Superlinear and Bandwidth Friendly Geo-replication for Store-And-Forward Systems

Daniel Brahneborg[1][a], Wasif Afzal[2][b], Adnan Čaušević[2][c] and Mats Björkman[2][d]
[1] *Infoflex Connect AB, Stockholm, Sweden*
[2] *Mälardalen University, Västerås, Sweden*
{*first.last*}@*infoflexconnect.se*, {*first.last*}@*mdh.se*

Keywords: Store-and-forward, Replication, SMS.

Abstract: To keep internet based services available despite inevitable local internet and power outages, their data must be replicated to one or more other sites. For most systems using the store-and-forward architecture, data loss can also be prevented by using end-to-end acknowledgements. So far we have not found any sufficiently good solutions for replication of data in store-and-forward systems without acknowledgements and with geographically separated system nodes. We therefore designed a new replication protocol, which could take advantage of the lack of a global order between the messages and the acceptance of a slightly higher risk for duplicated deliveries than existing protocols. We tested a proof-of-concept implementation of the protocol for throughput and latency in a controlled experiment using 7 nodes in 4 geographically separated areas, and observed the throughput increasing superlinearly with the number of nodes up to almost 3500 messages per second. It is also, to the best of our knowledge, the first replication protocol with a bandwidth usage that scales according to the number of nodes allowed to fail and not the total number of nodes in the system.

1 INTRODUCTION

Many services are accessed over the internet, making them vulnerable to "inevitable, frequent, opaque, expensive, and poorly understood" internet outages [Aceto et al., 2018]. Outages caused by network failures are only opaque for the users when the packets can be routed another way or to a different destination. If the service runs on a single server or even in a single data-center, from the perspective of the clients accessing it, it is therefore not a matter of "*if*, but rather *when*" it will fail [Yousif, 2018].

Internet outages caused by local[1] non-deliberate reasons, e.g., power blackouts, fires, and earthquakes, may affect an area of up to 1000 square km with a time-to-recover in the order of hours or days [Cetinkaya and Sterbenz, 2013]. For a service to remain accessible during such an outage, it must therefore run on multiple servers in independent and geographically separated data centers, possibly even on different continents. However, replicating data between such servers can be problematic due to limited network bandwidth and long round-trip latencies ("digital information travels along fiber optic cables at almost exactly $2/3$ the speed of light in vacuum [...] the mnemonically very convenient value of 1 ms RTT [round-trip time] per 100 km of cable" [Percacci and Vespignani, 2003]).

Much of the internet traffic today is to web services, with clients accessing servers assumed to be consistent between each other. However, the CAP conjecture by Brewer [Brewer, 2000], later proven by Gilbert and Lynch [Gilbert and Lynch, 2004], states that the expected behaviour for a web service prevents it from being both Consistent, Available and Partition tolerant at the same time. If partition tolerance is a given, the system can, for instance, either sacrifice consistency by allowing read operations without first verifying the data with the other nodes, or sacrifice availability by disallowing read operations entirely. In either case, the response times from the system can be reduced, and throughput increased, by using only a subset of the system nodes [Thomas, 1979], [Maekawa, 1985], [Kumar, 1991] (as opposed to all of them [Ricart and Agrawala, 1981]) to manage

[a] https://orcid.org/0000-0003-4606-5144
[b] https://orcid.org/0000-0003-0611-2655
[c] https://orcid.org/0000-0001-8009-9052
[d] https://orcid.org/0000-0002-2419-2735

[1] There is obviously not a lot that can be done about outages on a global scale.

consistency.

To ensure data consistency in a replication protocol, the number of nodes to which read operations are replicated, denoted as R, plus the number of nodes to which write operations are replicated, denoted as W, must always be strictly larger than n [Ahamad and Ammar, 1989]. For example, we can set $R = W = \lceil (n+1)/2 \rceil$ for all operations. This gives us $f = \lfloor (n-1)/2 \rfloor$ as the number of nodes allowed to fail before there is a risk for data loss or inconsistencies. To optimize performance of read operations, a common replication method, used by e.g. Redis[2] and Spread[3], is to simply broadcast all updates to all nodes in the system, either directly or via an elected master. This corresponds to the case where $W = n$ and $R = 1$, which obviously satisfies $R + W > n$.

Broadcasting the operations allows the client to use any one of the n nodes to perform the required operations, which is critical for functionalities such as shopping carts. An item added to a shopping cart in a request to one web server node, should still be there when another request, which is routed to another web server node, adds a second item to the cart. However, as n increases, the required bandwidth for the data replication using existing methods also increases, in the order of kn per node.

Despite their popularity, we will not consider web servers in this work. Instead we will focus on the requirements for a replication protocol as it would be used by a store-and-forward system, a software architecture which provides a buffer between producers and consumers of data [Eugster et al., 2003]. This architecture decouples producers and consumers in time, thereby allowing them to work at different paces. It also gives the possibility to dynamically add and remove consumers in response to varying loads from the producers.

Store-and-forward systems have an important trait as compared to a more general data storage: there are no external readers. Once a data tuple has been received and stored by the system, it is up to the system itself to select which tuple is going to be forwarded next, and by which node. The case of a reader accesses accessing a random node to request the value of a particular data tuple, simply does not manifest itself in these systems.

