multiple threads is becoming insufficient in this day and age.

Over the past few years, the constant increase in data volume coupled with stalling processor speeds and I/O capabilities have caused a shift in the development of computing systems. In order to keep up with this growing volume of data, produced by either the web, business systems or scientific instruments, applications are forced to scale out to distributed systems.

While certainly appealing from a potential benefits point of view, the development of distributed applications comes with its own set of challenges and pitfalls. The first big challenge is finding an adequate programming model that is both expressive and well-equipped to deal with parallelism and asynchronous events. The second important challenge, characteristic to the distributed environment, is designing a system that can handle faults.

We believe that the Rx programming model is a good answer to the first challenge, being built from the ground up with the purpose of handling asynchronous, push-based event streams. However, out of the box it only offers parallelism via multi-threading, by using the Scheduler abstraction. This thesis aims to take the first steps towards executing an Rx program on multiple machines, by extending the library with operators for fault-tolerant remote execution. The addition of such operators would facilitate Rx users to look towards scaling Rx computations across clusters.

1.2. BACKGROUND

The reactive extensions library defines a set of abstractions and operators that enable users to fluently create, compose and consume asynchronous, push-based data streams. From now on, whenever we refer to Rx, we specifically refer to its RxMobile flavor, which is a lightweight implementation in Java and Scala, targeting the Android environment.

The main abstractions defined by the library are the Observable<T> and Observer<T> interfaces which are used to model asynchronous data streams with values of type T.

The Observable represents a source of events ordered in time and can emit three different types of events: a value, an error or a signal that the stream stream is completed. An additional restriction placed on the semantics of the stream is that the stream ends when an error or completed event is sent. This practically means that a stream consists of zero or more values followed by either an error or a completed event.

The events emitted by an observable are captured asynchronously with the help of an Observer, which represents a consumer of the stream. Observers are typically created by defining the functions that will execute when a value, error or completed event is emitted. Implementations of an observer can be assumed to be synchronized, which means that a maximum of only one event will be treated at any given time. In the Rx world, connecting the stream of events with its consumer(s) is called subscribing, and it will cause the events to be sent to the subscribed observers.

With this in mind, let's look at the code snippet below that shows an example Rx query where lines is an Observable<String> that reads a file line by line.

```
val lines: Observable[String] = Observable.create(...)
lines.flatMap(line => Observable(line.split(" "))
   .scanLeft(0)((count, word) => count + 1)
   .subscribe(println, Console.err.println, () => println("Completed")
```

The query in this example implements the by now famous word count example. Every line received by the flatMap operator is split at the space character and the resulting words are pushed as a new Observable to the scanLeft operator. For every word received, the scanLeft operator increments a counter that starts with an initial value of 0 and pushes the updated counter value to the stream observer that will print each event received on the screen.

One thing to note is that Observables in Rx are lazy and by default execute synchronously on a single thread. When a new Observable is created, no events are pushed until an observer subscribes and when that happens, any event produced is sent in a

synchronous fashion, on the same thread on which the subscribe method is called. This means that in the example above, the subscribe function will block the thread until the whole file is read from disk or until an error is raised.

Asynchronous execution and multithreading is supported in Rx via operators that instruct observables to operate on particular Schedulers. The two operators in case are subscribeOn, which instructs the stream to run the create function on a scheduler and observeOn, which instructs the stream to send events to its observers on a specific scheduler.

The previously presented example could be parallelized by having the processing of each line execute on a different thread. This could be easily achieved by changing the flatMap lambda to:

```
line => Observable(line.split(" ").observeOn(NewThreadScheduler())
```

In this case, the work of actually splitting the line into words is still executed on the thread that called the subscribe function, but the rest of the operators in the stream will execute on a new thread.

If we consider the example presented above, one could gain significant speedup by executing parts of the query in parallel, on multiple threads. Analogously, one could imagine use cases where it would be useful to execute parts of an expensive query in parallel, but this time on multiple remote machines.

1.3. GOALS AND CONTRIBUTIONS

The main goal of this work is to show a possible approach to designing and implementing a system that extends the Rx library with operators for fault-tolerant, remote execution, while maintaining an interface consistent with the vanilla Rx library.

Such operators have multiple potential uses. Out of the box, they enable users to write long-running, fault-tolerant queries that integrate seamlessly with the rest of the Rx API. Additionally, they allow developers to parallelize parts of an Rx stream over multiple machines, which can bring significant speedup in certain classes of applications. Lastly, these operators have potential to be used as building blocks for higher-level abstractions such as cluster schedulers that efficiently distribute a stream over multiple machines and coordinate its execution.

In order to achieve this goal, the primary contributions of this thesis are:

- A novel implementation of the Raft consensus algorithm, based on the Rx libray
- A extension to Raft that allows clients to send and execute arbitrary code on the remote statemachine.
- An extension to the Raft consensus algorithm that allows a cluster to forward messages to another cluster on behalf of the client. This has the benefit of freeing the client from having to act as a message broker between the two clusters.
- The implementation of a replicated statemachine capable of executing Rx streams.

- Orthogonal extensions to the Rx library that allow users to interact with remote streams in a way similar to standard Rx operators.

REFERENCES

[1] S. Lohr, *The origins of 'big data': An etymological detective story,* http://bits.blogs.nytimes.com/2013/02/01/the-origins-of-big-data-an-etymological-detective-story/ (2013).

[2] D. Laney, *3d data management: Controlling data volume, velocity and variety,* META Group Research Note (2001).

[3] A. De Mauro, M. Greco, and M. Grimaldi, *A formal definition of big data based on its essential features,* Library Review **65**, 122 (2016).

[4] M. Beyer, *Solving 'big data' challenge involves more than just managing volumes of data,* http://www.gartner.com/newsroom/id/1731916 (2011).

[5] E. Meijer, *Your mouse is a database,* Queue **10**, 20 (2012).

[6] J. Dean and S. Ghemawat, *Mapreduce: simplified data processing on large clusters,* Communications of the ACM **51**, 107 (2008).

[7] *Apache hbase,* https://hbase.apache.org/.

2

Related Work

The amount of data produced has been growing significantly over the past few years, which in turn has driven up the demand for data processing frameworks. Additionally, the increasing need for real-time analytics, driven by internet-based services, has shifted the focus from batch processing to stream processing systems. Currently, there are a number of well-known stream processing frameworks, the most notable of which are: Apache Spark, Apache Flink, Apache Storm, and Google Millwheel. This chapter will take a look at these frameworks' features and their approach in dealing with faults.

2.1. APACHE STORM

Apache Storm[1] is a real-time, distributed and fault-tolerant stream processing framework that aims to make it easy to reliably process unbounded streams of data.

Applications in Storm are designed as topologies[2], which are directed graphs with vertices representing computation and edges representing streams of data (represented as tuples) flowing between computation nodes. Graph vertices in a Storm topology can be one of two types: either spouts or bolts. Spouts are stream sources for the topology and read data from an external source (typically queues such as Kafka[3] or Kestrel[4]) and emit it into the topology. On the other hand, bolts represent the computation part of the stream and are responsible for processing incoming tuples and passing the results to the next set of bolts downstream.

Clients interact with the system by submitting topologies to a master node (called the Nimbus), which is responsible for distributing and coordinating the execution of a topology. The computation vertices of a topology are run on worker nodes, which are used to execute on or more worker processes. Each worker node also runs a supervisor process that is responsible for communicating with the master node. The supervisor's job is to receive assignments from the master node, spawn worker processes and monitor their health.

From a fault-tolerance point of view, both the Nimbus and supervisors are fail-fast and stateless and all their state, as well as the coordination between them is managed by Zookeeper[5] or kept on local disk(s). This makes sure that in case of master node failures the workers will continue to make progress, and in case of worker failures they can be restarted by the supervisor processes. However, Nimbus is still a potential point of weakness in the system, since if it is down, users will not be able to submit new topologies and running topologies can't be re-assigned to different machines.

Out of the box, Storm provides two types of semantic guarantees: at least once, or at most once message processing semantics. Using the at most once processing semantics, tuples entering a topology are either processed once or not at all, while using at least once semantics the tuple is guaranteed to process once, but it might also be processed multiple times. This potential shortcoming is addressed by Trident[6], which is an alternative interface to Storm that offers exactly once message processing semantics.

2.2. APACHE SPARK AND SPARK STREAMING

Apache Spark[7–9] is an open-source cluster computing framework that builds upon frameworks like MapReduce[10] and Dryad[11] by introducing a data structure that enables efficient data reuse in a broad range of applications. This abstraction is called a resilient distributed dataset (RDD) and is an immutable, fault-tolerant, partitioned collection of records that allows users to explicitly persist intermediate results in memory. Most cluster computing frameworks based on MapReduce write their intermediate results to stable storage (e.g. a distributed file system), which incurs significant overheads due to serialization, replication and disk I/O. Keeping intermediate results in memory enables Spark to be significantly faster than MapReduce for iterative algorithms, which typically visit their dataset or partial results multiple times.

Spark defines a rich set of operators on RDDs that allows users to transform data or control RDD partitioning to optimize data placement. RDDs can only be created through deterministic operations on data in stable storage or other RDDs. Additionally, each RDD keeps information about how it was created/computed (called its lineage), which allows the framework to recompute partitions starting from data in stable storage.

At their core, RDDs are targeted at batch applications that apply the same operations on a whole dataset. On top of this data abstraction, Spark provides a number of libraries that provide support for structured and semi-structured data [12], machine learning[13] and graph processing[14].

On an architectural level, Spark relies on a distributed storage system(such as HDFS, Cassandra or Amazon S3) and a cluster manager (either the Spark native cluster manager, Apache Mesos[15] or Hadoop Yarn[16]). Spark users write their applications in a functional programming style, by invoking operations such as map, filter and reduce on RDDs. The execution of these operators is scheduled by the cluster manager on Spark's worker nodes. The worker nodes have the responsibility of receiving data, storing partitions of input/RDDs and executing tasks.

The scheduling strategy is very similar to Dryad's and it will try to assign tasks to machines based on data locality. If a job needs to process data that is available in memory on a certain machine, then the job will be scheduled on that machine.

Fault-tolerance is achieved by keeping track of an RDD's lineage. If a job fails, but the input RDDs are still available, then it will be re-run using the existing inputs. In the worst case scenario, missing RDDs will be recomputed using data from stable storage.

Natively, Spark is a framework for batch data processing, but it does provide the Spark streaming[17] extension for streaming analytics. The main abstraction behind this extension are discretized streams (D-streams), which structure a streaming computation as a series of micro-batch computations. D-streams are built on top of RDDs, which enables Spark streaming to leverage Spark's scheduling and fault-tolerance mechanisms, but comes with a latency penalty equal to the micro-batching interval.

2.3. GOOGLE MILLWHEEL

Google Millwheel[18] is a scalable, low-latency and fault-tolerant stream processing framework. Computations in Millwheel are expressed as directed graphs, where nodes represent the user's computation logic and records are delivered continuously along the edges of the graph. The Millwheel system manages fault-tolerance, persistent state and the continuous flow of records between nodes and guarantees exactly once message delivery from the user's perspective.

From an architectural point of view, each computation in the graph can run on one or multiple machines, with load-balancing and distribution handled by a replicated master. Computations are split in a set of key intervals and assigned to sets of machines. In response to varying system load, these key intervals can be moved around, split or merged, depending on the needs of the system. Any persistent state in the system is saved to a reliable database such as Bigtable[19] or Spanner[20]. Failure recovery is handled by reading metadata from this backing store whenever a key interval is assigned to a new owner.

One of the contributions of Millwheel is the implementation of a low watermark system, which is used to provide a bound on the timestamps of future records arriving at a computation. The low watermark system allows the user to determine if he has a complete picture of the data up to that time. For data consistency, this system is implemented as a central authority and tracks all low watermarks values and writes them to reliable storage. Due to the usage of low watermarks, Millwheel does not require a strict monotonicity for inputs and can process out-of-order data.

2.4. APACHE FLINK

Similar to Spark, Apache Flink[21, 22] is an open-source system for processing both streaming and batch data. In Flink, a program is a directed acyclic graph of stateful operators connected by data streams.

However, in contrast to Spark, where both the batch and stream processors were running on top of a batch processing engine, Flink does the opposite: it is a system based on a streaming dataflow engine that is used to execute both stream and batch processing jobs. Flink programs can be written using a multitude of APIs, but eventually they will all be transformed into dataflow graphs to be executed by this engine. For efficient execution, operators can be parallelized into multiple instances and data streams can be split into partitions.

From a system architecture point of view, a Flink cluster is composed of three types of processes: the client, the job manager and at least one task manager. The client transforms the user program into a dataflow graph and submits it to the job manager, which is responsible for scheduling operators, tracking state and progress and coordinating checkpoints and recovery. Finally, the task mangers are the worker nodes in a Flink cluster; they process one or more operators and report progress to the job manager.

Regarding fault-tolerance, Flink offers reliable execution and exactly-once processing semantics. The system is built with the assumption that its data sources are persistent and replayable (e.g. files or durable message queues such as Apache Kafka[3]) and failures are handled by checkpointing and partial re-execution. In order to take consistent checkpoints of parallel operators without halting their execution, Flink uses a mechanism called Asynchronous Barrier Snapshotting[23]. This injects special control sequences (barriers) into data streams, which are then used by operators to decide when to snapshot (effectively splitting the stream into logical sub-streams).

