IN4391 Distributed Systems
An implementation of a graph-based CRDT in Scala

Abri Bharos
4665392
a.r.j.bharos@student.tudelft.nl

Hakan Ilbaş
4714245
h.ilbas@student.tudelft.nl

Erwin Russel
4973976
e.f.j.russel@student.tudelft.nl

Andrzej Westfalewicz
5247179
a.westfalewicz@student.tudelft.nl

Abstract

This paper demonstrates the concepts and the implementation of a graph-based Conflict-free Replicated Data Type (CRDT). The implementation is done in Scala using the HTTP services of the Akka Actor framework. Next to explaining what a graph-based CRDT is and why it achieves strong eventual consistency (SEC), this paper goes over the approach of implementing such a data structure and verifying that this implementation is correct. The implementation is verified using integration tests and demonstrates behavior adhering to SEC. The system is also benchmarked by comparing it to a popular graph database engine - Neo4j. The project, alongside, more content can be found at github.com/HakanIlbas/DS_project.

1 Introduction

A distributed computing system is a system whose components are located on different networked computers, which communicate and coordinate their actions. As such, the distributed system will appear as if it is one interface or computer to the end-user [9]. To achieve this appearance of a single system, it is therefore necessary to achieve consistency, among other capabilities. A Conflict-free Replicated Data Type (CRDT) is a data structure that can be replicated on the aforementioned distributed computing system. The single instances acting as replicas of each other can be updated independently and concurrently without coordination. With CRDTs it is always possible to mathematically resolve inconsistencies that might come up [8].

A graph-based CRDT incorporates a graph structure with vertices and arcs that can be replicated conflict-free over a distributed system. Such graph-based structures are useful for distributed algorithms such as Google's PageRank Algorithm or other many-to-many relational networks as can be found in online social networks [8]. This paper demonstrates the implementation of a graph-based CRDT in Scala and the evaluation of this implementation.

Section 2 goes in-depth about what a graph-based CRDT is and how it mathematically proves Strong Eventual Consistency. Section 3 gives a short analysis of relevant work in the field of CRDTs. Section 4 explains the implementation details of our implementation and the functional tests. Section 5 shows the non-functional performance tests of the system and their results. Finally, section 6 concludes this paper and discusses where to go next.

2 The Graph-Based CRDT

This section will analyze the paper by Shapiro et al. "Conflict-free Replicated Data Types" [8] based on which this paper is written. It will also justify the choice of implementing the directed graph CRDT.

2.1 Theoretical discussion

First the paper states the problem of eventual consistency, which allows any replica in a distributed system to accept updates without remote synchronization, of being very ad-hoc and error prone. A solution to the CAP problem and Eventual Consistency is proposed, namely Strong Eventual Consistency (SEC). A data type that satisfies this condition is called a Conflict-free Replicated Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite the operations performed or any number of failures.

Then the paper poses the system model which is a system of processes interconnected by an asynchronous network. The network can partition and recover. The replicas in this asynchronous network are then represented by a state-based model denoted by a tuple (S,s_0,q,u,m), where S is the set of states called payload, s_0 the initial state, q the query method, u the update method and m the merge method.

A couple of definitions are set up such as *causal history* which states that a state-based object has a causal history which appends updates to the replica and merges local and remote histories by union. An update is delivered when it is included in the causal history of a respective replica. Next *eventual consistency* is defined as a combination of eventual delivery (updates are delivered to all correct replicas), convergence (equivalent state is eventually reached), termination (all methods terminate). Then *strong eventual consistency* is achieved when eventual consistency is achieved together with strong convergence (correct replicas with the same updates have the same state).

The paper then proposed two types of models: state-based convergent replicated data type and operation-based commutative replicated data type (CmRDT). The latter, which relates to graph CRDTs - the topic of this report, is defined by a tuple: (S, s_0, q, t, u, P),where S, s_0 and q same as previously mean state domain, initial state and query method. n op-based object has no merge method; instead an update is split into a pair (t, u), where t is a side-effect-free prepare update method and u is an effect-update method. This separation was a motivation for our implementation of the DataStore (Section 4.1). The paper the defines causal history and commutativity for the new model and proofs that CmRDT is SEC. The theorem mentions a very important condition: "causal delivery of updates". This means that operations delivered to one instance in a given order have to arrive to all other nodes in the same order.

The paper ends the theoretical discussions by stating and proving these facts: SEC is a solution to the CAP problem, CvRDTs and CmRDTs are equivalent, SEC is incomparable to sequential consistency. Note that the first fact can be considered only partially true, as the replicas are not consistent until the synchronization occurs, even though the synchronization is guaranteed to succeed.