Ensuring a consistent relative order between the data tuples is now the only remaining reason to broadcast read and write operations to a majority of the nodes. If this ordering requirement can be disregarded, as is the case for data tuples representing independent or commutative operations [Shapiro et al., 2011], we can fundamentally change the replication logic. Under these conditions, a store-and-forward system can freely choose any $f + 1$ nodes for the storage of each data tuple, and any subset of f nodes can still fail without risking data loss.

This work aims to define a data replication protocol specifically tailored for store-and-forward systems handling independent data tuples. The bandwidth required for the replication should be less than the order of kn per node, to avoid using most of the bandwidth replicating the data tuples as opposed to delivering them. Ideally, the replication overhead should be in the order of kf per node. Each data tuple should be forwarded only once, though a minuscule number of duplications are acceptable in exceptional circumstances. We do not specify this requirement as simply "at least once", because that would allow for all data tuples to be forwarded repeatedly which breaks the bandwidth requirement.

The replication protocol we will describe in this work satisfies the aforementioned requirements, and allows replication writes in all network partitions with at least $f + 1$ nodes. We claim the following contributions in relation with this protocol.

1. A high level description of its functionality.
2. An open sourced proof-of-concept implementation of the data replication parts of the protocol.
3. A performance analysis on throughput and latency, both when deployed within a local network and for a geo-distributed system configuration.

Following this introduction is a description of the assumptions we have made about our system model, and a sample application context. Section 2 describes the proposed protocol. Section 3 describes the experiment conducted to evaluate its performance, with the results presented in Section 4 and discussed in Section 5. Finally, Section 6 discusses related work, and Section 7 holds conclusions and possible future work.

1.1 System Model

Our system model comprises a collection of n nodes, named $\text{node}_1, \text{node}_2, \ldots, \text{node}_n$. Each node can exchange data with any other node, and may join and leave the system at any time. The nodes are crash-recovery, so they may also rejoin after crashing. Our model is asynchronous as the nodes may be geographically distant from each other.

In accordance with the store-and-forward architecture, we have a set of producers and consumers, each one connected to a subset of the system nodes. Data tuples are received from producers, stored in a local queue and subsequently forwarded to one of

[2] https://redis.io

[3] http://www.spread.org

the consumers, after which they are removed from the queue. The data tuples are therefore managed by the store-and-forward application for a relatively short time, typically less than 1 second.

The data tuples contain the following fields.

id A globally unique id.

payload
Opaque application specific payload.

owners
An ordered list of $f+1$ unique node identifiers. The first node referenced in this list is the one which originally received this tuple, and the remaining nodes are the data tuple specific failover nodes.

Security concerns such as authentication and encryption are not part of the model. There are also no byzantine failures [Lamport et al., 1982], with nodes sending arbitrarily erroneous data.

1.2 Example Application

One application area matching our system model is application-to-human messaging, e.g. an SMS gateway. Such gateways are used by SMS brokers, connecting clients via internet to mobile network operators. These clients are companies sending authentication codes, meeting reminders and similar information. Using SMS for these messages is convenient, as this technology makes it possible to reach all customers without requiring any additional software on their mobile phones. Figure 1 shows a schematic view of this setup. In this use case, the replication would be done between multiple SMS gateways belonging to the same SMS broker, without affecting the protocols towards the client companies or the operators.

We will use an SMS gateway for the motivation of various assumptions and decisions throughout this paper. For example, n is in this context typically at most 10, and f at most 2. The payload field in the data tuple consists of the sender's and recipient's phone numbers, the message text, and possibly additional other information.

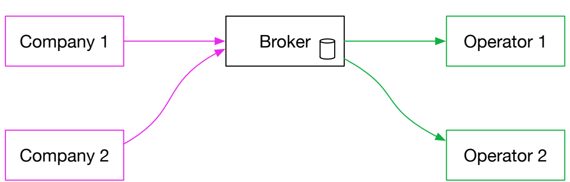

Figure 1: Companies sending text messages, an SMS broker, and mobile network operators.

2 PROPOSED SOLUTION

We now describe our proposed replication protocol, named GeoRep, for use between n nodes, of which f nodes can fail without data being lost. A program, ExampleApp, is running on each node, using a context independent subsystem implementing the replication protocol.

The main data flows for the case with two nodes are shown in Figure 2. A producer, of which there may be many, sends data to ExampleApp on one of the nodes. ExampleApp then tells GeoRep to store the data in its persistent storage, and replicate it to the other node. When ExampleApp has forwarded the data to a consumer, it tells GeoRep to delete the data on all nodes. The GeoRep modules communicate with each other for replication and failure detection. When a failed node has been observed, GeoRep tells ExampleApp to forward the data tuples adopted from the failed node.

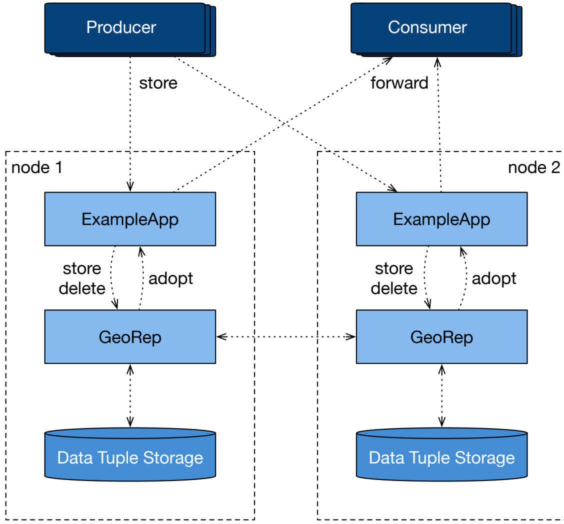

Figure 2: Architecture overview for ExampleApp running on two nodes.