REFERENCES

[1] *Apache storm,* http://storm.apache.org/.

[2] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, *et al.*, *Storm@ twitter,* in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (ACM, 2014) pp. 147–156.

[3] J. Kreps, N. Narkhede, J. Rao, *et al.*, *Kafka: A distributed messaging system for log processing,* in *Proceedings of the NetDB* (2011) pp. 1–7.

[4] *Kestrel: A simple, distributed message queue system.* https://github.com/twitter-archive/kestrel.

[5] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, *Zookeeper: Wait-free coordination for internet-scale systems.* in *USENIX annual technical conference*, Vol. 8 (2010) p. 9.

[6] *Trident api overview,* http://storm.apache.org/releases/1.0.0/Trident-API-Overview.html.

[7] *Apache spark - lightning-fast cluster computing,* https://spark.apache.org/ ().

[8] M. Zaharia, *An architecture for fast and general data processing on large clusters* (Morgan & Claypool, 2016).

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, *Spark: Cluster computing with working sets*. HotCloud **10**, 95 (2010).

[10] J. Dean and S. Ghemawat, *Mapreduce: simplified data processing on large clusters,* Communications of the ACM **51**, 107 (2008).

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, *Dryad: distributed data-parallel programs from sequential building blocks,* in *ACM SIGOPS operating systems review*, Vol. 41 (ACM, 2007) pp. 59–72.

[12] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, *Shark: Sql and rich analytics at scale,* in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data* (ACM, 2013) pp. 13–24.

[13] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, *et al.*, *Mllib: Machine learning in apache spark,* The Journal of Machine Learning Research **17**, 1235 (2016).

[14] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, *Graphx: A resilient distributed graph system on spark,* in *First International Workshop on Graph Data Management Experiences and Systems* (ACM, 2013) p. 2.

[15] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, *Mesos: A platform for fine-grained resource sharing in the data center.* in *NSDI*, Vol. 11 (2011) pp. 22–22.

[16] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, *Apache hadoop yarn: Yet another resource negotiator,* in *Proceedings of the 4th annual Symposium on Cloud Computing* (ACM, 2013) p. 5.

[17] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, *Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters.* HotCloud **12**, 10 (2012).

[18] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, *Millwheel: fault-tolerant stream processing at internet scale,* Proceedings of the VLDB Endowment **6**, 1033 (2013).

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, *Bigtable: A distributed storage system for structured data,* ACM Transactions on Computer Systems (TOCS) **26**, 4 (2008).

[20] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, *et al.*, *Spanner: Google's globally distributed database,* ACM Transactions on Computer Systems (TOCS) **31**, 8 (2013).

[21] *Apache flink - scalable stream and batch data processing,* https://flink.apache.org/ ().

[22] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, *Apache flink: Stream and batch processing in a single engine,* Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **36** (2015).

[23] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, *Lightweight asynchronous snapshots for distributed dataflows,* arXiv preprint arXiv:1506.08603 (2015).

3

THE STATE MACHINE APPROACH

Distributed systems are often structured in terms of services and clients. Typically, each service runs on one or more servers and exposes operations that the clients can invoke via network requests. Obviously, the most simple way to implement a service is to use a single, centralized server, but this approach is only as fault tolerant as the physical machine used to host that service. If this level of fault tolerance is not acceptable, a fault tolerant version of the service can be implemented by replicating the server and running replicas on multiple isolated machines that can fail independently. This general solution is called the state machine approach[1] (or state machine replication) and, at a high-level, is a method of obtaining a fault tolerant service by replicating servers and coordinating client interactions with server replicas.

3.1. STATE MACHINES

Services, servers and most programming structures for supporting modularity define state machines. A state machine is a mathematical model of computation that consists of state variables (which encode its possible states) and commands which are used to transform its state. A state machine can be in exactly one of its possible states at any given point in time and its commands can change the state variables and/or produce some output. The state machine commands are implemented by deterministic programs and each executes atomically with respect to other commands.

Formally, a state machine is defined by the tuple *(S, I, O, Tf, Of, Start)* where:

- `S = set of possible states`
- `I = set of inputs`
- `O = set of outputs`
- `Tf = transition function (Input x State -> State)`
- `Of = output function (Input x State -> Output)`
- `Start = initial starting state`

A client interacts with a state machine by sending a request to execute a command (Input). The request should contain information about the command that needs to be executed and any data that might be needed by the state machine during execution. After executing the command, the state machine might change its internal state and/or produce output to another system, another peripheral device, or to clients awaiting responses from previous commands. In a nutshell, a state machine defines a deterministic computation that reads and processes a stream of requests and occasionally produces some output. An important characteristic of state machines is that their output are completely determined by the sequence of Inputs processed, independent of time or other factors.

3.2. FAULT TOLERANCE WITH REPLICATED STATE MACHINES

A fault-tolerant version of a state machine can be implemented by replicating it and running a replica on multiple isolated physical machines in a distributed system.

The basic idea is that, if each replica starts in the same Start state, executes the same commands in the same order (triggered by the same sequence of Inputs), they will each do the same thing and end up in the same State while generating the same Outputs. Intuitively, a fault in one of the replicas would be noticeable as a difference in the internal State or in the Outputs produced compared to the other (non-faulty) replicas. This is only true if the state machine is deterministic. If it is not, we couldn't know for sure if the variance in State/Outputs is caused by faults or non-determinism.

In order to tolerate a number of failures f, an ensemble implementing a replicated state machine must contain at least 2f + 1 replicas. In this case, the final output of the replicated state machine is the Output produced by the majority of the replicas. This means that the minimum number of replicas needed for fault tolerance is 3, in which case the system can tolerate 1 failure and the other 2 replicas are used to establish the correct State and Output. Two replicas are not enough since there is no way to tell which replica is faulty and which one is not.

Popular examples of replicated state machines include Chubby[2] and Zookeeper[3] which are systems that provide hierarchical key-value stores and synchronization primitives. They are typically used for storing configuration data and coordinating clients.

However, replicated state machines are just the basic building block for making a system fault-tolerant. They can be employed in a number of different ways, depending on the usage scenario.

The most common setup involves a replicated state machine (typically made out of 3 or 5 servers) which other nodes in the system use to coordinate their activities. This is commonly used by systems that provide configuration management or distributed locks [2, 3].

Another common setup is where a leader is used to coordinate the other nodes in the system and the replicated state machine is used to store critical meta-data and coordinate leader elections. This approach is typically used in large-scale storage systems that rely on a single cluster leader like GFS[4], HDFS[5] and RAMCloud[6].

In the case of systems that replicate very large amounts of data[7–9], a third approach is used, where data is partitioned across many replicated state machines and any operations that span multiple partitions use a two-phase commit protocol to maintain consistency.

3.3. CONSENSUS

Having all state machines start in the same Start state is trivial, which means that the key to obtaining a fault-tolerant state machine is to ensure that all replicas receive and process the same sequence of Inputs. This challenge, also known as consensus, is a fundamental problem in fault-tolerant distributed systems and is faced by all systems that need to be consistent while also providing high levels of availability.

Replicated state machines are typically implemented using a replicated log that is used to store the Inputs before their appropriate commands are executed. In order for the state machines to process the same sequence of commands, all the logs must contain the same Inputs, in the same order. That is achieved with the help of a consensus algorithm.

Figure 3.1 shows a typical architecture of a replicated state machine running on 3 servers. From a high-level viewpoint, the consensus module runs on each replica and receives Input from clients, stores them in the local log, coordinates with the consensus modules of the other replicas to make sure that every log contains the same Inputs and, once the Inputs are properly replicated, they are applied/executed on the local state machine. Each state machine will process the Inputs in log order and any Outputs produced are returned to the clients. As a result, the group of servers running replicas appear to form a single, highly-reliable state machine.

The consensus modules ensure that the replicated state machine never returns an incorrect result under non-Byzantine conditions (network delays, network partitions, packet loss, message duplication and message reordering). They also guarantee that the replicated state machine is available as long as a majority of servers are operational and able to communicate with other servers and clients. Lastly, as a side-effect, the consensus module diminishes the impact of slow machines, since a command can complete (and an Output can be returned) as soon as a majority of the cluster has finished pro-

cessing it.

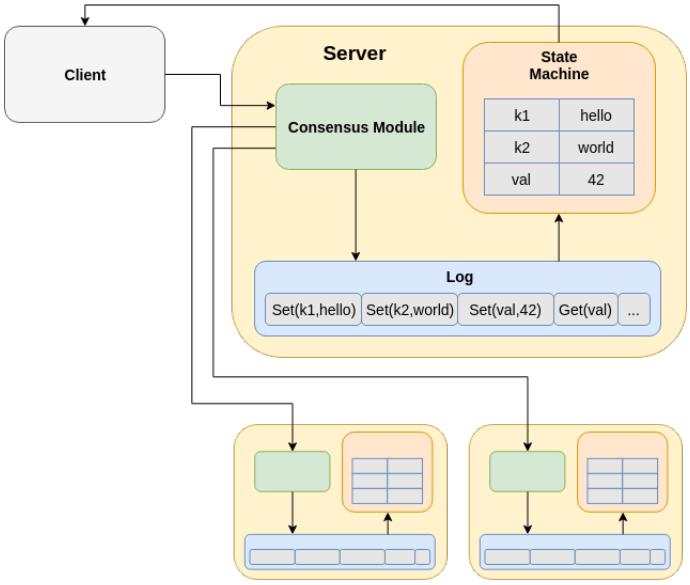

Figure 3.1: Replicated state machine architecture

Over the last two decades, a number of consensus algorithms have been developed with the most well-known being Paxos[10, 11], Viewstamped Replication[12–14] and Zab[15]. Paxos has been by far the most popular, but each of these options are notoriously difficult to understand and implement correctly. As a result, Raft[16, 17] was developed as a fully-featured consensus algorithm focused on understandability and ease of implementation. Because of its ease of understanding, Raft was the preferred choice as the basic infrastructure of my system. I developed a Raft implementation in the Scala [18] programming language using the gRPC[19] framework for network communication and RxMobile (a lightweight alternative to RxJava[20]) for handling asynchronous event streams.

3.4. THE RAFT CONSENSUS ALGORITHM

Raft implements consensus by first electing a leader and then giving the leader responsibility for keeping the replicated log consistent. The Leader is the only server in the cluster that can accept requests/Inputs from clients, which it then appends to the log and replicates to the rest of the cluster. Once the leader detects that a log entry has been successfully replicated, it applies the entry to its state machine and instructs the other servers to apply it to their state machines. Using this approach, the consensus problem is split into 2 logical steps: leader election and log replication.

A Raft cluster contains several replicated servers, each of them being in one of three possible states: Follower, Candidate or Leader. Under normal circumstances, there will be exactly one Leader, while all other members of the cluster will be Followers. Servers

in the Follower state only wait and respond to requests from Leaders and Candidates. They are otherwise passive and issue no requests on their own. As mentioned before, the Leader handles client requests, while the Candidate state is used only during the leader election process.

Raft divides time into terms of arbitrary length, identified by consecutive integers. The beginning of a new term is determined by an election, when one or more candidates attempt to become leader. Whichever candidate wins the election will serve as leader for the rest of that term. Internally, each server stores a current term variable, which increases monotonically over time and acts as a logical clock [21]. These term variables are exchanged and compared whenever servers communicate, which allows cluster members to detect obsolete information such as stale leaders or delayed requests from previous terms. Whenever a server finds that its term is out of date (meaning there is at least one server in the cluster with a higher term number), it will immediately update its term to the higher value and revert to the Follower state. Any requests from servers with a stale(lower) term number are also immediately rejected.

3.4.1. LEADER ELECTION

All servers in a Raft cluster begin in the Follower state and remain in that state as long as they receive messages from a Leader or Candidate. Leaders maintain their authority by periodically sending Heartbeat messages to the rest of the cluster. Each server keeps an internal timer that they use to keep track of how much time has elapsed since they last heard from a Leader/Candidate. If a certain period of time (called *election timeout*) elapses without any communication, the Follower will assume there is no current Leader and will attempt to become Leader themselves by starting the leader election process.

To start an election, the Follower increments its current term, transitions to the Candidate state, votes for itself and sends vote requests to the other cluster members. The server will remain a Candidate until it either:

- wins the election
- discovers another Leader
- another election timeout elapses

To win an election, a Candidate must receive votes from a majority of servers in the cluster. Each server is allowed to vote for at most one candidate in any given term, and it does so on a first-come, first-served basis. Once a candidate receives a majority of votes, it transitions to the Leader state and begins sending Heartbeat messages. Whenever a Follower receives heartbeats, it will reset their election timeout. This makes sure that the Raft cluster does not start elections while a Leader is still operational.

While waiting for votes, a Candidate might receive a message from a Leader. In this case, the candidate compares its term with the Leader's and if the Leader's term is at least as large, then the Candidate will recognize the Leader as legitimate and stops the election process by reverting to the Follower state. If the Leader's term is smaller though, then the candidate will reject the message and continues the election.