2.2 Directed graph CRDT

The paper then continues on with giving a few examples of different CRDTs: integer vectors, counters, U-set and the directed graph CRDT, which is the topic of this report. The authors begin the discussion on the Directed graph CRDT by suggesting its use case in page rank algorithm. This was also their motivation to allow in the data structure design for arc to non-existing vertices as page links on websites can point to nowhere or may not yet be added to the graph. The design is then further discussed by elaborating on the concurrent addition of an arc and a removal of that arc's source. Three different suggestions were made where the one in which removing the vertex takes precedence over adding an arc was argued to be the most reasonable. This scenario was turned into a test case in the system (shown in the appendix A.1). After the these discussions an implementation of a directed graph is discussed which bases on having sets of unique ids for every vertex or arc and translating all the operations to work on these sets. These operations are described in pseudo-code. Finally the proposed design is proven to be a CRDT by showing how problematic cases(e.g. concurrent addition and removal of a vertex) commune. The paper concludes with a comparison to previous work, summary of the paper and future work suggestions.

2.3 Motivation for this report

In computer science, a graph is an abstract data type that represents an graph which contains edges and directed vertices. Some important applications and algorithms need to use graphs as an abstract data type some mentioned in the summary above. However graph-based CRDTs are not very common. While some popular systems, like Redis[1] or DynamoDB[2], are using CRDTs, there are no popular solutions for graph-based CRDTs. Therefore it would be interesting to implement it. Shapiro et al. [8] already gives a full implementation in pseudo-code for a graph-based CRDT. Our implementation however, contains some small differences when comparing it to [8]. We will explain the implementation of our graph-based CRDT later on.

3 Related Work

As explained before, this paper explores CRDTs by studying conditions for convergence under a SEC model. It studies a number of interesting CRDT implementations. Since our system is implementing the directed graph data structure, the most relevant implementations for this report are the graph-based CRDTs. There has been some other work done in this specific area as well, however the graph data isn't available in these solutions as a service. In this section we will explore some of this work and give an overview of it.

An Optimized Conflict-Free Replicated set [3] on cloud computing platforms and claims that "solutions for addressing concurrent updates tend to be either limited or very complex and error-prone". It therefore introduces a set-based CRDT implementation that can be transposed to other data types (such as sequences, maps and graphs).

FabricCRDT: A Conflict-Free Replicated Datatypes Approach to Permissioned Blockchains [5] focuses on blockchain technologies, more specifically on Hyperledger Fabric. This system makes use of an Execute-Order-Validate lifecycle, which has latencies that could potentially bottleneck scalability. They claim CRDTs provide a solution here, which is why the authors introduce FabricCRDT. This approach integrates CRDTs into Fabric. They show that this offers higher throughput, without introducing failures.

Specification and Verification of Convergent Replicated Data Types [10] presents techniques for verifying CRDTs. This verification covers convergence properties, as well as behavioral specifications. Its contributions are mainly machine checked proofs for the correctness of CRDTs.

A Conflict-Free Replicated JSON Datatype [4] focuses heavily on practical applications of CRDTs, especially within JSON data structures. They claim that modifications that are made to the data structures are well-understood when there is a single copy of the data, but that this is not the case when the data is replicated and modified on multiple devices.

[1]https://redis.io/
[2]https://aws.amazon.com/dynamodb/

They therefore introduce a CRDT algorithm that takes care of client-side merging.

4 Implementation

This section contains four subsections explaining the implementation of our graph based CRDT. Every subsection explains the implementation of a certain module in the system. The general view on the implementation is shown in figure 1.

4.1 DataStore

The DataStore module contains the main logic for the graph-based CRDT. This module contains (among other functions), 8 functions which are used to create and manipulate the graph. All operations within one instance have to be performed in linear succession. Since the DataStore object is accessed by multiple threads from Akka and from the Synchronizer (section 4.2), each of these functions have a lock. The DataStore object contains a few more private functions, but these are called by the public functions and have no functionalities on their own. The general idea behind our implementation for the public methods is to create commands (i.e. OperationLog objects), which are the side-effect free methods from the theoretical discussion. These commands are then executed by the private methods, which are effect-update methods. The same commands are then synchronized with the other instances, which perform an update on their DataStores with the same private functions.