2.1 Protocol Description

We here describe the activities done when GeoRep starts and stops, how data is replicated, and how node failures are handled. All use cases in this section are described from the viewpoint of an individual node.

2.1.1 Startup

At startup, the application layer in ExampleApp provides the selected value for f to its GeoRep subsystem, and an initial list of other nodes. GeoRep first

loads any previously stored data tuples into appropriate data structures in memory. It then waits for contact requests, while also trying to make contact with the other nodes.

In response to a contact request from $node_x$, GeoRep returns a welcoming message with its list of currently known nodes. This list includes temporarily stopped nodes and their expected return times (see Section 2.1.3, "Failover", below). The contacted node informs the others about $node_x$. $Node_x$ tries to connect to the existing nodes, getting their respective lists of known nodes. If any node gets an update during this phase, the full list is broadcast to all other nodes. If the expected return times for a specific node differs, the maximum value is used. Eventually, this will converge, from which point all nodes send periodic heartbeats to all other nodes unless other data has recently been sent.

If a node returns after a short time, each welcoming message will also contain the list of entries adopted by each node. These entries can then be removed by the returning node to avoid duplications.

2.1.2 Replication

The application layer in ExampleApp requests an entry to be replicated. GeoRep creates a list of f other nodes known to be alive out of the existing $n-1$ ones, putting it in the owners field of the entry. If $f > n-1$, this fails immediately.

The entry, plus the owners field, is replicated to the f nodes. Once all those nodes have responded, control returns to the application. If multiple threads request entries to be replicated at the same time, these are all sent as a single network packet. When receiving an entry from another node, it is stored locally and a response sent back, but no other action is taken. Figure 3 shows the replication when $n = 5$ and $f = 2$.

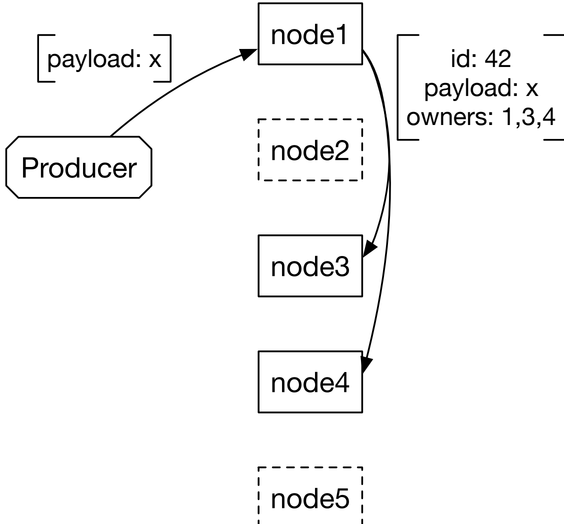

Figure 3: Replicate a payload to a subset of size 2 of the 5 known nodes, here nodes 3 and 4. Nothing is sent to nodes 2 and 5.

2.1.3 Failover

If nothing is received on $node_1$ from $node_2$ for some time, $node_1$ suspects $node_2$ to be dead. After this, no more entries are replicated to $node_2$ until $node_2$ sends something to $node_1$ again.

The reason for this lost connection may be a network outage, leaving multiple isolated subsets of the original n nodes still in contact with each other. Each network partition with such a subset of nodes can continue to run as before, as long as their set size is greater than f.

After some configurable time, or after the recovery timeout given by $node_2$ when it exited, $node_2$ is considered dead. If $node_1$ ends up as the first node in the owners list for one or more entries, the application is notified, one entry at a time. The identifiers of the adopted and successfully sent entries are stored for a limited time, making it possible to notify $node_2$ should it return.

As $node_1$ knows the identifiers of the rest of the nodes to which each entry was replicated, it will try to inform those nodes about updated statuses. Only the nodes in the owners list will ever send updates and deletes for a particular entry.

2.1.4 Exiting

When ExampleApp exits and tells GeoRep to shut down, this is broadcast to all other nodes, including a timeout for when the node expects to be back. This timeout is also stored locally. The timeout tells the other nodes when they can start adopting that node's messages. If the original node comes back after the timeout has expired, it can assume all of its messages have been adopted by the other nodes.

2.2 Duplication Analysis

We now consider the cases that can occur in the same situation as in Section 2.1.2, when $n = 5$ and $f = 2$, and a message is replicated from $node_1$ to $node_3$ and $node_4$. The cases are shown in Figure 4. Neither $node_2$ nor $node_5$ have ever heard of this message, so whether they remain in contact with the other nodes does not matter.

A. As long as $node_1$ is alive, it will try to deliver the message to the consumer, and the statuses of the

other nodes do not matter.

B. If $node_3$ concludes that $node_1$ is dead or for some other reason unreachable, it will adopt the message and try to deliver it.