There's also a possibility that a Candidate neither wins nor loses an election, which can happen if multiple servers become Candidates at the same time and the votes get split in such a way that no Candidate obtains a majority. When this happens, the election

timeout will eventually trigger again, in which case each candidate starts a new election by incrementing its term and requesting new votes from the cluster. In order to prevent this split-vote situation from repeating indefinitely, Raft uses randomized election timeouts which practically force Candidates to time out and start new elections at different times. This effectively ensures that, in most cases, a single server will time out and win the election before any other servers time out.

3.4.2. LOG REPLICATION

Once a Leader has successfully been elected, it can begin servicing client requests. Each request contains a command to be executed by the replicated state machine and, as soon as it is received by the leader, it will be appended to the Leader's log and sent to the other cluster members for replication.

Each log entry stores a state machine command along with the term number when the entry was received by the Leader (the term is stored to detect any inconsistencies between logs on different server). When the entry has been replicated, the Leader applies the entry to its state machine and returns the result to the client. A log entry is safe to apply to a state machine when it has been *committed* and this happens only once the leader that created the entry has successfully replicated it to a majority of the servers in the cluster. If an entry at index n in the log is committed, this also commits all entries up to index n.

The leader also keeps track of the highest index known to be committed, and will include this number in all messages sent to followers. This makes sure that followers also find out which entries are committed and are safe to apply to their state machines. In case multiple entries are committed at once, they will be applied to the state machines in log order.

The Raft algorithm was designed in such a way as to guarantee that if two log entries on different servers have the same index and term, then they store the same command and both logs are identical in all entries up to that index. During normal operation, the logs of the Leader and Followers always stay consistent, but this can change in the case of Leader crashes. For example, a Follower might be missing entries from the Leader, might have extra entries compared to the Leader, or both. Such situations are solved in Raft by having the Leader always force Followers to duplicate his own log, so in the case of conflicting entries, the Follower will override his entries with the ones received from the Leader.

3.4.3. FAULT TOLERANCE

The previous steps for leader election and log replication are still not sufficient to ensure that all state machines execute the same commands in the same order. Using the previous rules, it is still possible to have a Follower unavailable for a period of time, then be elected Leader and rewrite previously committed entries from the log. This is especially dangerous since committed entries are considered to be safe to apply to the state machine and overriding these entries would corrupt the state of the system.

This issue is solved by adding an additional restriction to the election process: a server will not grant a vote to a Candidate that does not have a log that is at least as up-to-date than its own. To determine which log is more up-to-date, Raft compares the

index and term of the last entries. The log with the higher term is more up to date and if the terms are equal, then whichever log is the longest is more up to date. This rule, coupled with the fact that a Candidate must receive a vote from a majority of servers to become Leader, ensures that any new Leader will contain all previously committed entries (since committed entries are, by definition, present on the majority of servers).

In the case of Follower or Candidate crashes, any messages sent to the crashed servers will fail and Raft handles this situation by retrying indefinitely. Once the affected servers restart, they will eventually receive the messages from the Leader and continue normal operation. If a failure occurs after receiving a message, but before responding, then the server in case will receive the same message again after restarting. This is not a problem, since Raft messages have the same effect if they are repeated.

To correctly handle crashes and restarts, Raft servers must persist enough state to stable storage to ensure that the state of the system cannot be corrupted. This means that all servers must persist their current term and votes (otherwise, a server could vote twice in the same term or overwrite log entries from newer leaders with entries from defunct leaders). Each server also needs to write newly appended log entries to stable storage. Leaders must do this before sending them to the rest of the cluster for replication and Followers before responding to Leaders. This makes sure that any committed entries can not be lost when servers restart. One thing to note is that if a server loses any of this information, it cannot safely rejoin the cluster under the same identity.

3.4.4. IMPLEMENTING EXACTLY-ONCE SEMANTICS

With the algorithm described so far, Raft provides at-least-once semantics to its clients, which means that it is possible for the replicated state machine to apply the same command multiple times.

Such a situation may arise if a client submits a command to the Leader which appends the command to its log, commits it and applies it to the state machine, but crashes before responding to the client. Since the client does not receive a response, it will eventually retry with the same command, in which case the new Leader appends it, commits it and also applies it to the state machine. Since both commands have been committed, they will both be applied and even though the client intended for the command to be executed once, it is actually executed twice. This is especially problematic if the commands applied to the state machine modify its internal state. In that case, the system can enter a corrupt state, or produce incorrect results which are difficult for clients to detect or recover from.

This issue is very common and is encountered by most distributed systems. In order to offer linearizable semantics [22], where each operation appears to execute exactly once, Raft servers must be able to filter out duplicate requests. The basic idea is to have the servers save the results of client commands and re-use these results instead of executing the same request multiple times. This is implemented by adding a new logical step in the communication flow between a client and the Leader. Before sending any command, a client needs to register itself with the Leader, which will assign a unique identifier to each client. The clients will then use this identifier in any future communication with the Leader.

Clients are also responsible for assigning a unique sequence number for each com-

mand they submit. Each server state machine maintains a session object for each client, which tracks the latest sequence number processed, along with its associated response. Whenever a server receives a command, it will first check the session and if it has already been executed, it will respond immediately with the stored response, without re-executing the request. If it is a new command, it gets applied to the state machine, and the new sequence number and response are stored in the session and old values discarded.

This approach can also be extended to allow multiple concurrent requests from the same client by having the session track a set of sequence number and responses. To make this work, the client must include the lowest sequence number for which it has yet to receive a response in each request. The state machine can then use this number to discard responses for lower sequence numbers.

While the addition of session objects provide linearizable semantics, they also introduce an additional issue: servers must now agree when to (deterministically) expire a client's session. If that does not happen, then state machines on different servers could diverge from one another. For example, the state machine could become inconsistent if one server expired the session for a particular client and then reapplied the duplicated commands from that client, while another server kept the session alive and filtered duplicates.

There are multiple possible approaches to address this issue; one would be to set an upper bound to the number of sessions and use a least-recently-used policy for eviction. Another possibility, which is also the one I used for my implementation of the system is to use an agreed-upon time source for session expiration. This works by having the leader append a timestamp to every command received and appended to the log. The timestamp gets replicated together with the command, and the state machines will use this timestamp to deterministically expire inactive sessions. This additional mechanism imposes a new requirement on the clients, which now need to keep sending keep-alive requests during periods of inactivity to make sure their sessions are not expired prematurely. The Raft cluster treats these requests as normal (empty) client commands; they get replicated and committed, but don't have any effect on the state machine.

Taking into account these changes to the algorithm, whenever a Leader receives a request from the client, it will append it to the log, replicate it and commit it. If the request is to register a new client, the Leader will assign a unique id and include it in the response to the client. If the request contains a command, the Leader will attempt to apply it to the state machine. If the state machine session contains no record of that client, or if the command's sequence number has already been discarded it is not safe to apply this command, in which case the Leader will send a sessionExpired error to the client. Otherwise, if the command has already been processed, the Leader can send the response stored in the session back to the client and if it hasn't, it can be safely applied to the state machine. Once the state machine successfully executes the command, the result can be stored in the session and sent back to the client.

3.4.5. CLIENT'S PERSPECTIVE

From a client's perspective, the first thing it needs to do in order to send commands for processing is to find the Raft cluster. To keep things simple, in my implementation clients are required to know the address of the servers in the Raft cluster beforehand, but this could easily be pushed out to an external service available at a well-known location, such as DNS.

Secondly, since client requests are processed only by the Raft Leader, there must be a way for a client to find the Leader. There are multiple ways to approach this, the simplest which is to have the client try to connect to a random server. If this server is not the Leader, then it will reject the request, and the client can try again with another server from the cluster.

This process can be sped up significantly using the fact that servers usually know the address of the current leader and thus they can include it in their response. Another, more sophisticated approach is to have the Follower proxy the client's request to the current leader. However, this requires additional mechanism to implement correctly and for simplicity, the first approach was chosen.

Additionally, even without request proxying, Raft must take additional measures to prevent stale leadership information from delaying client requests indefinitely. For example, a stale Leader might be partitioned from the rest of the cluster, while still being able to communicate with the client. In this case, the client's request could be delayed forever, since the Leader would never be able to commit the client's command. This is solved by having the Leader step down and revert to Follower if an election timeout elapses without it being able to reach a majority of the cluster. If a Leader steps down, it will also notify any pending clients which can then retry their requests with other servers.

Similarly, stale leadership information in Followers can unnecessarily delay client requests. Currently, Followers record the leader's identity so that they can redirect clients, but without additional mechanism it is possible for two Followers to redirect to each other. To prevent this from happening, it is necessary that Followers discard Leader information whenever a new election takes place or the term changes.

Once the client successfully located the Leader of the Raft cluster, it first needs to register before it can begin sending requests. During registration, the client gets assigned a unique identifier, which it will need to append to any command sent to the cluster. Once a client is successfully registered, it can begin sending requests/commands. The client is responsible for assigning a unique and monotonically increasing sequence number to each command that it sends. It also needs to keep track of which commands it received a response for and resend any commands (with their original sequence number) for which it did not receive a response for a certain period of time. Additionally, the client also keeps track and updates a response sequence number variable, which is appended to every command sent to the distributed state machine. This variable tracks which is the lowest command sequence number for which the client still misses a response. The session object then uses this value to decide which commands from the session can be safely discarded.

3.4.6. ARCHITECTURE

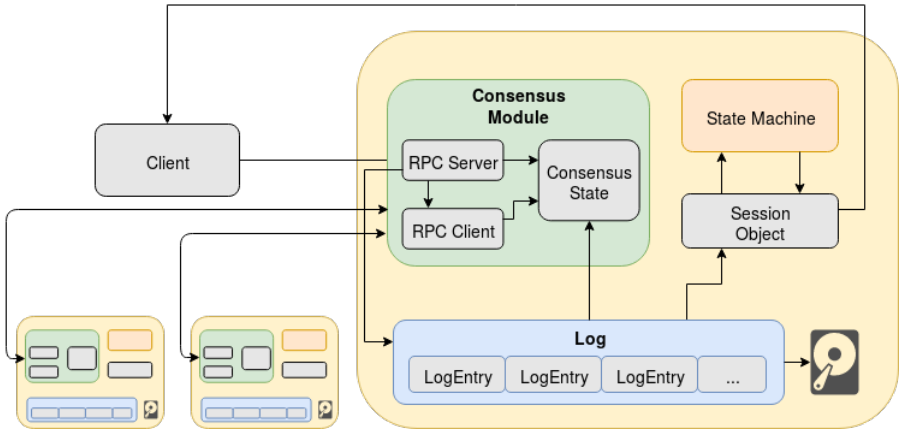

Figure 3.2: Raft architecture

Figure 3.2 shows the architecture of a Raft cluster with 3 members. RPC messages from the client arrive at the Leader where they get added to the local Log and sent to the rest of the cluster for replication.

The Consensus State module on the Leader keeps track of how many members of the cluster have successfully replicated the new entry. These values are updated after the Log has been successfully persisted to disk on the leader, or after RPC responses are received from the Followers.

Once a majority of the cluster has replicated the entry, it can be 'executed' on the state machine. This is done by first checking the session object to see if the command has already been executed. If it has, then the stored result is sent back to the client, otherwise the command gets applied to the state machine and the result is stored in the session before being sent to the client. If there's no entry of the client in the session object, or the session values for this command have been discarded, then the client will receive a sessionExpired error and it will have to decide how to recover from that.

3.4.7. EXTENSIONS

In addition to the core Raft algorithm, the paper[16] also describes two extensions to the basic consensus algorithm that add additional features. The first one addresses cluster membership changes and allows the cluster to change its configuration while running by adding or removing servers. To achieve this safely, changes are limited to one server (added/removed) at a time. Newly added servers won't have any log entries, and thus, in order to avoid availability issues, they will initially be in a new state: non-voting member. Servers in this state are not counted for voting or commitment majorities. However, they will receive log entries from the Leader and they will remain in this state until they catch up with the rest of the cluster, or the configuration change is aborted.

The second extension, log compaction, addresses the issue of the replicated log grow-

ing in size during normal processing of client requests. Since log entries are always appended to the log and never deleted, some form of log compaction becomes necessary in order to keep the system running. The general idea is based on the observation that much of the information in the log becomes obsolete over time and can be discarded.

The Raft paper discusses a number of different approaches for log compaction, depending on the needs of the system implementer, but they are all based on the fact that the internal state of the state machine at a moment T has been reached by applying a number nT of log entries. The idea is that if the state at moment T is snapshotted and written to disk, and the snapshot is loaded when the server restarts, it is then safe to discard the nT log entries without having a loss of information. One thing to note is that the session object also needs to be snapshotted together with the statemachine, otherwise it would be possible for the same command to be reapplied after a restart.

I won't be discussing these extensions is more detail since they are both orthogonal to the core Raft algorithm and are not essential to understanding the remainder of my work.

REFERENCES

[1] F. B. Schneider, *Implementing fault-tolerant services using the state machine approach: A tutorial,* ACM Computing Surveys (CSUR) **22**, 299 (1990).

[2] M. Burrows, *The chubby lock service for loosely-coupled distributed systems,* in *Proceedings of the 7th symposium on Operating systems design and implementation* (USENIX Association, 2006) pp. 335–350.