- `addVertex(vertexName)`: this function is called to add a new vertex to the graph with the name given in the vertexName parameter. If there is already a vertex with the same name, the vertex remains the same, but only an additional ID is added to the vertex.
- `addArc(arcSourceVertex, arcTargetVertex)`: this function is called to add a new arc to the graph from the vertex called arcSourceVertex to the vertex called arcTargetVertex. To add an arc, only the source needs to exist in the graph. The target does not need to exist yet, however the arc will remain hidden until the target is added. This was conscious adaptation made by the authors of [8] to better fit many use cases including the page rank algorithm. If there is already an arc with the same source and target, the arc remains the same, but only an additional ID is added to the arc.
- `removeVertex(vertexName)`: this function is called to remove a vertex from the graph with the name given in the vertexName parameter. This function will perform a cascade delete (i.e. will also remove all arcs starting from the vertex vertexName). Note that this is a disparity from the original design where there was a precondition for the vertex to have no arcs, but we believe this modification will make the service easier to use. It also allowed for an easier implementation of the synchronization of this operation, as the removeVertex operation is preceded by a list of removearc operations and each instance will append own not-yet-deleted arcs to the list. The operation fails and returns false whenever the vertex does not exist.
- `removeArc(arcSourceVertex, arcTargetVertex)`: this function is called to remove an arc from the graph, from the vertex called arcSourceVertex, to the arc called arcTargetVertex. To remove the arc the arc needs to exist. If the arc does not yet exist, the function will return false and the operation will fail.
- `lookUpVertex(vertexName)`: this function checks whether the vertex with vertexName exists in the graph and returns true if it does, and false if it does not exist in the graph.
- `lookUpArc(arcSourceVertex, arcTargetVertex)`: this function is similar to the function above, but then for arcs. This function looks up if there is an arc present in the graph from arcSourceVertex to arcTargetVertex, and returns true if it does, and false if the arc does not exist in the graph. As mentioned before, if the target doesn't exist the arc is considered hidden and the function will return false.
- `applyChanges(changes)`: this function is called during the synchronization with other instances. The changes parameter is a list of all the operations that a node needs to apply to its DataStore. The applyChanges function uses private functions to manipulate the state of the DataStore to not cause any deadlocks.
- `getLastChanges(skip)`: this function is called by the Synchronizer 4.2 to acquire the operations performed on that DataStore since the last synchronization. The skip parameter indicates how many elements from the beginning of the queue of changes should be omitted.

It is worth to note that the private functions, although they mirror the functionalities of the public ones, do not perform the same checks. For example applyRemoveVertex will not fail if the vertex to be removed is missing. This discrepancy is caused by the fact that the private functions are called during synchronization process. For example if a vertex is concurrently deleted on two instances and the instances perform synchronization, each instance will get the message to remove an already removed vertex. The strategy is to not modify the current states (as it is already correct - the vertex is deleted) and mark the synchronized operation as processed. We have chosen to return false when the public removeVertex function is called, to improve usability and to give users more feedback on the state of the DataStore.

The graph structure is represent in the edge list format. The vertices are stored within a HashMap indexed by their name. The reason for the HashMap is for efficiency, such that we do not have to perform a linear search. The Vertex class is a container for an array of unique ids and a HashMap

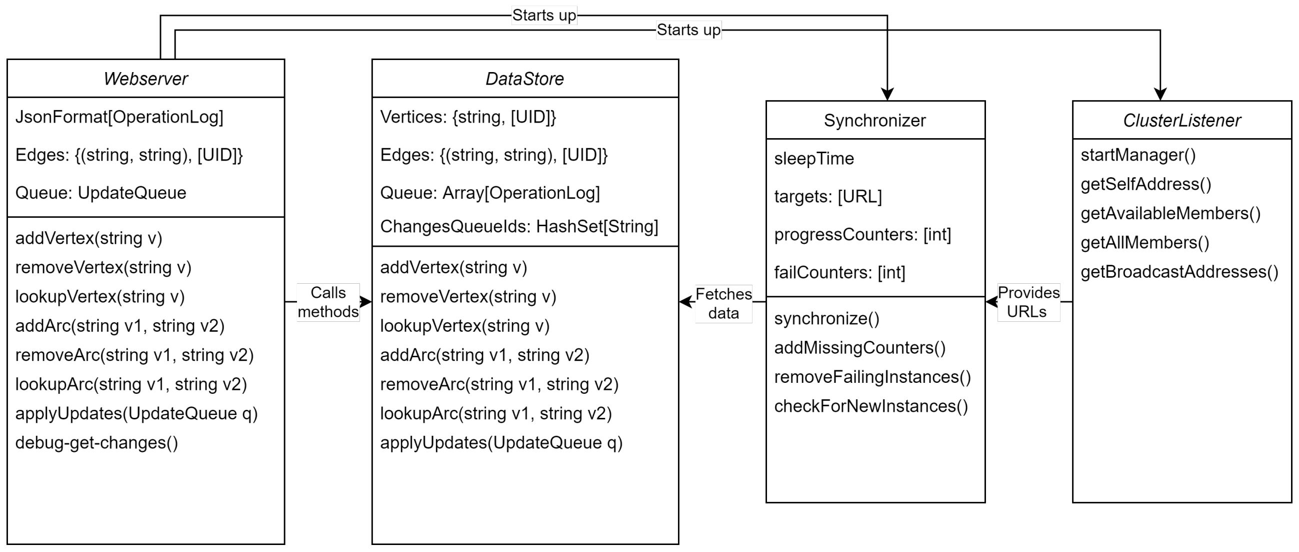

Figure 1. Diagram of the main classes in the implementation

of outgoing arcs. Arcs do not have their own class - their structure is limited to a single array of ids.