C. If $node_4$ loses contact with both $node_1$ and $node_3$, it will then try to deliver the message itself.

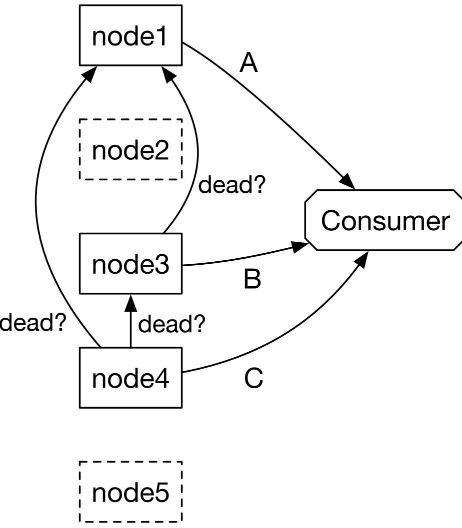

Figure 4: Possible duplications.

There is no way for a node to know if any of the other nodes are dead or are unreachable for another reason. In case multiple nodes can communicate with the consumer but not with each other, messages could therefore be duplicated. The probability for this is low, and these duplications are therefore acceptable. We consider it much more likely that a lost node is dead or has lost internet connectivity entirely, and thereby also the connectivity to the consumer. In both cases the message is delivered only once.

2.3 Data Tuple Lifecycle

Figures 5 and 6 illustrate the replication and failover from the perspective of a data tuple. The `Inactive` state has a dashed border to show that it is a passive state, waiting on an externally initiated event. The solid arrows arrows represent state changes on the first node, and dotted arrows on the failover nodes.

First, in Figure 5, a producer sends the data tuple to some node, whereby the data tuple enters the `Received` state. This corresponds to the arrow from `Producer` to `node1` in Figure 3. Next, this node sets the owners field, and replicates the updated data tuple to the selected failover nodes, where they are stored in the `Inactive` state. In Figure 3, these are the arrows on the right, from `node1` to `node3` and `node4`. When

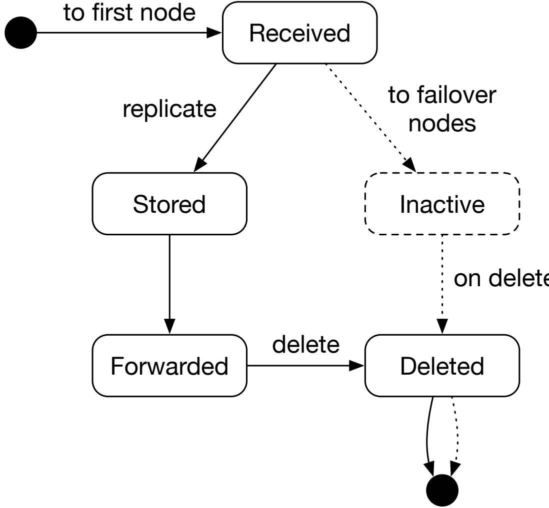

Figure 5: The lifecycle of each data tuple on the first node.

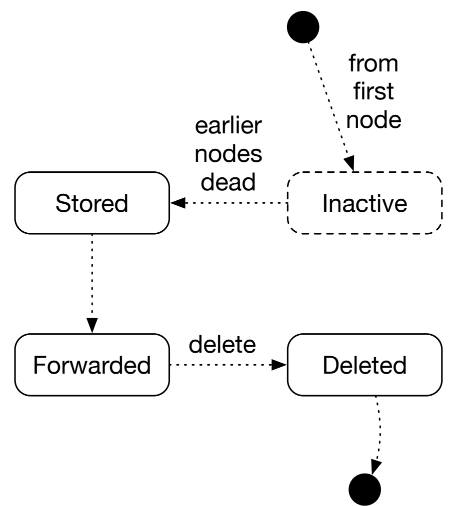

Figure 6: The lifecycle of a data tuple in case of failover.

the failover nodes have confirmed this operation, the data tuple on `node1` moves to state `Stored`. It stays in this state until the application has forwarded the data.

In the normal case, the application will forward any data tuple in the `Stored` state, and then move them to the `Forwarded` state. This instructs GeoRep to inform the failover nodes, i.e., `node3` and `node4` in Figure 3, that this data should be deleted. Finally, the data tuple is removed from the local storage in GeoRep on the first node as well.

Figure 6 illustrates the cases shown as B and C in Figure 4, when a failover node discovers that all earlier nodes in the owners field no longer respond to its heartbeat requests. It then moves the data tuple from state `Inactive` to `Stored`, and informs the applica-

tion about this change. The lifecycle then proceeds as above, causing the data tuple to be forwarded and then deleted on any remaining failover nodes. As described in Section 2.2, there is a possibility for the same data tuple to enter the `Stored` state and therefore be forwarded by multiple nodes. We do not need to create a mechanism to prevent that, as such duplication are acceptable according to our requirements.

2.4 Source Code

The source code, consisting of about 3500 lines of C, is publicly available[4]. This includes both the proof-of-concept implementation of the replication protocol and the test application and scripts used in the experiment described next.

3 EXPERIMENT

For a study of the proposed protocol, we designed a controlled experiment [Robson and McCartan, 2016]. The goal was to evaluate the throughput with a varying number of clients and servers in a few different configurations.

3.1 Tasks

A common benchmark suite for replication systems is the Yahoo! Cloud Serving Benchmark (YCSB) [Cooper et al., 2010], which exists in several different versions with varying proportions between writes and reads. Using this suite makes it easy to compare different solutions, but as it is designed for web server type systems and not store-and-forward systems, YCSB was not meaningful for us. Instead, we used a sequence of tasks corresponding with the queue related operations performed by this type of systems. The test application created the messages itself, and discarded them when all tasks described below were completed.