[3] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, *Zookeeper: Wait-free coordination for internet-scale systems.* in *USENIX annual technical conference*, Vol. 8 (2010) p. 9.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The google file system,* in *ACM SIGOPS operating systems review*, Vol. 37 (ACM, 2003) pp. 29–43.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, *The hadoop distributed file system,* in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on* (IEEE, 2010) pp. 1–10.

[6] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, *et al.*, *The case for ramclouds: scalable high-performance storage entirely in dram,* ACM SIGOPS Operating Systems Review **43**, 92 (2010).

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, *et al.*, *Spanner: Google's globally distributed database,* ACM Transactions on Computer Systems (TOCS) **31**, 8 (2013).

[8] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, *Megastore: Providing scalable, highly available storage for interactive services.* in *CIDR*, Vol. 11 (2011) pp. 223–234.

[9] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, *Scalable consistency in scatter,* in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (ACM, 2011) pp. 15–28.

[10] L. Lamport, *The part-time parliament,* ACM Transactions on Computer Systems (TOCS) **16**, 133 (1998).

[11] L. Lamport *et al.*, *Paxos made simple,* ACM Sigact News **32**, 18 (2001).

[12] B. M. Oki, *Viewstamped replication for highly available distributed systems*, Tech. Rep. (DTIC Document, 1988).

[13] B. M. Oki and B. H. Liskov, *Viewstamped replication: A new primary copy method to support highly-available distributed systems,* in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing* (ACM, 1988) pp. 8–17.

[14] B. Liskov and J. Cowling, *Viewstamped replication revisited,* (2012).

[15] F. P. Junqueira, B. C. Reed, and M. Serafini, *Zab: High-performance broadcast for primary-backup systems,* in *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on* (IEEE, 2011) pp. 245–256.

[16] D. Ongaro, *Consensus: Bridging theory and practice*, Ph.D. thesis, Stanford University (2014).

[17] D. Ongaro and J. K. Ousterhout, *In search of an understandable consensus algorithm.* in *USENIX Annual Technical Conference* (2014) pp. 305–319.

[18] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, *An overview of the Scala programming language*, Tech. Rep. (2004).

[19] *grpc - a high performance, open-source universal rpc framework,* http://www.grpc.io/.

[20] *Reactivex - an api for asynchronous programming with observable streams,* http://www.reactivex.io/.

[21] L. Lamport, *Time, clocks, and the ordering of events in a distributed system,* Communications of the ACM **21**, 558 (1978).

[22] M. P. Herlihy and J. M. Wing, *Linearizability: A correctness condition for concurrent objects,* ACM Transactions on Programming Languages and Systems (TOPLAS) **12**, 463 (1990).

4

ADDITIONAL INFRASTRUCTURE

4.1. REMOTE CODE EXECUTION ON THE STATE MACHINE

A number of commonly used Rx operators transform streams by applying user-defined functions to their elements. For example, the map operator takes a transformation function of the type T => R and uses it to transform each element of the stream. On the other hand, the filter operator takes a function of the type T => Boolean and only keeps stream elements that return true.

In order to implement a remote state machine that can execute such operators, we need to add support for arbitrary code execution on our Raft cluster. Scala has the ability to serialize/deserialize closures, which are implemented as anonymous classes, but that is not enough for our use case due to the fact of how serialization works in Java/Scala.

Both languages make a distinction between the concepts of class and data (an object). The class represents the code to be executed, while the object represents the state values associated with that code. Different instances of the same class can have different state, but they all refer to the same code. Classes in the JVM are identified by their fully qualified name, which consists of their class name, package name and the ClassLoader instance that loaded that class.

Once a class is loaded into a JVM successfully, it will be reused and not loaded again. The basic class loaders in a JVM application are the "bootstrap" class loader, responsible for loading key Java runtime classes, the extension class loader that loads classes in the java.ext.dir path and the app class loader, that loads classes in the java class path. All class loaders, except for the "bootstrap" class loader have a parent, and that parent is the class loader instance that loaded that class loader.

Whenever a class loader has to load a class, it will try to load it itself, and if it fails it will propagate the call to the parent class loaders, until it eventually reaches the bootstrap class loader. If a class definition still cannot be found, then the findClass() method is called, which by default throws a ClassNotFoundException and developers are expected to implement this method to define custom class loaders.

When an object is serialized, its data (internal state) is written to an OutputStream along with other metadata such as its class identifier. Whenever the object is deserialized, its class identifier is read and the current class loader will attempt to load the correct class. However, the class bytecode (the implementation of the class) is not included in the OutputStream at serialization, which means that when deserializing the object using the standard approach, the appropriate class must already be loaded. Since the goal of the library is to allow users to write arbitrary code inside Rx operators, it is impossible to have the class definitions already present on the Raft servers, which means they must be loaded dynamically [1].

4.1.1. SENDING INDIVIDUAL CLIENT CLASSES TO A RAFT CLUSTER

The first attempt to tackling this problem was to set up a class server on the client and have the raftCluster contact this server to request any missing classes during deserialization. On the server side, to avoid naming collisions, a different class loader is used per client. Thus, when the raft Leader would receive a command that it could not deserialize, it would contact the client and request the bytecode for the missing classes. Once all the missing class definitions were received, they would be attached to a special type of StatemachineCommand and appended to the raft log.

```
case class ClassDefCommand(command: StatemachineCommand, classDefs:
    Map[String, Array[Byte]]) extends StatemachineCommand
```

Attaching the class definitions to the command and appending them to the log makes sure that they are also sent and replicated to the rest of the cluster, and thus exempts other servers from having to request the class definitions from the client. It's also worth noting that only the first log entry that contains a certain command type has to contain the class bytecode. Once these definitions have been read and loaded into a class loader, they will be reused and not loaded again.

Since log entries are guaranteed to be applied in log order, it is safe to skip the inclusion of class definitions in later log entries, since they would always be already loaded. Of course, if an extension like log compaction is used, where entries from the log are discarded, it becomes necessary to save (and load, upon restart) the class definitions to stable storage. Otherwise, if the log entry that contains the definitions is discarded, and the server is restarted, it might become impossible for the server to deserialize and reapply some commands from its log, leading to a corrupt statemachine state. For example, this would happen if the client is no longer available during deserialization, and thus the server would have no way of finding the class definitions for the commands it needs to deserialize and apply.

While this approach works in practice, it also introduces some latency due to the communication needed to load all the classes necessary from the client. Whenever the server would attempt to deserialize a command that it had no class for, it would contact the class server on the client and request the bytecode for that class. Upon receiving the reply, the server would continue with the deserialization process until it finished or found another missing class, in which case it had to contact the client again and repeat the process. In a nutshell, for every missing class, the server would have to wait for a message round trip before it could continue with the deserialization. This delay might become significant as it increases linearly with the number of missing classes on the server.

4.1.2. SENDING THE CLIENT APPLICATION TO A RAFT CLUSTER

The second attempt to handle sending client class definitions to the server was to package the client application with its dependencies into a jar file, that would then be sent to the raft server on client registration.

On the server side, whenever a register client command is committed, a new jar class loader is created using the received jar file and associated with the client session. Any new commands received from this client will then be deserialized using this new class loader. The previous observation regarding log compaction still stands, if the register client log entry can be discarded, then the jar file must be saved to stable storage and kept until the client session expires.

However, this second approach does have the potential of creating a significantly large register client message, especially if the client-side code has a lot of large dependencies. This could be improved by analyzing the bytecode of sent commands with libraries like [2], which would make it possible to append only the classes that are actu-

ally used in the serialized command. However, due to the added complexity necessary to make this work, we chose to keep things simple and package and send the jar with dependencies, which is also what other systems like Spark[3] and Storm[4] also do to ensure that the client classes are present on their worker nodes.

4.2. FORWARDING MESSAGES TO OTHER CLUSTERS

While a system based only on the core raft algorithm would definitely work, it would run into an issue caused by the way Raft was designed: as a replicated server that processes commands from the clients and replies with responses. This means that in the case of queries that chain multiple ObserveOnRemote operators, the client will always have to act as an intermediary between each remote step since it will always be the one that receives the responses.

For example, let's consider a query like the one below:

```
Observable.of(words)
   .observeOnRemote(raftClusterA).filter(length > 5)
   .observeOnRemote(raftClusterB).map(count vowels)
   .observeOnRemote(...)...
```

In this case, the client will stream words to raftClusterA, which will perform the filtering on length and stream the results back.

The client will basically receive the responses only to forward them to raftClusterB for further processing. The same thing will happen for every observeOnRemote step in the query. Ideally, in such a case, we would like the client to be responsible only for producing the input events of the stream and not act as an intermediary between the different remote steps in the query.

To achieve this, each raftCluster needs to be able to propagate commands to the appropriate downstream clusters. As a side effect of this approach, the client would also have the possibility of exiting the system once all its events have been successfully replicated by the first cluster in the stream.

What this means in practice is that the upstream cluster needs to be able to also act as a client for the downstream. From the perspective of the client, raft servers will have the same behavior as before: they receive commands, replicate and process them and send back responses.

From the perspective of the downstream cluster though, a raft server upstream will behave the same as any other client: it needs to find the cluster Leader, register, send keep alive messages, keep track of commands sent and increment their sequence numbers accordingly, keep track of responses received and lowest sequence number for which a response is still missing and also resend commands for which there was no response for a certain period of time.

4.2.1. MESSAGE FLOW OVERVIEW

Considering these new requirements, the message flow in the Raft server is very similar with the default algorithm up until the message gets applied to the statemachine. A command is received from the client, is appended to the log and replicated and, once it

is successfully committed, applied to the statemachine.

The differences start here: once the command is applied, the statemachine needs to produce both a response that will be sent back to the client and a command that will be sent to the downstream servers. Sending a response back to the client is necessary, even if the response is empty, because it lets the client know that the server successfully received and replicated the command. Since we want to tolerate failures, the commands produced by the statemachine need to be saved and stored until they have been successfully sent and replicated to downstream.

The forwarding module is responsible for registering the client on the downstream, sending the commands produced by the statemachine to the downstream clusters, sending keepAlive messages and keeping track of received responses and sequence numbers.

As soon as the statemachine on raftClusterA produces an output that contains a command for raftClusterB, the forwarding module is notified that it needs to register a client on raftClusterB, and the command is stored in the session object. The forwarding module cannot start sending commands to the downstream cluster until it has received a valid clientId, and this clientId has been successfully replicated. Without this precaution, it would become possible for commands to be duplicated on the downstream statemachine which would result in a corrupt state. Additionally, any commands received while the clientId is missing, will be stored in the statemachine sessions.

Once the clientId received from the downstream cluster has been successfully replicated, it is safe to start sending commands to the downstream, in the order they were produced. There is one additional precaution we must take in order to make sure that we comply with raft's rules: the forwarding module cannot update it's response sequence number until the majority of the upstream cluster is aware that a valid response has been received.

The problem is similar to registering a new client, but in this case would result with a sessionExpired exception instead of a corrupted downstream statemachine. For example, let's consider a leader that receives a response for command 0 and immediately updates the response sequence number and sends the update with command 1. As soon as the downstream cluster applies command 1, it will discard command 0 from the session. Now, if the upstream leader crashes before the majority of its cluster finds out that command 0 has received a response, the new leader will resend command 0. On the downstream, this would result in a sessionExpired exception, since command 0 has just been discarded.

Thus, in order to avoid this possible issue, the upstream Leader needs to deterministically remove commands from its session by appending RemoveCommand entries (with appropriate sequence numbers) to its log. Once these log entries are successfully replicated and committed, the downstream commands can be discarded and response sequence numbers updated.

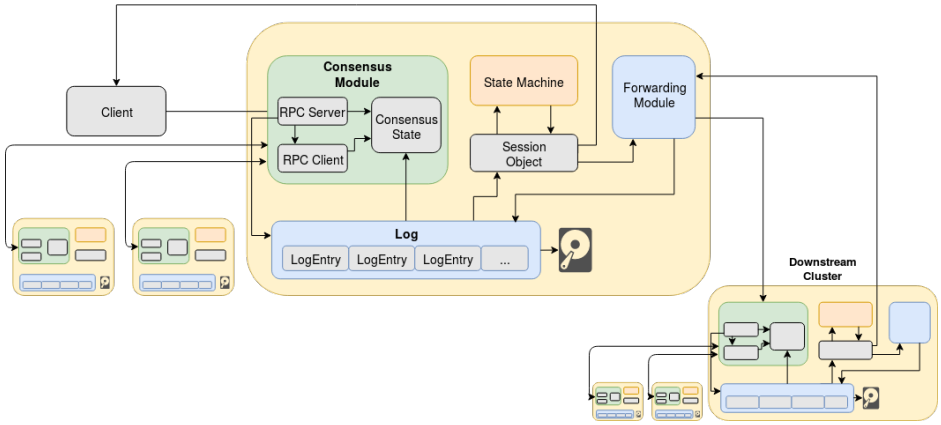

Figure 4.1: Raft forwarding architecture

4.2.2. STATEMACHINE CHANGES

From an interface point of view, the statemachine no longer returns only a response, but an output that can be either a simple response or a response and sequence of downstream commands.

```
trait Statemachine {
   def apply(command: StatemachineCommand): StatemachineOutput
}

trait StatemachineResponse extends StatemachineOutput

trait ForwardingResponse extends StatemachineOutput {
   def response: StatemachineResponse
   def commands: Seq[DownstreamCommand]
}
```

Since the commands will be sent to another raft cluster for processing, they need to not only be valid commands for that cluster's statemachine, but also contain identifying information about the downstream cluster (i.e the network addresses of its members).