OperationLogs are stored within a single table as their order has to be retained. To remove linear search on this array we decided to duplicate the ids of the operation into a HashSet already containing applied changes, and perform searches on that HashSet.

4.2 Synchronizer

The role of the Synchronizer module is to broadcast operations (e.g. addArc, removeVertex etc.) that are performed on a given instance, to the other instances and as a result create a shared common state. All performed operations are being queued with the DataStore module to be later retrieved using the getLastChanges function (which we described earlier) by the Synchronizer. The Synchronizer will then make HTTP POST requests to other instances causing them to call the function applyChanges on their DataStores. This process repeats after a configurable amount of time (time between synchronization). Additionally, when running on Kubernetes, the Synchronizer periodically checks for new instances within the cluster. Finally an instance that fails to respond OK to many consecutive calls will be removed from the synchronization list.

4.3 WebServer

This module contains the HTTP server of the system. Since we are constrained to use Scala for the project, we are limited in the amount of HTTP servers we can choose from. We started with http4s [2], since it seemed easy and intuitive, but we eventually switched over to Akka HTTP [1], since we struggled with (de)serialization because of a lack of good documentation with http4s. Akka HTTP seemed easy and intuitive, just as http4s, but the documentation was also excellent and contained a lot of good examples to help us getting started.

Our HTTP server exists of a single file. This file contains the formats to (de)serialize JSON objects, as well as all of the routes we use for our API. In total we have 9 routes which are proxies for the respective methods in the DataStore module. Their responsibility is to extract the parameters form the requests and provide an abstraction level between the data and HTTP requests. These routes are:

- POST on "/addvertex": this route expects a HTTP POST request with a JSON body including a single vertex in the following format: {"vertexName": "abc"}. If successful the server will return with the string "true", otherwise the server will return "false" with HTTP status code 400.
- POST on "/addarc": this route expects a HTTP POST request with a JSON body including a single sourceVertex, and a single targetVertex in the following format: {"sourceVertex": "abc", "targetVertex": "xyz"}. If successful the server will return with the string "true", otherwise the server will return "false" with HTTP status code 400.
- DELETE on "removevertex": this route expects a HTTP DELETE request with a JSON body including a single vertex in the following format: {"vertexName": "abc"}. If successful the server will return with the string "true", otherwise the server will return "false" with HTTP status code 400.
- DELETE on "removearc": this route expects a HTTP DELETE request with a JSON body including a single sourceVertex, and a single targetVertex in the following format: {"sourceVertex": "abc", "targetVertex":

"xyz"}. If successful the server will return with the string "true", otherwise the server will return "false" with HTTP status code 400.

- POST on "applychanges": this route expects a HTTP POST request with a JSON body with a serialized array of OperationLog objects. If successful the server will return with the string "true", otherwise the server will return "false" with HTTP status code 400. This route is called by the Synchronizer.
- GET on "lookupvertex": this route expects a HTTP GET request with a query parameter in the format "vertexName=abc". If the vertex exists in the graph, the server will return "true" and "false" otherwise.
- GET on "lookuparc": this route expects a HTTP GET request with query parameters in the format "sourceVertex=abc&targetVertex=xyz". If the arc exists in the graph, the server will return "true" and "false" otherwise.
- GET on "debug-get-changes": this route expects a HTTP GET request with no parameters. It will return a JSON array with all operations performed on this instance. This method should be used only for development purposes.
- GET on "address": this route expects a HTTP GET request with no parameters. It will return a text message containing the address of this instance and other instances within the Kubernetes cluster. This method will fail if it is not executed in a Kubernetes environment and should be used only for development purposes. If the system is not in a Kubernetes environment it will return "false" with HTTP status code 400.

When we start up a server we can also specify on which port it should start up. We use the following command to start up a server at port 8080:

```
sh ./sbt_run.sh 8080
```

The reason to why we have a script calling sbt is that Kubernetes needs a special configuration file to start up. However this configuration does not work with sbt. And thus we need two different configuration files. The script makes sure to use the correct configuration file, and then starts up the application with sbt.

When the API is running, clients can make HTTP request to the service. For testing the API can be accessed with for example cURL or Postman. Changes take immediate effect on the node which receives a request. Synchronizing with the other instances depends on the configuration of the synchronizer. This needs to be changed manually in the Synchronizer code before compilation.