1. A new message was stored. The application waited for acknowledgements from the required number of servers.
2. A message was extracted from the queue.
3. The extracted message was deleted.

3.2 Factors

Due to the nature of the experiment, we were able to use a fixed design [Wohlin et al., 2012]. In addition to the usual Independent and Dependent factors, we found it relevant to describe the independent factors that we set to constant values, and the dependent factors which we chose to ignore. These are all described in more detail below, and summarized in Table 1. The throughput is measured in messages per second (MPS).

Table 1: Experiment factors

Type	Factor	Value(s)/Unit
Independent	Servers, n	1, 3, 5, 7
	Clients	1, 3, 10, 30, 100
	Separation	Local, Remote
Constant	Protection, f	1
	Transient	5 s
	Steady-state	30 s
Dependent	Throughput	MPS
	Min RTT	Microseconds, μs
Ignored	Recovering	MPS
	Duplications	Ratio

3.2.1 Independent Factors

The primary factors in this experiment were selected to give a deeper understanding of the behaviour in different circumstances.

The number of servers was varied from 1 to 7. Testing a replication method on a system with a single server may seem strange, but this gave a baseline so we could observe the cost of the increased data safety provided by the replication.

Next, the number of client connections was varied with a ratio of about 3 from 1 to 100. A real system would have more clients than 100, but it would also run on machines with more resources than we had available for this experiment.

Finally, we used servers both within the same data center and in multiple time-zones. The physical distances between the servers showed the effect of high round-trip times.

3.2.2 Constants

The number of nodes that would be allowed to fail, f, was set to 1 for all tests. A higher value for f would mainly lead to more network traffic and possibly longer round-trip times as seen by the clients. Investigating the exact effects of this parameter was not considered important at this time.

All configurations were tested for 35 seconds. First, there was a transient phase of 5 seconds, allowing the CPU caches and TCP parameters to stabilize. Next, the code continued to run in the steady-state phase for another 30 seconds.

[4] https://bitbucket.org/infoflexconnect/leaderlessreplication

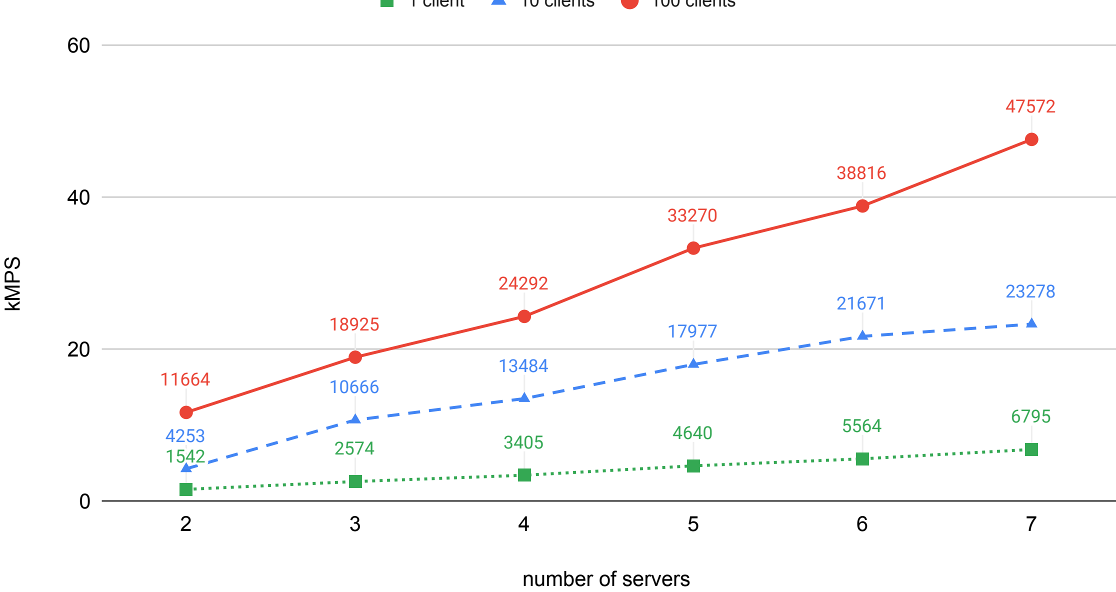

Figure 7: System throughput as a function of the number of servers, all running in the same data center.

3.2.3 Dependent/response Variables

For all configurations, i.e. the combinations of one particular value for each of the independent variables, the response variable of most interest to us in this experiment was the total system throughput. This throughput was defined as the number of messages processed per second, according to the sequence of tasks described in Section 3.1. We also measured the minimum RTT between each pair of nodes.

3.2.4 Ignored Response Variables

Other response variables that might be of interest mainly concern the behaviour when a failed server is detected, and the time-span afterwards during which the system is reassigning messages to new servers. Just as in the evaluation of MDC-Cast [Nehme et al., 2018], another replication protocol for geo-separated environments, we focus on the performance of the failure-free case.

3.3 Procedure

Before each test, all servers were reset to a known empty starting state. The files for local storage were removed, so they could be recreated without content. The application was then started on all servers, with the selected values for the independent variables given to it as command line parameters.

The test application counted the number of messages processed each second by each server, values that were then summarized into a result for the full system. The median of the values for each of the 30 seconds in the steady-state phase was then calculated.