Also, to keep things simple, the forwarding module will only used received responses to discard commands from the command sessions. If any responses returned are also valid statemachine commands, they will not be re-applied to the local statemachine. This means that whenever the statemachine produces a ForwardingResponse, the response should effectively be Empty/No-Op.

```
case class DownstreamCommand(downstreamClusterClient: RaftClusterClient,
    command: StatemachineCommand)
```

The downstreamClusterClient parameter of a downstream command identifies a 'virtual' client on the downstream. The id part of a RaftClusterClient object is completely

determined by the implementer of the state machine and does not refer to a registered clientId on the downstream.

```
case class RaftClusterClient(raftCluster: Set[ServerAddress], id: String)
```

4.2.3. SESSION CHANGES

In the core Raft algorithm, the session object keeps track of (command, response) pairs for each client. Any command received that was already present in the session object would not be applied to the statemachine and the client would receive the stored response. Values from the session object would be deterministically discarded based on the response sequence number received from the client.

This approach still works perfectly well with the changes to the statemachine interface. The only change is that now it is possible for the statemachine to also produce downstream commands in addition to responses. This means that we must also keep track of which downstream clients have been registered, which commands we still need to deliver and deterministically remove them once they have been successfully sent and replicated to downstream.

Since the session object already needs to be as persistent as the statemachine, it is a natural choice to use it for keeping track of downstream commands.

```
val pendingCommands: Map[RaftClusterClient, CommandSession]
val registeredClients: Map[RaftClusterClient, String]
```

Thus, in addition to tracking commands and responses for clients, the session object will now also track which commands still need to be sent to which RaftClusterClient and which clientIds are associated with which RaftClusterClients.

```
case class CommandSession(startingSeqNum: Int, producers: Set[String],
    commands: Seq[StatemachineCommand])
```

As mentioned before, it is important to keep track of which clientId is used to push commands to downstream for a certain RaftClusterClient. If the clientId is not saved, then whenever a leader changes, a new clientId will be generated, which can allow for duplicated commands to be applied on the downstream cluster.

For example, let's take the case of a cluster with 3 servers. The Leader receives a command, replicates and applies it to the statemachine, sends the response back to the client and the command to the forwardingModule. The forwardingModule contacts the downstream server, creates a clientId and sends the command, which is replicated and applied on the downstream cluster.

If the Leader now changes on the upstream, the newly elected Leader will not know what clientId the previous Leader used and thus create a new one and resend the command. Even though the command has the same sequence number as before, it will be reapplied since the clientId is different. This will corrupt the state of the statemachine, since the upstream cluster intended for the command to be applied once, but it was actually applied twice.

Another thing to note is that we also need to keep track of which sequence numbers to use when sending commands. Without keeping track of sequence numbers, any leader change will cause the sequence number to reset, which might result in a sessionExpired error and cause the downstream to refuse new commands.

My approach was to keep track of a 'seed' sequence number (startingSeqNum) in the session object and use this value in the forwardingModule, incrementing it with every new command sent. Once responses for commands would be received, the forwardingModule would append the confirmed sequence number to the log via a RemoveCommands entry. When this message gets committed, the value for startingSeqNum is updated in the commandSession and the respective commands are dropped.

For example, let's consider a case when the Leader receives a command from a new client, the pendingCommands map is empty and the clientId has just been successfully registered. Let's say that applying the command to the statemachine produces 2 downstream commands to raftClusterB. Once the command is applied and committed, since there is no entry for raftClusterB, a new commandSession object is created with a starting value of 0 for startingSeqNum, and the 2 downstream commands.

The forwardingModule will use this value of 0 as the sequence number of the first command, and increment it for any new command sent. Thus, the second command will get a sequence number of 1. Once the forwardingModule receives the response for the first command (which, as noted before, it will be NoOp/Empty), it will append the message RemoveCommands(raftClusterB, 1) to the log.

When this entry is committed, the commandSession for raftClusterB will update its startingSeqNum to 1 and delete the first command from the commands sequence. Also, the forwardingModule can now safely increase its response sequence number and attach the new value(1) to any keepAlive messages that it sends. This lets the downstream cluster know that it is safe to discard the session value for the command with sequence number 0. Similarly, after a response is received for the second command, the forwardingModule will append RemoveCommands(raftClusterB, 2) to the log, which will cause the commandSession to remove the second command and update its starting sequence number variable to 2. Thus, any new commands that will be sent to raftClusterB will start from a sequence number of 2.

4.2.4. PREVENTING INFINITE KEEPALIVE MESSAGES

Since we want the forwardingModule to send commands to downstream servers, this also means that the forwardingModule has the responsibility of sending keepAlive messages to the downstream, while commands arrive from clients and are committed/applied to the statemachine. This raises a new problem, where the forwardingModule needs a way of finding out when it can stop sending keepAlive messages to a certain cluster. Without any additional mechanism, the forwarding module would keep sending keepAlive messages infinitely since it has no way of knowing if it has sent all the commands to downstream.

This is where the producers set in the CommandSession comes in. This variable is used to keep track of which upstream clientIds pushed values to the downstream RaftClusterClient. Whenever a command from a clientId produces a downstream command that needs to be sent to a raft cluster, that clientId is added to the producers set for the

RaftClusterClient. Similarly, whenever the session for a clientId expires, that clientId is removed from the sets of producers for the appropriate RaftClusterClient.

Thus, whenever a RaftClusterClient ends up with an empty producers set, we know that there won't be any new commands coming in and thus we can notify the forwardingModule that it is safe to stop sending keepAlive messages as soon as it has finished delivering the current commands.

Also worth noting is the fact that if the downstream statemachine needs to be able to load/execute arbitrary code from clients, it will need to receive the appropriate client jar files in order to be able to deserialize commands. This is also where the producers set proves useful: when the forwarding module needs to register a new client, it will read the producers set and attach all client jars to the register client message.

There's also one additional change needed to cover an edge case in the standard Raft algorithm. The standard algorithm attaches a timestamp to any command that gets appended to the log and applied to the statemachine. This entry will get replicated to all servers, and when it is applied to the statemachine, this timestamp value is used to decide which sessions expired and which not. However, since heartbeat entries do not get appended to the log, it means that at any moment, the session object could potentially have undetected expired sessions. For example, if one client sends just one command and stops sending keepAlives after receiving a response, its session will remain in the session object until another command gets applied.

This is not a big issues for the standard algorithm, since in the worst case, a few objects will be kept in memory longer than needed. However, this is more problematic when the forwardingModule comes into play, since it relies on session expiration to decide when to stop sending keepAlives to downstream clusters. To solve this issue, the RPC server on the Leader keeps track of how much time has passed since it has received a message from a client. If no message has been received for a period of time greater than or equal to a sessionTimeout, it will append a NoOp message to the log. This event will get assigned a timestamp and it will be replicated the rest of the cluster and applied to the statemachine. This basically forces the server to detect any expired sessions that are still in the session object, and will cause the forwardingModule to stop sending keepAlives.

4.2.5. SENDING MESSAGES FROM THE LEADER

The initial implementation of message forwarding used the statemachine to synchronously send commands to the downstream servers. This however raised a number of issues. First of all, since the statemachine is replicated, sending commands from the statemachine means that we will send the same command at least once from each replica. This, coupled with the fact that commands are applied to the statemachine in parallel with log entries being appended and replicated, meant that the statemachine state could have become inconsistent.

For example, considering a cluster with 3 servers, once a command gets committed, it gets queued to be applied to the statemachines. If one of the servers is slow in applying the commands, it becomes possible that the Leader sends the command downstream, receives a response, replicates a new command and sends it to the downstream cluster with an updated response sequence number. In this case, once the downstream receives

the command from the Leader, it will discard the session entry for the previous command. If the slow server didn't send the previous command up until this point, it will now receive a sessionExpired error because the session entry has been discarded, instead of receiving the result. This means that it is possible for the statemachine state on the slow server to differ from the rest of the cluster, which corrupts the replicated state machine.

Another issue with this approach is that, depending on cluster configuration, it might have a significant impact on the downstream cluster's ability to process client commands. For example, if a cluster with 5 replicas wants to send commands to another cluster, and those commands are sent from the statemachine, that downstream cluster will eventually receive the same command at least 3 times. Assuming there is no delayed/slow statemachine in the initial cluster, all these duplicated commands will get appended to the log, replicated and applied sequentially to the statemachine (first will actually be processed, the rest will use the result stored in the session). Since commands are appended to the log and processed by the statemachine sequentially, this will delay the processing of commands from other clients, compared to a scenario where there are no duplicated commands.

Thus, a more viable approach is to use a single server to send commands to the downstream. The Leader makes a natural choice for this, since Raft guarantees that there will only ever be a maximum of one Leader in the system at a certain time. Moreover, registering a new client and removing downstream commands from the pending commands queue needs to happen deterministically, on all replicas at the same (logical) time, which means that this needs to be done through a command that gets appended to the log and replicated.

Having this functionality running on the Leader makes things easier, since the Leader is the only server that is allowed to append entries in the log. In theory, it is possible to have forwarding modules on other servers handle the sending of commands, but they would need to contact the Leader whenever a new client has been registered, or a command has been successfully sent and needs to be removed from the queue.

To enforce this, whenever a new Leader is elected, the forwardingModule on that server receives a snapshot of all the unregistered RaftClusterClients from the session, together with the downstream commands and their starting sequence numbers. The forwarding module will then start registering the missing clients and start sending pending commands and keepAlives for clients that have been already registered.

Whenever a new command gets applied to the statemachine and produces a downstreamCommand, this also gets sent to the forwarding module, if the client is already registered. If the client is not registered, the command is saved in the session and a RegisterDownstreamCommand is sent to the forwarding module instead. Any new incoming commands for this client will also be saved until the clientId is successfully registered on the downstream. As soon as that happens, any saved commands are passed to the forwardingModule for delivery.

If and when the Leader steps down, it will stop the forwardingModule from sending any commands and keepAlives. Any responses that are currently in flight towards the old Leader will also just be ignored, since the Leader won't be able to append any entries to the log anyway.

This is safe to do, since the newly elected Leader will pick up where the old Leader left off and re-send any commands that have not yet been removed. Even if the downstream cluster already processed and sent a response for these commands, they will still be present in the session and thus will not corrupt the statemachine.

REFERENCES

[1] S. Liang and G. Bracha, *Dynamic class loading in the java virtual machine,* Acm sigplan notices **33**, 36 (1998).

[2] *Asm - an all purpose java bytecode manipulation and analysis framework,* http://asm.ow2.org/.

[3] *Apache spark - lightning-fast cluster computing,* https://spark.apache.org/.

[4] *Apache storm,* http://storm.apache.org/.

5

THE RX STATE MACHINE

5.1. INTRODUCTION

Going back to the start of this paper, one of our goals is to implement a system that allows part of an Rx query to execute on a remote machine. The query below shows an example use-case of what we would like to achieve: the cluster raftClusterA will receive a stream of words and execute the filter and map parts of the query, after which it will send the results to raftClusterB.

```
Observable.of(words)
   .observeOnRemote(raftClusterA).filter(length > 5).map(toLowercase)
   .observeOnRemote(raftClusterB).map(countVowels)
   .observeOnRemote(...)...
```

Considering the code snippet below as an example of a typical Rx query, if we look at the map operator, the possible events that it can receive are Subscribe, onNext, onError and onComplete. Subscribe has the role of providing an observer to which the map operator should push results, while onNext, onError and onComplete signal potential events in the stream. OnNext provides the next element to be processed, onError signals that an error was produced while onComplete marks the end of the stream.


```
Observable.of(words).filter(length > 5).map(toLowercase)
   .subscribe(printObserver)
```

In addition to these events, there is an additional way in which the stream can be cancelled, and that it by unsubscribing the consumer at the end, the printObserver. In the Rx library this is done via a Subscription object that is wired throughout the stream, which means that unsubscribing the printObserver will also be visible in the map and filter operators.

Naturally, our Rx statemachine running on raftClusterA will also need to be able to handle the same set of events. However, since part of the query will now run on a different machine, or in a different context(JVM), it would be impossible to detect when the observer unsubscribes without making it an explicit event. This makes our current set of possible statemachine events: Subscribe, Unsubscribe, onNext, onError and onComplete.

From the perspective of a statemachine running on raftClusterA, there are 2 major possible use cases, illustrated below:

```
Observable.of(words)
   .observeOnRemote(raftClusterA).filter(length > 5).map(toLowercase)
   .subscribe(printObserver)
```

```
Observable.of(words)
   .observeOnRemote(raftClusterA).filter(length > 5).map(toLowercase)
   .observeOnRemote(raftClusterB).map(count vowels)
   .subscribe(printObserver)
```

The differences are fairly straight forward: in the first example, the stream ends on the remote machine, while in the second example the statemachine will produce events that need to be pushed to raftClusterB.