4.4 Kubernetes

In order to test the system in a closer to real world setting, the choice was made to deploy the CRDT implementation on Kubernetes[3]. Kubernetes is an open-source container-orchestration system for automating computer application deployment in something called pods, scaling of these pod deployments, and cluster management.

To deploy our application, we first had to package it into a Docker[4] container, Docker allows for OS-level virtualization to deliver software in packages called containers. The Akka framework allows for automatic packaging into a docker container which can be deployed on Kubernetes. Minikube was used to host the Kubernetes deployment locally, which allowed for testing and verifying the implementation.

The Kubernetes deployment is setup with configuration files that specify roles, deployment and services. Role-based access control (RBAC) is a method of regulating access to computer or network resources based on the roles of individual users within a organization. A deployment describes a desired *state*, for instance, one would like to have 5 replicas of a certain Docker image running. Services define a policy by which the cluster can be accessed, these define e.g. a load-balancer, which balances incoming request over the pods in a cluster. When testing the CRDT, it is necessary to talk to pods directly. One should be able to choose a pod and perform an update, while choosing another pod and see if the Synchronizer has pushed the updates to the respective pod. To achieve this, port forwarding was used which allowed to forward local host ports to incoming ports of a pod.

In the repository there is a shell script that starts the Minikube environment and pushes the Kubernetes deployment (which is explained more in-depth in the next paragraph). Furthermore, it builds the Docker container with the Scala application that is pushed to Kubernetes and waits for the cluster to start the pods. When the pods are started, the script requests the pod identifiers to open the port forwarding.

The Akka framework uses the Kubernetes API to discover other members of a cluster, this is used in our implementation to let the Synchronizer know which other addresses we need the broadcast the operations. Kubernetes doesn't provide replication services fitting our use case so custom synchronization logic had to be implemented in the Synchronizer for this project. However, Kubernetes is used by over 25,000 companies which makes it the go to framework for cluster orchestration.

4.5 Evaluation and Testing

To test our implementation we designed a set of Python scripts executing specific integration test cases. These scripts arrange specific states on instances by performing request, wait for the synchronization and perform more operations or verify the state. Many tests are designed to cause conflicts in normal data structures - such as adding and deleting a vertex

[3]https://kubernetes.io/

[4]https://www.docker.com/

concurrently. Other test will perform seemingly easy operations to check if the behavior of basic operation matches the one described in the original design. Note that while executing these test one has to also observe the output of the instances detect potential error messages.

The test were especially useful as they detected a typo when we were changing the implementation of some core logic. The full list of these test cases can be found in the appendix A.1.

5 Benchmarking

This chapter describes the non-functional testing that we did besides the test cases mentioned above (Section 4.5). These test are verifying the performance in different configurations and investigate how the performance changes with some parameters. For the test data, we have generated an instructions set that creates and operates on graphs similar to a Erdős–Rényi random graph[7]. All used scripts and datasets can be found in the repository in the benchmarking folder.

5.1 Comparison to an other system (Neo4j)

To compare to other implementations, a Kubernetes implementation would be preferred since it allows for similar runtime environment and easy deployment scaling for scalability testing. Many CRDT implementations have been considered but were not related to graph structures and others featured incomplete repositories.

Ultimately, it was chosen to compare our system against the well-respected Graph Database Neo4j, since there was a ready-made implementation on Kubernetes which could be used for benchmarking. While it is not a CRDT, it implements the same data structure of a graph with nodes and relations which allows for running the benchmarking experiments one to one. Furthermore, Neo4j is the world's most widely deployed graph database used by Walmart, eBay, Adobe and more, which adds credibility to the comparison of the graph-based CRDT implementation.

The implementation uses RAFT [6] as a consensus algorithm, whereas our CRDT achieves consensus by design. This also mean that in the clusters that are run locally, the Neo4j Core replicas feature a Leader and Followers. Consequently, only one Pod in the Kubernetes cluster allows for both read and write operations, whereas all the other pods only allow for read operations.

5.2 Scalability

First, we want to discuss the performance gain of adding a new instances to the system. As the instances are broadcasting their changes to all other instances in the cluster we expect a drop in performance gain for large numbers of instances. Our tests consisted of two stages: write-heavy stage, when the instances were receiving a lot of write requests and read-heavy stage, when the instances were handling mostly

Number of instances:	2	4	6	8
Minimal CRDT:	4	3	4	4
Average CRDT:	4.46	5.67	6.04	6.38
Maximal CRDT:	31	96	128	98
Minimal Neo4j:	13	14	13	16
Average Neo4j:	31.81	33.51	36.52	45.31
Maximal Neo4j:	4799	5937	14976	6657

Table 1. Scalability results in read-heavy scenario.