3.4 Execution

We used a total of thirteen servers during January 2020, all of them being the smallest ones offered by DigitalOcean[5] at that time: 1 GB memory, 25 GB disk, and 1 virtual x64 CPU. They all ran CentOS 7.5, with the working directory on the filesystem XFS. The code was compiled using gcc 4.8.5.

4 RESULTS

Here we present a summary of the results. The full list of measurements for all tested configurations is available together with the source code.

We got distinct results for three cases: a single server, a cluster in a local network, and a cluster in a geo-separated network. For a single server, which effectively just means the key-value store LevelDB, we got a peak throughput of 87 588 MPS.

In a local network, the total system throughput increased with the number of nodes up to 47 572 MPS

[5] http://digitalocean.com

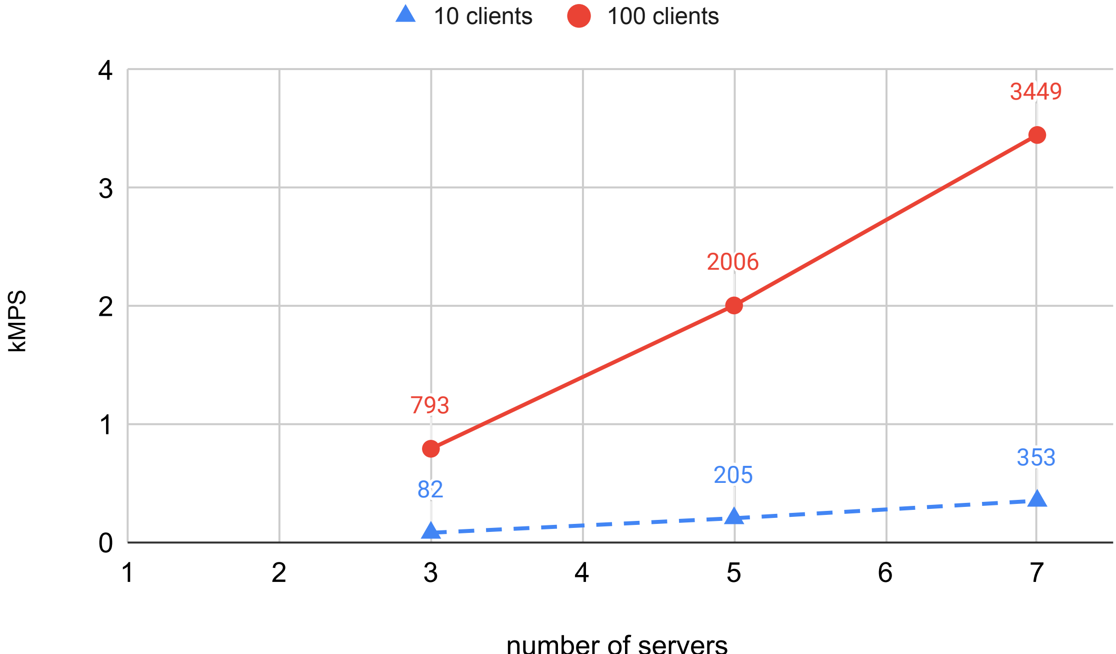

Figure 8: System throughput as a function of the number of servers, running in different data centers on multiple continents.

on 7 nodes, shown in Figure 7. The minimum RTT varied between 143 μs and 420 μs.

When GeoRep was deployed in a cluster with servers in separate time-zones, throughput again increased with the number of nodes. The peak throughput levels were obviously lower than in the local case due to the longer round-trip times. Figure 8 shows how the throughput reached 793 MPS for 3 nodes and 3449 MPS for 7 nodes. The throughput also increased almost perfectly with the number of clients, which we can see as on 7 nodes 10 clients reached 353 MPS, and 100 clients reached, as mentioned, 3449 MPS.

For 3 geo-separated nodes, the minimum RTT averaged 174 ms. For 7 nodes, some pairs of nodes ended up relatively close to each other (e.g. New York and Toronto in North America, and Amsterdam, Frankfurt, and London in Europe), resulting in the average minimum RTT decreasing to 108 ms. Figure 9 shows the RTT between a few selected pairs of nodes. The profiles for nodes geographically close to each other are notably similar.

5 DISCUSSION

GeoRep was able to leverage the ordering independence and performed better as the number of clients, and thereby also the number of parallel requests, increased.

With a larger number of nodes, n, the average distance between each pair of nodes in the geo-distributed case and thereby their pair-wise round-trip times, decreased. The system as a whole thus became superlinear, providing throughput which increased faster than the number of nodes, from 264 MPS per node for 3 nodes, to 493 MPS per node for 7 nodes, as shown in Figure 10.

With more powerful servers it may be possible to support a larger number of clients, getting an even higher system throughput. We also expect the throughput to increase further by using a larger number of nodes than 7.

Next, we will discuss the various types of identified validity threats.

5.1 Threats to Validity

The identified validity threats are grouped [Cook and Campbell, 1979, Jedlitschka et al., 2008] for better overview.