This will have an influence on the output of the statemachine when processing client commands. In the case of streams that end locally, the statemachine will only produce responses for the client, in order to notify it that its commands have been received and applied. In the second case, the statemachine will produce both responses for the client and new events (commands) for the downstream cluster.

5.2. INTERNAL STATE

In order to set up an Rx stream that runs operators sent by the client and processes onNext, onError and onComplete events, the statemachine needs to create an object that acts both as an observer (and thus can accept Rx events) and an observable, such that it can be transformed using the Rx operators sent by the client.

In the Rx world, this role is played by a Subject, and it represents the input side of the stream running on the statemachine. After transforming the events pushed to this subject using the client's operators, we need to observe the produced events by subscribing to the transformed stream. The functionality of the observer that we subscribe will differ based on whether the stream will end locally or its results need to be pushed to another machine.

This pairing of a subject on the input side and an observer on the output side is all we need to set up a stream that processes events from the client. Whenever a new Rx event gets applied to the statemachine, it will be pushed to the input subject, get transformed by the operators and the result pushed to the observer.

If the results need to be sent to another machine for further processing then they will need to be collected from the observer and the statemachine needs to return them as downstream commands. Otherwise, if the stream ends on the cluster, the statemachine can just return an Empty/No-Op message as the result.

This approach can also be generalized to support the processing of multiple streams on one statemachine. In this case, instead of keeping track of just one pair of subject and observer, the statemachine needs to keep track of multiple such pairs and discriminate them based on a streamId.

Thus, the internal state of the Rx statemachine is map from streamIds to RxStreams, where RxStreams represent a pairing of input subject and ResultObservers.

```
val streams: Map[String, RxStream]
case class RxStream(subject: Subject[Any], observer: ResultObserver[Any])
```

For streams that end locally, the ResultObserver behaves just like a regular observer, but for streams that need to send commands to other clusters, it will save any events produced by an input event. After an input event has finished processing in the stream, any saved events produced are wrapped into downstream commands and returned as part of a ForwardingResponse.

5.3. COMMANDS

As mentioned before, the 5 basic events that the statemachine needs to support in order to process Rx streams are: Subscribe, onNext, onError, onComplete and Unsubscribe. Since streams running on the cluster are identified by a streamId, any event that needs to be applied to the statemachine needs to specify the target stream by including the appropriate streamId.

5.3.1. SUBSCRIBE

Whenever the statemachine receives a Subscribe message it means that a client wants to set up a new stream for execution on this cluster. Thus, the subscribe message should contain the necessary information about the new stream: streamId, the operators that need to be executed and whether the stream ends locally or should produce downstream commands.

```
trait StatemachineObserver

case class Subscribe(streamId: String, operators: Seq[Operator],
    observer: StatemachineObserver)
```


Upon receiving a Subscribe message, the statemachine will first create a new result observer using the information present in the statemachineObserver parameter.

```
case class LocalStatemachineObserver(onNext: Any => Unit, onError:
    Throwable => Unit, onCompleted: () => Unit) extends
    StatemachineObserver
```

In the case of a local stream, we will create a regular observer using the onNext, onError and onComplete lambdas.

On the other hand, if the stream does not end locally, we will create an observer that caches any events produced into a list and transforms them to downstream commands. The target cluster of the downstream commands is determined using the raftClusterClient parameter passed to DownstreamStatemachineObserver.

```
case class DownstreamStatemachineObserver(streamId: String,
    raftClusterClient: RaftClusterClient) extends StatemachineObserver
```

Finally, the streamId parameter on the statemachine observer is used to identify which stream on the downstream statemachine needs to receive the produced events (either onNext, onError, onComplete or Unsubscribe).

Once an observer is created, the statemachine will then check if a stream with the supplied streamId already exists and if that is the case, it will push an onError event to the newly created observer. If no stream exists with the supplied streamId, then a new subject will be created and the operators will be sequentially applied to construct the stream.

Any errors that might occur during this process will be pushed to the created observer. Lastly, once all the operators are successfully applied, the observer will be sub-

scribed to the newly created stream and a new entry containing the streamId, and RxStream will be added to the streams map.

As a return value, the statemachine will return an Empty message in the case where the stream ends locally or an output containing an Empty response and downstream commands for the remote case.

5.3.2. RXEVENT (ONNEXT/ONERROR/ONCOMPLETE)

Handling Rx events is fairly straight forward. An RxEvent message will contain the streamId, together with the actual event that needs to be pushed to the stream. In the example below, Notification is an abstract class defined by the Rx library which has 3 children: OnNext, OnError and OnComplete.

```
case class RxEvent[T](streamId: String, notification: Notification[T])
    extends StatemachineCommand
```

Whenever the statemachine receives an RxEvent, it will first check if the streamId exists in the map. If it does not, or if it is unsubscribed, it will return an Unsubscribed response. This message will be ignored by any forwarding module upstream, but it is used in notifying the initial client know that the stream has unsubscribed and it can stop producing events.

However, if the streamId exists and the stream is not unsubscribed, the statemachine will push the event to the input subject and return the appropriate output (Empty response for a local stream and an Empty response together with possible downstream commands for remote streams).

5.3.3. UNSUBSCRIBE

Unsubscribe messages contain a single parameter, the streamId that has been unsubscribed on the downstream.

```
case class Unsubscribe(streamId: String) extends StatemachineCommand
```

Whenever the statemachine receives this message, it needs to iterate through its stream map and unsubscribe any streams that have observers with that streamId. In order to free up resources, any streams that are unsubscribed can also be removed from the map. The statemachine will always return an Empty response as an output for this command.

5.3.4. SETUPSTREAM

The current set of statemachine events do a good job of handling streams that have only one ObserveOnRemote operator. However, if multiple observeOnRemote operators are chained, we need a way to notify each remote stream of it's source stream location in order to generate correct unsubscribe messages.

Consider the example of an infinite stream on the client, that gets transformed remotely and then observed on another cluster.

```
Observable.of(infinteWords)
   .observeOnRemote(raftClusterA).map(toLowercase)
   .observeOnRemote(raftClusterB).filter(length < 10).take(10)
   .subscribe(save)
```

In this example, without additional mechanism, the observable on raftClusterB would terminate, but raftClusterA has no way of knowing about it, since responses are not re-applied to its local statemachine. In order to get around this problem, we introduce an additional statemachine command, SetUpstream, that is used to notify the stream on raftClusterB which stream on raftClusterA it should unsubscribe when it unsubscribes.

```
case class SetUpstream(streamId: String, upstreamClusterClient:
    ClusterClient, upstreamStreamId: String) extends StatemachineCommand
```

Whenever the statemachine receives this message, it stores it to be used when streamId is unsubscribe. When this happens, the upstreamClusterClient and upstreamStreamId fields are used to generate an Unsubscribe(upstreamStreamId) command that is going to be sent to the upstream cluster.

6

RX LIBRARY EXTENSIONS

6.1. OBSERVABLES AND REMOTEOBSERVABLES

In the Rx world, an Observable is an abstraction for a stream of zero or more events and represents a source of data. Events produced by an Observable can be transformed using various operators, but the defining operation for an Observable is the subscribe function, which is used to register an observer (a consumer) to the stream.

This function defines what happens when an observer subscribes to the observable and can be used to implement observables with custom functionality. After an observer subscribes, the Observable will start emitting items by calling the observer's methods (onNext, onError, onComplete).

For example, an observable that opens and reads a file line by line could be implemented by overriding the subscribe function to open and read the file and emit lines by calling the onNext method on the newly subscribed observer. Any errors that occur should be pushed to the observer using the onError method and, once all the lines are read, the observer's onComplete method should be called.

In an analogous fashion, the RemoteObservable represents an observable stream executing on a remote raft cluster.

```
trait RemoteObservable[T]{
   def cluster: Set[ServerAddress]
   def subscribe(observer: StreamSetupObserver[T]): Unit
}
```


Similarly, the main operation to be performed on a RemoteObservable is also a subscribe, but this time we don't subscribe a regular observer, but a StreamSetupObserver. Additionally, the RemoteObservable also keeps track on which cluster it is executing on. This information is used by some operators to decide whether events produced by a remoteObservable need to be sent to another cluster or not, depending on where they need to be observed.

One thing to note here is that the information used to identify a cluster are the addresses of the servers that form the cluster. This approach was used for simplicity and it can break when used in conjunction with raft extensions such as cluster membership changes. A more robust approach would be to use an identifier for each cluster and use a third party service to find the addresses of servers belonging to a certain cluster, but we will leave that for future work.

6.1.1. SUBSCRIBING TO A REMOTEOBSERVABLE

As mentioned above, subscribing is the defining operation of an Observable and can be used to implement custom observables or operators. In the Rx world, the subscribe function takes a single parameter: the observer that is interested in receiving the events of the stream.

Operators can be implemented by overriding the subscribe function and creating new observers that subscribe to the upstream source(s), apply some transformation on the events and push the results to the downstream observer that was provided to subscribe. This means that the observer received as a parameter by subscribe encodes the whole series of transformations that will happen downstream.

On the RemoteObservable side, the observer we subscribe needs to also be able to encode the series of transformations that happen downstream, but the big difference in this case is that different parts of the stream might be executing on different machines. It is thus unnecessary for an observer to keep track of the whole chain of transformation. Every statemachine that executes part of the chain needs to only know which operators need to be applied locally and where the resulting events should be sent.

```
case class StreamSetupObserver(operators: Seq[Operator],
    statemachineObserver: StatemachineObserver)
```

Taking this into consideration, a StreamSetupObserver contains the sequence of operators that will be executed on a certain remote statemachine, together with the statemachineObserver, which defines whether the stream ends on the statemachine or produces events elsewhere.

In addition to subscribing stream setup observers, ideally we would also be able to subscribe regular observers to a RemoteObservable source. This would mean that the remote stream does not produce any events downstream and 'ends' on the statemachine, but this could be useful for a variety of applications.

For this use case, it might be worth looking into making the RemoteObservable a subclass of Observable, but this does not work at the moment due to the fact that an observer is not serializable. As a workaround, the library provides subscribe functions that wrap the onNext, onError, onComplete lambdas into a local statemachine observer.

These subscribe functions are implemented as extension methods on the RemoteObservable type and whenever called, will create new StreamSetupObservers with an empty sequence of operators and a local statemachine observer created using the functions passed as parameters.

6.2. OBSERVEONREMOTE OPERATOR

The ObserveOnRemote operator takes a cluster parameter and is applied to an Rx stream to send the events produced by the stream to the statemachine executing on the cluster.

```
Observable.of(words).observeOnRemote(raftCluster)
```

This operator returns a RemoteObservable which represents the remote stream. Since the goal is to send any incoming events to the remote cluster for processing, this operator needs to take care of all the steps necessary to make that happen.

First of all, it needs to create a clientId on the raftCluster, after which it needs to send a Subscribe command containing the operators and the statemachineObserver.

Secondly, if the upstream is also a remote observable, and the downstream is not a local statemachine observer, it needs to send an appropriate SetUpstream command.

Lastly, the operator also needs to subscribe to the source observable and send any onNext, onError, onCompleted events produced to the raftCluster. Any commands/events sent to the cluster need to be transformed into valid raft messages, which means they need to have a correct sequence number attached and include any class definitions that

are missing on the cluster. Additionally, this operator also needs to make sure that the session is kept alive while messages can still be sent and any commands without responses are resent to the cluster.

Let's consider an example where we subscribe to a remoteObservable and print the produced events:

```
Observable.of(1).observeOnRemote(raftCluster)
   .subscribe(v => println(v))
```

Since we subscribed using an onNext lambda, the subscribe extension method on the remote observable will create a StreamSetupObserver initialized with an empty sequence of operators and a local statemachine observer.

As a last step, this StreamSetupObserver will be subscribed to the upstream remote observable. Upon subscribing on the upstream, a new clientId will be created on the raftCluster and for the sake of this example, let's assume the value returned for clientId is 5.

Once the client is registered, the subscribe function will send a Subscribe command using 5 as clientId, a randomly generated UUID as streamId, a sequence number of 0 and the operators and statemachineObserver contained by the StreamSetupObserver. In this case, the operators sequence will be empty, and the statemachineObserver will be the LocalStatemachineObserver containing the lambda that creates the printObserver.

After sending the Subscribe message, the next step is subscribing to the upstream observable which in this case produces the value 1 and completes. These events will be wrapped in statemachine commands (RxEvent) and sent using the same streamId as before.

On the statemachine side, when the Subscribe message is received, it will create a new subject and attempt to apply the operators passed in by the client. Since in this example the sequence of operators is empty, no transformation will be applied. After this, an observer is created using the lambda passed to the LocalStatemachineObserver and subscribed to the stream (which, in this example, consists of just the subject).

The resulting pairing of subject and observer is then wrapped into an RxStream object and added to the streams map and finally, an empty response is sent back to the client.

When the two events produced by the observable (onNext(1) and onCompleted) arrive at the statemachine, the streamId value is used to push these events to the appropriate subject. Since in our example we subscribed a local observer, no downstreamCommands will be created and the client will receive empty responses.

Back on the client side, when responses are received for all 3 messages sent (Subscribe, onNext(1) and onComplete) the sending of keepAlive messages will stop, which will eventually clear out the client session on the cluster.