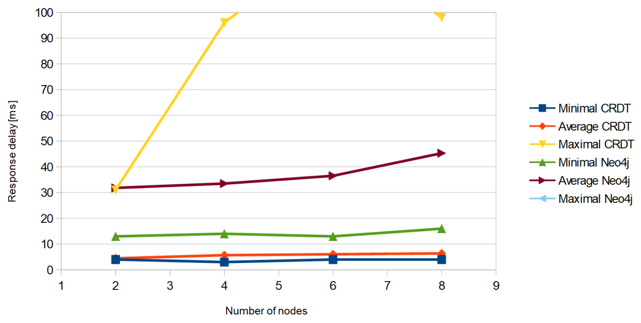

Figure 2. Scalability results in read-heavy scenario

lookups. The testing scripts records the minimal, average and maximal response times for operations in both stages.

The summary of the read-heavy tests is shown in table 2 and in the figure 2, while the write-heavy results are shown in table 3 and in the figure 3. All results can be found int the projects repository.

From figure 2 we can see that our graph-based CRDT implementation performs much better than Neo4j in read-heavy scenarios. Especially when we compare the maximum response delay between our implementation and Neo4j. Our implementation its maximum response delay is 128 ms for 6 instances, whereas Neo4j its maximum response delay is through the roof. We cannot reasonably show it in the graph because its in the thousands (with the maximum value of 14976ms). Our implementation its response delay is better in every case whenever we compare it to Neo4j, mostly thanks to the fact that we are using in memory storage while Neo4j goes for persistent storage. We can see that our response delay also spikes at 6 instances (where Neo4j goes through the roof), however our average response time is very close to our minimal response time, which shows how lightweight our implementation is.

The situation in write-heavy scenarios is very similar to read-heavy scenarios. Once again the CRDT in general performed better than Neo4j, because it benefits from the in-memory representation. The major difference is that the maximal response times for our CRDT implementation are somewhat similar to the maximal times in tests on Neo4j. This is caused by the need to synchronize the updates, which

Number of instances:	2	4	6	8
Minimal CRDT:	4	4	4	4
Average CRDT:	4.75	7.17	11.93	14.13
Maximal CRDT:	69	283	4593	1105
Minimal Neo4j:	13	14	18	21
Average Neo4j:	24.39	35.10	48.21	52.28
Maximal Neo4j:	1099	1339	2185	8172

Table 2. Scalability results in write-heavy scenario.

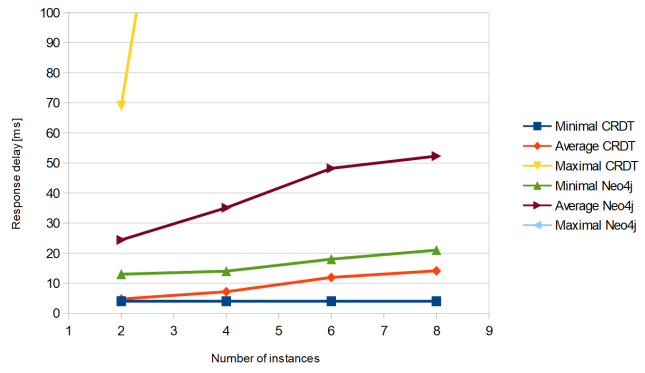

Figure 3. Scalability results in write-heavy scenario

blocks the handling of the request for a given time. The average response time increases approximately 3 times (from 4.75 to 14.13 ms), while the number of nodes rises 4 times (from 2 to 8 instances), which represents a good trend for scalability in small clusters.

5.3 Time to consistency

A typical feature in eventual consistency systems is that the state among the instances temporarily diverges as operations on one instance are performed on other instances after some time. We designed a test to find out how long it takes for the operations to be broadcasted to other instances.

In the test, the instances are constantly being called with request similar to those in the scalability test, which represent the usual traffic the system may deal with. While this is happening, a script will periodically insert a specific vertex into on of the instances and then start to ask for it on another instance. The script records minimal, average and maximal times between the addition and first positive lookup. We expected that the time between synchronizations is a huge factor to these values, so we run the test for different configurations - with time between synchronization equal to 500ms, 1s, 2s and 4s.

The tests were performed on a single desktop machine with Windows 10 operating system, Intel Core i7-1065G7 processor, 12GB 3200MHz RAM. Full results can be verified in the repository, while the summary is displayed in table 3 and in figure 4.