5.1.1 Internal

Internal validity threats concern the causal relationship between two variables. Even though an exist-

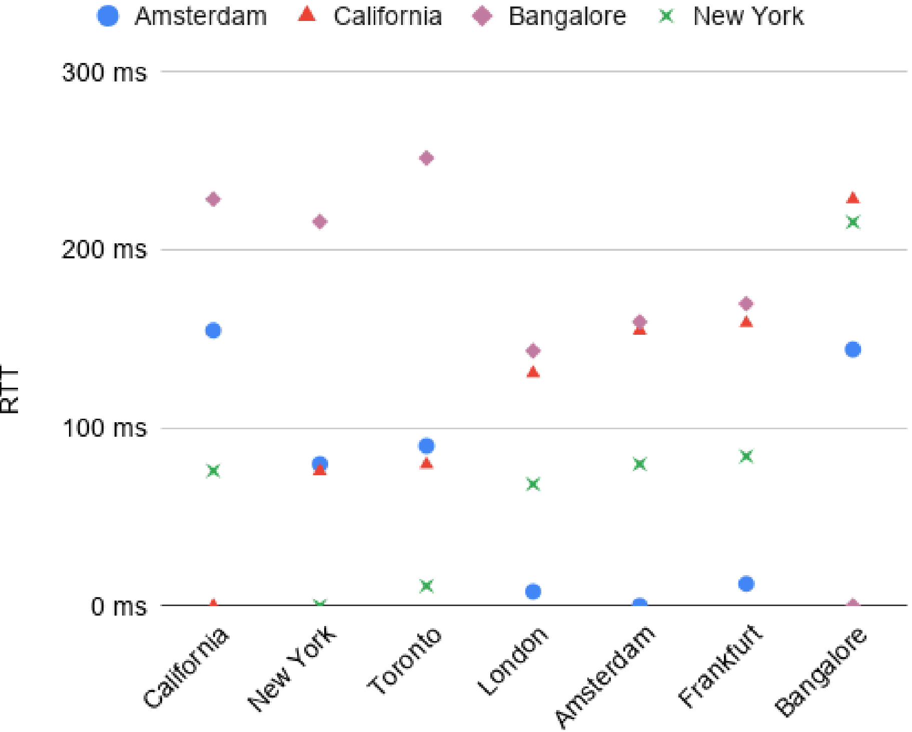

Figure 9: Round-trip time (RTT) for various pairs of servers.

ing SMS gateway was the driving force for the requirements addressed by GeoRep, a new and minimal application was written for these experiments. This avoided the threat of any confounding variables introduced by the gateway implementation and simplified the reproducibility.

To address the threat of additional confounding factors, all cases were run for a relatively long time. As we focused on the median, any temporary variances in the environment were effectively filtered out.

5.1.2 External

External validity threats concern whether the results are still valid in a more general context. Due to not having a coordinating server, our proposal is only usable for situations where the stored elements have no relative order.

6 RELATED WORK

Other store-and-forward systems are application-to-application queues, e.g. Apache Kafka [Kreps et al., 2011]. In Apache Kafka the data in the system can be spread over multiple subsets of the nodes, with each such subset being called a partition. A partition has an elected leader, which handles all reads and writes, and zero or more replicas which are kept in sync using a very efficient mechanism. Should the leader become unavailable, one of the replicas takes its place. This gives an automatic ordering of the events, but at the cost of being sensitive to the network latency between the client and the replica leader. GeoRep avoids this cost, as it has no leader. Instead, clients are free to connect to any node of their choice, thereby minimizing the latency time and as a result maybe also maximizing the throughput.

For systems where a global ordering must be maintained, the replication protocols are often based on a variant of Paxos [Lamport, 1998] or Raft [Ongaro and Ousterhout, 2014]. The Paxos variant Mencius [Mao et al., 2008] was designed to perform well even in wide-area networks with high inter-node latency. One of the ways they achieve this is to use a multi-master setup, where the leadership is divided among all nodes similarly to GeoRep. However, as all data is sent to all other nodes, the throughput does not increase when nodes are added to the system.

Many of the data replication protocols are based

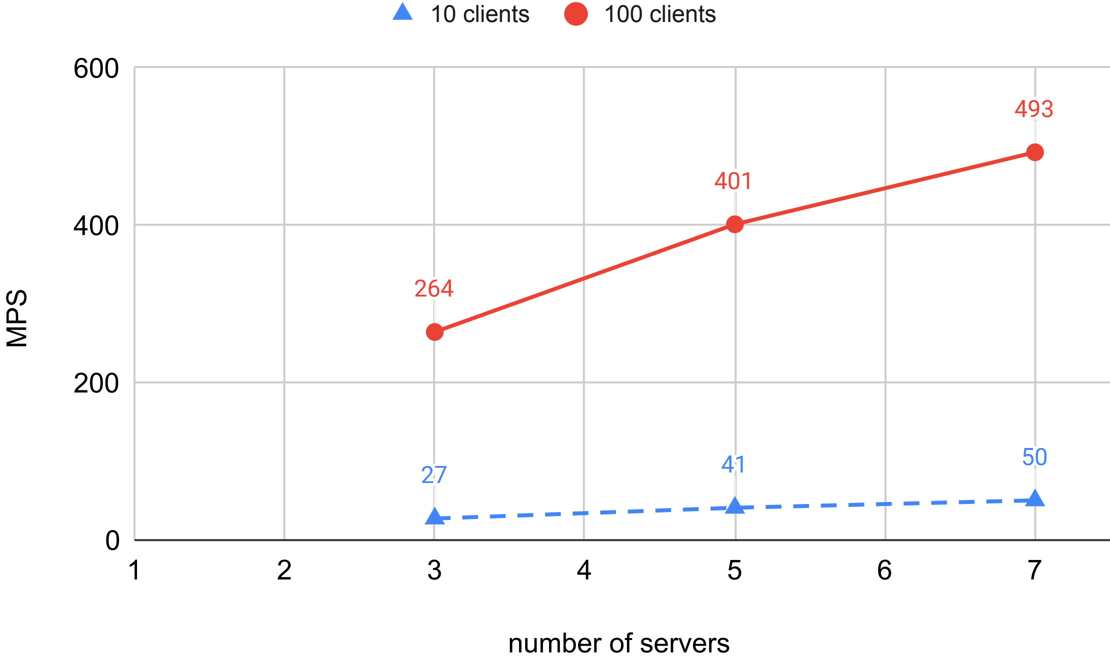

Figure 10: Throughput per node in the geo-distributed case.