6.3. OBSERVEONLOCAL OPERATOR

The ObserveOnLocal operator performs the reverse step to observerOnRemote: it is applied to a remoteObservable and transforms it into a regular observable.

```
Observable.of(words)
.observeOnRemote(raftCluster)
.observeOnLocal(localAddress)
```

Since events produced by the remoteObservable are going to be transformed to downstreamCommands and sent by the forwardingModule running on the raftCluster, this operator needs to make sure that an RPC server is running on the client.

This RPC server needs to respond to the commands that can be sent by the forwardingModule, namely register client, client requests and keepAlives. Since multiple remoteObservables might be sending events to this machine, the RPC server needs make sure it pushes the events received to the appropriate stream. Thus, similarly to the statemachine, the RPC server keeps track of a mapping from id to observer. This means that a new id needs to be created whenever an observer subscribes to observeOnLocal observable.

Once an RPC server is running on the local machine and an id has been created, observeOnLocal can use this information to create the DownstreamStatemachineObserver, which lets the statemachine running on raftCluster know where to send produced events. This observer is then used to create the StreamSetupObserver that will be subscribed to the upstream remoteObservable.

Let's adapt the previous example and have the statemachine send events back to the client where they will be printed.

```
Observable.of(1)
   .observeOnRemote(raftCluster)
   .observeOnLocal
   .subscribe(v => println(v))
```

When we subscribe the observer to the observeOnLocal operator, it will first check if an RPC server is running and start one if that is not the case.

Once the RPC server is up and running the observer will be passed in to the RPC server which will add it to its streams map and return an id (let's say with a value of 1). Next, the observeOnLocal operator uses the server address and the id value to create the DownstreamStatemachineObserver and StreamSetupObserver (initialized with an empty sequence of operators). Finally, this StreamSetupObserver is subscribed to the upstream remote observable.

Next, the behavior of the statemachine and subscribe function on the remoteObservable are similar to the previous example, the only difference being that the statemachineObserver will be a DownstreamStatemachineObserver instead of a LocalStatemachineObserver. This only has an impact on the output of the statemachine, which will use the information from the DownstreamObserver whenever it needs to push results.

In the case of the Subscribe command, the stream does not produce any events, and thus the statemachine will just return an empty response back to the client. However,

in the case of onNext(1) and onCompleted, the statemachine will send empty responses back to the client and propagate these events (via the forwardingModule) to the address and stream contained in the DownstreamObserver.

Back on the client side, when the RPC server receives events, it will use the included streamId to find the appropriate observer to push events to. In this case, for the onNext(1) and onCompleted events the included streamId will be 1, which means these events will be pushed to the PrintObserver.

6.4. OPERATORS

Operators defined on a remoteObservable stream have the same semantics as their counterparts in the default library. As a matter of fact, the purpose of the Operator trait and its implementations is to just record the transformations that need to be applied to the remote stream. These transformations will then be sent to the remote statemachine as part of the Subscribe message that creates a remote stream.

In the Rx world there are two main types of operators when taking into account the number of streams they subscribe to.

6.4.1. OPERATORS THAT TRANSFORM A SINGLE STREAM

The first type of operator is the one that subscribes to a single source and applies some kind of transformation or filtering on the items emitted.

```
trait Operator

trait SingleObservableOperator[T, S] extends Operator{
  def toObservable(left: Observable[Any]): Observable[Any]
}
```

For example, the map operator takes a function as parameter and applies it to every item emitted by the source. The result will be a new observable that emits the results of these function applications.

```
source.map(i => i + 1).subscribe(printObserver)
```

To achieve this, the map operator creates a new observable by overriding the subscribe function. In the example above, when the printObserver subscribes to this map observable, a new internal observer will be created and subscribed to the source. This internal observer has the role of receiving items from the source, applying the function on every element and pushing the result to the printObserver.

Similarly, the map operator on a RemoteObservable also takes a function as a parameter, but returns a new RemoteObservable as a result. Another difference in this case is that the observer will be a StreamSetupObserver that has as parameters a sequence of Operators and a statemachine observer.

```
def map[S](fun: (T) => S): RemoteObservable[S] = {
   new RemoteObservable[S] {
      override def cluster = src.cluster
```

```
        override def subscribe(downstream: StreamSetupObserver[S]): Unit = {
          src.subscribe(StreamSetupObserver[T](observer.operators:+
              Map(fun), observer.statemachineObserver) )
        }
      }
    }
```

The behavior of map in the remote case is also very similar: it will add the Map operator to the list of operators passed from the downstream observer, create a new StreamSetupObserver with this new list of operators and subscribe it to the source. Additionally, any operator that does not push events on a new cluster will use the upstream cluster value as it's own.

The implementation of the operator classes is very straightforward, their only role being to record the sequence of transformations defined by the client in a serializable format that can be then sent to a raft cluster for execution. Thus, the only operation defined on this class defines how it should transform a source stream. It takes an observable as a parameter and returns a new observable created by applying the appropriate transformation.

```
case class Map[T, S](fun: (T) => S) extends SingleOperator[T, S] {
  override def toObservable(left: Observable[Any]): Observable[Any] = {
    left.map(fun)
  }
}
```

For example, in the case of the Map operator, the source observable should be mapped with the function passed in by the client. Implementing other operators that transform a single stream like Filter and Take is also very similar.

6.4.2. COMBINING OPERATORS

In addition to operators that transform a single stream, the Rx library also defines operators that combine two source observables into a single one.

```
  trait CombiningOperator[T, S, R] extends Operator{
    def operatorId: String
    def streamPosition: StreamPosition
    def toObservable(left: Observable[Any], right: Observable[Any]):
        Observable[Any]
  }

  sealed trait StreamPosition
  case class LeftStream() extends StreamPosition
  case class RightStream() extends StreamPosition
```

A good example for this case is the withLatestFrom operator. This operator subscribes to two observables (in this example: letters and numbers) and whenever letters emits an item it will be grouped with the last item emitted by the numbers observable.

This grouping of a letter and number will then be transformed using the function passed as a second parameter and the result sent to the printObserver. Similarly to the map operator, withLatestFrom will create two internal observers that it will subscribe to the two observable streams (letter and numbers) and will push results to the print observer.

```
letters.withLatestFrom(numbers).subscribe(printObserver)
```

In the remoteObservable case, withLatestFrom will behave in an analogous fashion, creating two StreamSetupObservers and subscribing them to the source remote observables.

```
val letters = lettersObservable.observeOnRemote(clusterA)
val numbers = numbersObservable.observeOnRemote(clusterB)
letters.withLatestFrom(numbers).subscribe(v => println(v))
```

However, in this case the numbers observable is executing on clusterB and the withLatestFrom observable will execute on clusterA. This is where the cluster value defined on the RemoteObservable becomes useful. This value is used to check if both remote observables are executing on the same statemachine; if that is not the case, the numbers observable will need to push its events on clusterA.

In the example below, this step is performed by the observeOnSameRemote extension method, which checks if the source stream is executing on the same cluster as the stream passed as parameter. If they are, the source stream is returned unchanged, otherwise its events will be pushed on the cluster of the parameter by adding an observeOnRemote(clusterA) step.

Without keeping track of which cluster a remoteObservable is executing on, operators would have been forced to always act as if a stream executes remotely. This means that in order to make sure that both streams produce events on the same statemachine, operators would always need to call observeOnRemote on the second stream. This introduces unnecessary delay if the stream already executes on the same statemachine as the operator, since the events produced would create downstreamCommands, which would need to go through the replication process before being applied to the statemachine.

```
def withLatestFrom[S, R](other: RemoteObservable[S]): RemoteObservable[R]
    = {
  new RemoteObservable[R] {
    override def cluster = src.cluster
    override def subscribe(observer: StreamSetupObserver[R]): Unit = {
        val operatorId = UUID.randomUUID().toString
        other.observeOnSameRemote(src).subscribe(StreamSetupObserver(operators
            :+ WithLatestFrom(operatorId, RightStream()),
            observer.statemachineObserver))
        src.subscribe(StreamSetupObserver(operators :+
            WithLatestFrom(operatorId, LeftStream()),
            observer.statemachineObserver))
    }
  }
```

```
}
```

Considering the current approach, the case of operators that subscribe to two source remote observables raises an issue caused by the fact that the input subjects on the statemachine side are created when the Subscribe message is received.

This means that when the first of the two Subscribe commands arrives at the statemachine and the input subject is created and operators (including withLatestFrom) are applied, there will be no existing input subject for the 'other' stream (since it will be created at a later time, when its Subscribe command is processed).

Furthermore, because commands are sent asynchronously to the statemachine, it is also impossible to know which stream's Subscribe command will arrive first. Since not all operators that subscribe to two observable streams are commutative, it is important to know on which side of the operator the created input subject should be.

A solution to these issues is to assign a unique id to the operator that needs to subscribe to two streams and specify whether the input subject needs to be on the left or on the right side of the operator.

However, we still have the issue of the missing input subject for the second stream, which can be solved by creating a new subject that will be used as an intermediary: it will be subscribed to by the combining operator and it will subscribe to the end of the second stream when it will be created. We cannot use this intermediary subject directly as the input subject for the second stream, because the stream might still need to apply operators to its events before sending them to the combining operator.

To handle this case, the statemachine will keep an additional map from operatorId to RxStream that tracks 'partially subscribed' streams. Whenever a Subscribe message encounters a combining operator, it will first check to see if the map contains its operatorId.

In the case of the first Subscribe message, the operatorId will not be contained and thus an intermediary subject will be created. Next the information present in the operator (LeftStream/RightStream) will be used to decide on which side of the operator to place the newly created stream. Once the operator has been applied, the intermediary subject created previously, together with the observer that will be subscribed at the end of the stream are added to this 'partially subscribed streams' map using the operatorId as key.

When the Subscribe message for the second stream encounters the combining operator, the operatorId will be present in the map variable. At this point, the statemachine stops applying the next operators in the sequence and subscribes the input subject of the 'partially subscribed' RxStream to the stream that was just created. This essentially completes the creation of a remote combiner stream.

Finally, the observer part of the partially subscribed RxStream, together with the newly created input subject for the second remote stream are added to the statemachine streams variable using the streamId as key. The entry for operatorId in the partially subscribed stream map is no longer useful and can removed.

Sharing the observer between the two RxStreams that push values into the combining operator makes sure that the statemachine produces correct output whenever it pushes events to either of the two input subjects.

```
case class WithLatestFrom[T, S, R](operatorId: String, streamPosition:
    StreamPosition) extends CombiningOperator[T, S, R]{
  override def toObservable(left: Observable[Any], right:
      Observable[Any]): Observable[Any] = {
    left.withLatestFrom(right)
  }
}
```

On the operator implementation side, things are very similar to the case where operators subscribe to a single stream. There's only one operation needed, which needs to apply the appropriate transformation to the left and right observable streams.

7

Results and Evaluation

7.1. INTRODUCTION

We have designed and implemented a system that extends the Rx library with operators for fault-tolerant, remote execution of observable streams. In addition to the main design goal of enabling this new set of functionality, we have attempted to keep an interface that is consistent with the current Rx API while maintaining a small dependency footprint. The rest of this chapter will discuss the design decisions taken during development and possible alternatives.

7.2. TARGET RX LIBRARY

The reactive programming paradigm and the reactive extensions library stand at the core of this project and thus the first choice during planning was to decide which flavor of Rx to target.

The possible choices here were the original Rx.NET implementation[1], it's JVM counterpart, RxJava/RxScala[2], or RxMobile, a lightweight version of Rx with native implementations in both Java and Scala. RxMobile aims to offer the same core functionality as RxJava with a significantly simplified implementation by freeing itself from the concerns of backpressure. The lack of familiarity and experience with the .NET ecosystem and tooling made the JVM versions the only realistic choices. Between these two options, RxMobile was preferred due to it's decreased complexity. While the initial implementation of this project was in RxMobile, it was eventually ported to RxScala as part of the effort to open-source the project code source.

Considering the fact that this work is a proof-of-concept and is not intended for production use, we feel that the choice of target library does not really impact it's value. The ideas presented could pe taken and applied in any of the Rx flavors currently available.

7.3. CONSENSUS IMPLEMENTATION

Consensus is a fundamental problem in distributed computing systems that aim to achieve system reliability in the presence of faulty processes. There are a number of algorithms that address this problem, among which the most popular are Paxos[3, 4], Raft[5, 6], Viewstamped Replication[7–9] and Zab[10]. We feel that we couldn't have gone wrong with any of these options and any of these algorithms would have performed adequately for our requirements, but we ultimately decided to use Raft due to its focus on understandability and ease of implementation.

Once we have decided on a consensus algorithm, the next step was to choose a suitable implementation. The raft consensus website keeps a list of open-source implementations that are in various stages of development[11]. Out of these, the most notable and mature ones at that time were etcd/raft[12, 13], Logcabin/raft[14] and Copycat[15, 16].

Since this work targets the JVM, we decided against using etcd/raft or Logcabin/raft since their implementations are written in Go and C++ respectively. We ultimately decided to roll our own Scala-based implementation, using Rx to coordinate and orchestrate the asynchronous events in the system. This decision paid off in the end, allowing us to become familiar with the Raft consensus algorithm and easily extend it where needed.