Time between synchronizations:	500	1000	2000	4000
Minimal:	10	10	11	48
Average:	272.5	549.9	1053.87	1991.92
Maximal:	522	1035	1969	4002

Table 3. Time to consistency against time between synchronizations

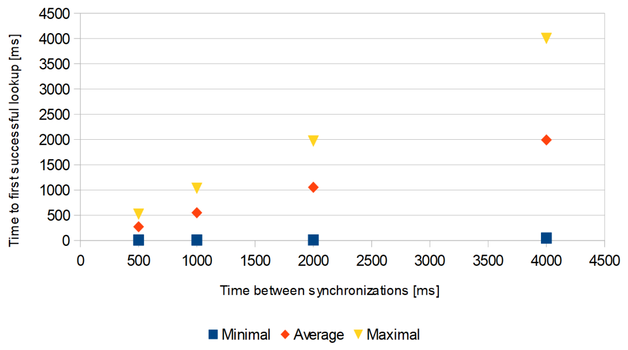

Figure 4. Time to consistency against time between synchronizations

Number of instances:	2	4	6	8
Minimal:	12	10	9	16
Average:	246.46	273.5	295.95	387.6
Maximal:	510	522	580	1295

Table 4. Time to consistency against number of instances

For 500ms, 1000ms and 2000ms, the smallest time to first successful lookup was within 10ms, which means that the tests performed only one failed lookup and the synchronization had to occur just after adding the vertex. In the 4000ms this situation is much less likely and the minimal time is 48ms. The average time to consistency is in all cases around half the time between synchronizations. This is also expected as the requests were sent randomly within the period and the lookup will succeed after the first serialization.

We also run the same test for different instances count and the time between synchronizations set at 500ms. Again full results can be verified in the repository, while the summary is displayed in table 4 and in figure 5. Here you can see steady rise of the average time, as the instances sometimes had to handle the synchronization request from the instances before they could have handled their own requests, until the limitations of the host machine start to affect the test. However the number of required synchronizations goes up with a square of the number of instances, which can be seen already for smaller clusters. To avoid major performance drop for

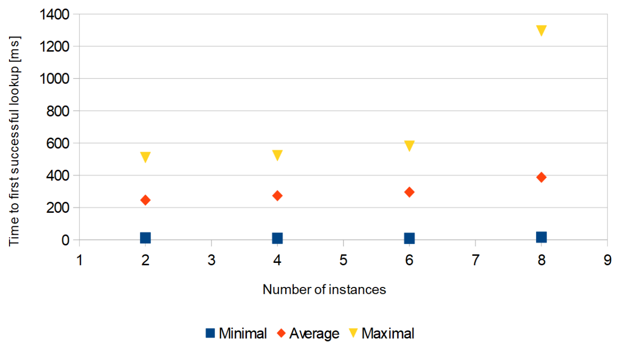

Figure 5. Time to consistency against number of instances

large clusters the time between synchronizations should be lower. This issue can be partially solve by re-implementing the synchronization logic to perform the update to only one other instance at a time and performing synchronizations more frequently - the changes could then propagate among the instances by taking routes with multiple hop counts.

The same test when run on the Neo4j system with 4 instances returned results: minimal time 22ms, average time 55.50 ms and maximal time 194ms. As Neo4j employs different synchronization strategies between the write replica and the read replicas this result can't be compared to our system, however based on previous results one can say that to achieve the same performance the time between synchronizations in our system has to be configured to around 100ms.

6 Conclusion

As distributed datastores are a necessity for large systems, the directed graph CRDT offers a solution for this type of data. It gives the assurance that the saved data will not be removed due to a conflict. In this paper we analyzed the solution proposed by Shapiro et al. [8] and designed a system based ob it. We discussed other solutions in the field and found that there is none exactly like ours. We implemented the system in an efficient way and tested it's correctness. We also tested its performance to find out that it performs better in specific scenarios than other available solutions.

Although it is fully functional, we see this system as a proof of concept. The data structure has proven to be correct and conflict resolutions were not necessary. We already mentioned some improvements in the benchmarking section (Section 5), but some were not addressed. We were restricted by our assignment to implement the solution in Scala, but seeing the simplicity of the DataStore implementation it might be a better choice to implement it in a faster and simpler language such as C. The system has also potential for some additional features. For example returning all neighbours of a node allows to search the graph with breadth-first search or depth-first search. Also, allowing to store data with the unique id for each vertex or arc would make the system more useful as a database.

References

[1] [n.d.]. https://doc.akka.io/docs/akka-http/current/index.html

[2] [n.d.]. Typeful, functional, streaming HTTP for Scala. https://http4s.org/

[3] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. 2012. An optimized conflict-free replicated set. arXiv:1210.3368 [cs.DC]

[4] Martin Kleppmann and Alastair R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (Oct 2017), 2733–2746. https://doi.org/10.1109/tpds.2017.2697382

[5] Pezhman Nasirifard, Ruben Mayer, and Hans-Arno Jacobsen. 2019. FabricCRDT: A Conflict-Free Replicated Datatypes Approach to Permissioned Blockchains. https://dl.acm.org/doi/10.1145/3361525.3361540

[6] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

[7] A Rényi P Erdős. 1959. On random graphs I. https://www.renyi.hu/~p_erdos/1959-11.pdf

[8] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *Conflict-free Replicated Data Types*. Research Report RR-7687. 18 pages. https://hal.inria.fr/inria-00609399

[9] Maarten van Steen and Andrew S. Tanenbaum. 2017. *Distributed systems*. Maarten van Steen.