on total order broadcast [Défago et al., 2004], which also requires all data to be processed by all nodes in the same order. This makes the situation easier for the upper layers of the protocol, but can never lower the bandwidth requirements. When the data payload is big enough to make the system network bound, throughput therefore instead notably decreases. Geo-Rep could in this situation provide a system throughput which increased faster than n, as shown in Figure 8 and Figure 10. To the best of our knowledge, the only other replication system achieving this is PaRiS [Spirovska et al., 2019].

7 CONCLUSIONS AND FUTURE WORK

To provide data safety against internet and power outages for store-and-forward systems, we created a new replication protocol. Particular consideration was given to the requirements for SMS traffic, e.g., having to work well in a geo-distributed configuration and a lack of global order. In a controlled experiment, the protocol was shown to scale well both with the number of nodes in the system and the number of parallel message producers. The highest recorded throughput for the geo-distributed case was close to 3500 MPS, but higher values ought to be possible using a larger number of more powerful machines. The source code used in the experiment is open sourced to enable replication studies.

For future work, we recall Figure 9 showing the round-trip times between some selected pairs of nodes. Instead of replicating messages to a random selection of nodes, we can select the f ones with the smallest RTT from where the message was received, ignoring nodes with an RTT lower than 10 ms. This minimum value ensures messages are always replicated outside of the critical region mentioned in the Introduction section.

ACKNOWLEDGEMENTS

This work was sponsored by The Knowledge Foundation industrial PhD school ITS ESS-H, H2020 project ADEPTNESS (871319) and Infoflex Connect AB.

REFERENCES

Aceto, G., Botta, A., Marchetta, P., Persico, V., and Pescapé, A. (2018). A comprehensive survey on internet outages. *Journal of Network and Computer Applications*, 113(2018):36–63.

Ahamad, M. and Ammar, M. (1989). Performance Characterization of Quorum-Consensus Algorithms for Replicated Data. *IEEE Transactions on Software Engineering*, 15(4):492–496.

Brewer, E. A. (2000). Towards Robust Distributed Systems. In *Principles Of Distributed Computing*. ACM.

Cetinkaya, E. K. and Sterbenz, J. P. (2013). A taxonomy of network challenges. In *International Conference on the Design of Reliable Communication Networks*, DRCN.

Cook, T. D. and Campbell, D. T. (1979). *Quasi-experimentation: Design and analysis for field settings*, volume 3. Rand McNally, Chicago.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM symposium on Cloud computing*, SoCC '10, New York, NY, USA. ACM.

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms. *ACM Computing Surveys*, 36(4):372–421.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131.

Gilbert, S. and Lynch, N. A. (2004). Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. In *Principles Of Distributed Computing*, PODC.

Jedlitschka, A., Ciolkowski, M., and Pfahl, D. (2008). Reporting experiments in software engineering. In *Guide to advanced empirical software engineering*, pages 201–228. Springer.

Kreps, J., Narkhede, N., and Rao, J. (2011). Kafka: a Distributed Messaging System for Log Processing. In *Proceedings of the SIGMOD Workshop on Networking Meets Databases*, NetDB, Athens, Greece.

Kumar, A. (1991). Hierarchical Quorum Consensus: A New Algorithm for Managing Replicated Data. *IEEE Transactions on Computers*, 40(9):996–1004.

Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169.

Lamport, L., Shostak, R., and Pease, M. (1982). The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401.

Maekawa, M. (1985). A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems*, 3(2):145–159.

Mao, Y., Junqueira, F. P., and Marzullo, K. (2008). Mencius: Building Efficient Replicated State Machines for WANs. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI, Berkeley, CA, USA.

Nehme, M.-J., Palix, N., Beydoun, K., and Quema, V. (2018). MDC-Cast: A Total-Order Broadcast Protocol for Multi-Datacenter Environments. In *IEEE Symposium on Reliable Distributed Systems*, SRDS. IEEE.

Ongaro, D. and Ousterhout, J. K. (2014). In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*.

Percacci, R. and Vespignani, A. (2003). Scale-free behavior of the internet global performance. *The European Physical Journal B*, 32(4):411–414.

Ricart, G. and Agrawala, A. K. (1981). An Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*, 24(1):9–17.

Robson, C. and McCartan, K. (2016). *Real world research*. John Wiley & Sons.

Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. (2011). A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, Inria – Centre Paris-Rocquencourt.

Spirovska, K., Didona, D., and Zwaenepoel, W. (2019). PaRiS: Causally Consistent Transactions with Non-blocking Reads and Partial Replication. In *IEEE International Conference on Distributed Computing Systems*, ICDCS. IEEE.

Thomas, R. H. (1979). A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Yousif, M. (2018). Cloud Computing Reliability—Failure is an Option. *IEEE Cloud Computing*, 5(3):4–5.