On the more technical side, there were a number of decisions we had to take when

implementing different parts of the system. On the networking side, we followed the Raft paper[5], which describes the algorithm in terms of RPC calls between Raft clients and servers. There are quite a few RPC libraries/frameworks available in the JVM ecosystem, and we considered some popular choices which include the native Java RMI, Apache Thrift[17–19], Google gRPC[20, 21], Twitter Finagle[22, 23] and Akka[24, 25].

Out of these, gRPC and Finagle were the most interesting alternatives, being simple to use and configure, lightweight and offering streaming APIs for client-server communication. We had experience with neither of these 2 options and eventually went with gRPC, despite being the younger project of the two. The beta state of the gRPC library and tooling caused a few issues during development, which probably wouldn't have happened with a more mature library like Finagle, but these issues/bugs were patched relatively quickly.

Another technical decision that has influence on the design of the system is the choice of how remote client code is sent and executed on the server. For simplicity, our approach was to pack the whole client application and its dependencies as a jar archive and send it to the cluster during the client registration phase.

While this approach works in practice, it also has the drawback that applications with large dependencies will encounter a big delay during client registration.

7.4. API

This work was inspired by the observeOn operator in the standard Rx library, which is used to execute the downstream operators on a different scheduler. Our idea was to come up with an implementation for a similar operator (observeOnRemote), but which is capable of executing the downstream on remote machine(s), with support for fault-tolerance. One of our goals while implementing the Raft-based back-end system and the Rx operators meant to interact with it, were to keep the API close to the standard Rx library.

Despite being similar to the regular observeOn, the current iteration of the observeOnRemote operator has a few notable differences. Immediately noticeable is the fact that the standard observeOn operator takes a scheduler as a parameter, while observeOnRemote takes a raft Cluster as parameter. In a way, the observeOnRemote operator is closer in semantics to an observerOn(Scheduler.worker) operator.

In practice, this has an impact on the ease of use: in the case of observeOn the scheduler will create an internal worker and schedule subsequent work to be executed on it, while in the observeOnRemote case the 'worker' will be the remote cluster referred by the Cluster parameter. This means that when the observeOnRemote operator is called, the remote cluster must already be up and running.

This is something that could definitely be improved upon in future work. A possible solution would be to use cluster schedulers such as Apache Mesos[28–30] or Apache Yarn[31, 32] to dynamically allocate and start the Raft cluster workers.

Another important difference from a usability perspective comes from the fact that the downstream of observeOnRemote executes on a replicated Raft cluster. This places some restrictions on what operations can be performed in the downstream of observeOnRemote, and these restrictions which do not exist in the case of observeOn. In a nutshell, the observeOnRemote downstream cannot perform any operations that are blocking,

non-deterministic or non-terminating. This will be further discussed in the limitations section.

7.5. LIMITATIONS

While the work presented in this report shows a possible approach to adding fault-tolerant remote execution operators to the Rx library, it does not come without limitations. Stemming from various factors, such as deliberately limiting the project's scope or restrictions imposed by the consensus algorithm, these limitations can be used as inspiration for further research.

7.5.1. MISSING SUBSCRIBEON EQUIVALENT

The work presented in this report extends the Rx library with remote execution support by implementing an operator analogous to the vanilla observeOn operator. ObserveOn takes a scheduler as parameter and is used in an Rx stream to specify the scheduler on which an observer will observe the events of a stream. ObserveOnRemote works in a similar way, but it will push the stream to a raft Cluster instead of a scheduler. One of the important limitations of this work is that it does not offer an analogous version of subscribeOn, the other operator that the standard Rx library provides to control stream threading. This was a deliberate decision, made in an attempt keep the scope of the project manageable.

7.5.2. NO RX SCHEDULERS IN REMOTE STREAMS

One of the main requirements of the replicated state machine approach is that the state machine must be deterministic, since every server needs to arrive at the same result after applying the same series of commands.

As an example, the results of state machine commands must not depend on things like each server's current time, or command scheduling order. In the case of our system, this means that the client must not use any Rx schedulers or non-deterministic operators in the Rx query that will be executed on the Raft server. If a client would use such operations, then it becomes possible for Raft servers to execute commands in different order (depending on various factors like scheduling order and command execution timing), which would lead the replicated state machine into a corrupt state.

7.5.3. NO INFINITE OR BLOCKING OPERATORS IN REMOTE STREAMS

This limitation is closely related to the requirement of keeping the replicated state machine deterministic. In order to achieve that, state machine commands are applied sequentially, one at a time. This means that any command that blocks the current thread, or runs infinitely, will effectively prevent subsequent commands from being executed by the state machine. In this case the Raft servers will still be able to receive commands from clients and replicate them, but the system will not make any meaningful progress since the state machine will be 'stuck' executing the same (blocked) command. It is the responsibility of the user to make sure these situations will not arise.

7.5.4. LIMITED SET OF OPERATORS

The fact that the current implementation of observeOnRemote returns a RemoteObservable has the advantage of easily tracking which part of the stream executes locally and which executes remotely. On the other hand, this also means that any operators that we want to execute on the remote Cluster needs to also be defined on the RemoteObservable type. We have given examples of how to implement the two major types of operators: operators with a single source observable and operators that combine two sources. Since this work is a proof of concept, we have only implemented a limited set of operators, but an implementation targeting end-users would need to implement a wider range of the Rx operator set.

REFERENCES

[1] *Rx.net: The reactive extensions for .net,* https://github.com/Reactive-Extensions/Rx.NET.

[2] *Rxjava: Reactive extensions for the jvm,* https://github.com/ReactiveX/RxJava.

[3] L. Lamport, *The part-time parliament,* ACM Transactions on Computer Systems (TOCS) **16**, 133 (1998).

[4] L. Lamport *et al.*, *Paxos made simple,* ACM Sigact News **32**, 18 (2001).

[5] D. Ongaro, *Consensus: Bridging theory and practice*, Ph.D. thesis, Stanford University (2014).

[6] D. Ongaro and J. K. Ousterhout, *In search of an understandable consensus algorithm.* in *USENIX Annual Technical Conference* (2014) pp. 305–319.

[7] B. M. Oki, *Viewstamped replication for highly available distributed systems*, Tech. Rep. (DTIC Document, 1988).

[8] B. M. Oki and B. H. Liskov, *Viewstamped replication: A new primary copy method to support highly-available distributed systems,* in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing* (ACM, 1988) pp. 8–17.

[9] B. Liskov and J. Cowling, *Viewstamped replication revisited,* (2012).

[10] F. P. Junqueira, B. C. Reed, and M. Serafini, *Zab: High-performance broadcast for primary-backup systems,* in *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on* (IEEE, 2011) pp. 245–256.

[11] *Raft consensus algorithm implementations,* https://raft.github.io/#implementations.

[12] *etcd documentation,* https://coreos.com/etcd/docs/latest ().

[13] *etcd: Distributed reliable key-value store for the most critical data of a distributed system,* https://github.com/coreos/etcd ().

[14] *Logcabin - distributed storage system built on raft*, https://github.com/logcabin/logcabin.

[15] *Copycat - novel implementation of the raft consensus algorithm for java 8*, http://atomix.io/copycat/ ().

[16] *Copycat: A novel implementation of the raft consensus algorithm (source code)*, https://github.com/atomix/copycat ().

[17] M. Slee, A. Agarwal, and M. Kwiatkowski, *Thrift: Scalable cross-language services implementation*, Facebook White Paper **5** (2007).

[18] *Apache thrift: Lightweight, language-independent software stack for rpc*, https://thrift.apache.org/ ().

[19] *Apache thrift: Lightweight, language-independent software stack for rpc (source code)*, https://github.com/apache/thrift ().

[20] *grpc - a high performance, open-source universal rpc framework*, http://www.grpc.io/ ().

[21] *grpc - the java grpc implementation. http/2 based rpc (source code)*, https://github.com/grpc/grpc-java ().

[22] *Finagle: Extensible rpc system for the jvm*, https://twitter.github.io/finagle/ ().

[23] *Finagle: A fault tolerant, protocol-agnostic rpc system (source code)*, https://github.com/twitter/finagle ().

[24] *Akka: Build highly concurrent, distributed, and resilient message-driven applications on the jvm*, http://akka.io/ ().

[25] *Akka source code*, https://github.com/akka/akka ().

[26] *Apache spark - lightning-fast cluster computing*, https://spark.apache.org/ ().

[27] *Apache storm*, http://storm.apache.org/.

[28] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, *Mesos: A platform for fine-grained resource sharing in the data center.* in *NSDI*, Vol. 11 (2011) pp. 22–22.

[29] *Apache mesos: A distributed systems kernel*, https://mesos.apache.org/ ().

[30] *Apache mesos: A distributed systems kernel (source code)*, https://github.com/apache/mesos ().

[31] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, *Apache hadoop yarn: Yet another resource negotiator*, in *Proceedings of the 4th annual Symposium on Cloud Computing* (ACM, 2013) p. 5.

[32] *Apache yarn: A resource manager for the hadoop cluster*, https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html ().

8

CONCLUSIONS AND FUTURE WORK

8.1. CONCLUSIONS

The goal of this thesis was to show a possible approach to designing and implementing a system that allows parts of an Rx stream to be executed remotely in a fault-tolerant fashion. As part of this work, we have shown an implementation for a distributed state machine capable of running Rx queries together with extensions to the Rx library that allow users to orchestrate and compose local and remote streams.

We believe our approach and contributions are valuable for a number of reasons. First of all, it opens up new use cases for the Rx library, enabling users to fault-tolerantly execute parts of a query or parallelize Rx streams over multiple physical machines.

Secondly, under the hood our system relies on a novel implementation of the Raft consensus algorithm that allows users to run generic distributed state machines and adds support for message forwarding and remote code execution. While there are other implementations of Raft that allow users to run generic statemachines (such as Copycat[1] or akka-raft[2]), we do not know of any that provide these features.

Additionally, since the infrastructure of our system was build from the ground up, it relies on a minimal set of dependencies (grpc, protocol buffers and Rx). We believe this helps with the understandability of the system and makes it a valuable learning tool for developing distributed systems. This approach also allows for new features or extensions to be implemented with relative ease, as evidenced by the two extensions described in chapter 4.

Last, but not least, we believe this work also has value as a foundation for future abstractions that could be developed on top of the remote execution operators. One particular interesting possibility is the implementation of cluster schedulers that would efficiently distribute a stream over multiple machines.

8.2. FUTURE WORK

While we believe the implementation presented in this report brings a valuable contribution to the Rx environment, there are also numerous opportunities to improve and extend it with new features.

First of all, the usability of operators like observeOnRemote could be improved by integrating with cluster schedulers like Mesos and Yarn. This could make the observeOnRemote api more similar to the standard observeOn and would free the user from having to statically identify and manage the cluster that will execute the downstream operators.

A second area for potential future work is related to the 'no schedulers' limitation discussed in the evaluation section. With the current implementation the user cannot use Rx schedulers in the downstream of observeOnRemote because that would make the remote state machine non-deterministic and could corrupt the distributed state. However, one possible way around this limitation would be to implement deterministic schedulers which would assure that the state machine replicas execute the same commands and produce the same outputs.

Another potential area for improvement is sandboxing the Raft worker process. The fact that the Raft statemachine can execute arbitrary user code poses a risk on the whole system. Sanboxing would mitigate this risk and make the system more robust and stable overall.

Finall, we hope that this work has sparked the reader's interest towards distributed systems and reactive programming. The standard Rx library offers an elegant API for manipulating and orchestrating streams of asynchronous events and provides standard operators that allow developers to control which schedulers execute the stream. We believe that the next logical step is to implement similar abstractions that operate at machine and cluster level instead of just threads. This work has shown a possible approach towards reaching this goal. While by no means complete, or production ready, we believe it is valuable as a starting point and inspiration for implementers looking to address this challenge.

REFERENCES

[1] *Copycat - novel implementation of the raft consensus algorithm for java 8,* http://atomix.io/copycat/.

[2] *Akka-raft - a raft implementation on top of akka cluster,* https://github.com/ktoso/akka-raft.

ACKNOWLEDGEMENTS

First and foremost, I would like to offer my gratitude to prof. Erik Meijer not only for his help and guidance during this project, but also for all the time we spent hacking and discussing technical topics at 't Postkantoor. Working together with you has sparked my interest in a number of computer science topics and has had a great impact on my technical skills and attitude towards our field. I'll always be grateful for the opportunities you presented me and the role you played in my development as a programmer.

Secondly, I want to thank Georgios Gousios for his time, honest advice and encouragement throughout the project. His feedback and comments proved very valuable and our discussions were always motivating.

I also want to thank Eddy and Georgi for all the great moments we spent together, be it hacking, learning crazy new things or doing it live. It's usually hard to find people that share the same enthusiasm for technology, programming and computer science, and I am happy that I did. This Master's program would have definitely been a lesser experience without you guys.

I would also like to thank my parents and family for their continued encouragement and unconditional support throughout my studies. I owe special thanks to my lovely girlfriend, Raluca, for motivating me and helping me get through any difficulties. And finally, I want to thank my friends and housemates for making these years enjoyable and for making Delft feel like home.

Sincerely,
Mircea Voda