[10] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2014. Formal Specification and Verification of CRDTs. In *34th Formal Techniques for Networked and Distributed Systems (FORTE) (Formal Techniques for Distributed Objects, Components, and Systems, Vol. LNCS-8461)*, Erika Ábrahám and Catuscia Palamidessi (Eds.). Springer, Berlin, Germany, 33–48. https://doi.org/10.1007/978-3-662-43613-4_3 Part 1: Specification Languages and Type Systems.

A Appendix

A.1 Test cases

The following table shows the test cases that was used for verifying that the implementation of the system is correct by standards of the original design by Shapiro et al. [8] Test cases were done with a cluster of three nodes.

Testcase	Operation	Node(s)	Output
1: test addVertex and synchronize	addVertex v1	1	TRUE
	lookupVertex v1	1, 2, 3	3x True
2: test hidden arcs	addVertex v1	1	TRUE
	AddArc (v1, v2)	1	TRUE
	LookupArc (v1, v2)	1	FALSE
	addVertex v2	1	TRUE
	LookupArc (v1, v2)	1	TRUE
3: test removeVertex	addVertex v1	1	TRUE
	– Time Delay –		
	removeVertex v1	2	TRUE
	– Time Delay –		
	lookupVertex v1	1	FALSE
	lookupVertex v1	3	FALSE
4: Shows cascade delete	addVertex v1	1	TRUE
	addVertex v2	1	TRUE
	addArc (v1, v2)	1	TRUE
	removeVertex v1	1	TRUE
	addVertex v1	1	TRUE
	lookupArc (v1, v2)	1	FALSE
5: removeVertex with two-way arcs	addVertex v1	1	TRUE
	addVertex v2	1	TRUE
	addArc (v1, v2)	1	TRUE
	addArc (v2, v1)	1	TRUE
	removeVertex v1	1	TRUE
	LookupArc (v1,v2)	1	FALSE
	LookupArc (v2, v1)	1	FALSE
6: concurrent add and remove	addVertex v1	1	TRUE
	– Time Delay –		
	addVertex v1	2	TRUE
	removeVertex v1	1	TRUE
	– Time Delay –		
	lookupVertex v1	1, 2, 3	3x True
7: test addArc and synchronization	addVertex v1	1	TRUE
	addVertex v2	1	TRUE
	– Time Delay –		
	addArc (v1, v2)	2	TRUE
	– Time Delay –		
	addArc (v1, v2)	3	TRUE

Testcase	Operation	Node(s)	Output
8: test removeVertex when it's not added	addVertex v1	1	TRUE
	− Time Delay −		
	removeVertex v1	2	TRUE
	− Time Delay −		
	removeVertex v1	3	FALSE
9: removeVertex cancels addArc	addVertex v1	1	TRUE
	addVertex v2	1	TRUE
	− Time Delay −		
	addArc (v1, v2)	1	TRUE
	removeVertex v1	2	TRUE
	− Time Delay −		
	lookupVertex v1	1	FALSE
10: Shows remove commune	addVertex v1	1	TRUE
	addVertex v1	2	TRUE
	− Time Delay −		
	removeVertex v1	1	TRUE
	removeVertex v1	2	TRUE
	− Time Delay −		
	lookupVertex v1	3	FALSE
11: tests addVertex	addVertex v1	1	TRUE
	− Time Delay −		
	addVertex v1	2	TRUE
12: tests addArc after it's synchronized	addVertex v1	1	TRUE
	addVertex v2	1	TRUE
	addArc (v1, v2)	1	TRUE
	− Time Delay −		
	addArc (v1, v2)	2	TRUE
13: test removeVertex when none is added	removeVertex v1	1	FALSE
	addVertex v1	2	TRUE
	− Time Delay −		
	removeVertex v1	1	TRUE
14: test removeArc when none is added	addVertex v1	1	TRUE
	addVertex v2	1	TRUE
	− Time Delay −		
	removeArc (v1, v2)	2	FALSE
	− Time Delay −		
	addArc (v1, v2)	3	TRUE
	− Time Delay −		
	removeArc (v1, v2)	1	TRUE
	− Time Delay −		
	removeArc (v1, v2)	2	FALSE