A Resilient Message Queue

Daniel Brahneborg, Romaric Duvignau
Wasif Afzal, Saad Mubeen

June 21, 2021

Abstract

In order to keep internet based services available despite inevitable local internet and power outages, their data must be replicated to one or more other locations. Incidentally, such replication also protects against data loss. However, for store-and-forward systems with geographically separated system nodes, we have not found one with sufficiently efficient network usage. The purpose of this work is to propose a novel solution for replicating data in such systems. Leveraging our application-specific semantics, such as a lack of relative order between replicated data tuples, we designed a new replication protocol. We verified the failover mechanism of our proof-of-concept implementation using simulated network failures, and evaluated it on throughput and latency in several controlled experiments using up to 7 nodes in 5 geographically separated areas. When we increased the number of nodes from 2 to 7, the system throughput increased as well, up to 24 562 messages per second (MPS). When the protocol was configured to replicate data to nodes as nearby as possible, such as between New York and Toronto, instead of always selecting random nodes, the total system throughput reached 40 149 MPS for the same set of servers. This is, to the best of our knowledge, the first replication protocol with a total network usage that scales according to the number of nodes allowed to fail and not to the total number of nodes in the system.

Contents

1 Introduction

All over the world, various types of disasters happen with both regular and irregular intervals [13,32,44,45]. These disasters, which could be both natural, technical and political, affect both network and power equipment, and therefore might lead to outages for internet services [1, 7, 60]. Such outages can sometimes be mitigated by using multiple geographically separated servers [13, 51, 52, 58], to avoid single points of failure. The servers exchange data with each other as necessary, allowing clients to connect to any one of them. However, as the communication is limited by the speed of light, we get round-trip times between the servers in the order of 1 ms per 100 km [47]. A reasonable minimal distance between servers to keep disasters from causing more than one of them to fail might be in the order of 1000 km, limiting the number of round-trips within the system to one hundred per second at best.

Maintaining the same data on multiple servers is not a new problem, and a common solution is to send all information regarding the processed data to all of them [11]. Let us here call this strategy "AllToAll"[1]. AllToAll is often managed via a master server as in Paxos [30, 39] or Raft [46], ensuring both that all data and its operations are communicated to all servers, and that the operations are processed in the same order everywhere [5].

The AllToAll strategy is easy to understand and reason about, and is implemented in various concrete tools and libraries, e.g., Redis[2] and Spread[3]. It forms the basis for eventual consistency [56], and for Convergent and Commutative Replicated Data Types (CRDTs) [53]. It is good for web applications and other request-response based systems as it gives good availability for external readers, which can send the requests to any one of the included servers and get reasonably current data in return. It also makes resilience, as described by the ResiliNets project [44] at `resilinets.org` , straightforward, as the system in case of a failed server can freely select one or more of the remaining servers to take over its work [35].

However, AllToAll has a number of shortcomings. It wastes network traffic [31,32], as the amount of data grows quadratically by the number of servers in the system. AllToAll requires all servers to be able to reach each other, possibly going via one or more other servers. When there is a network partition, in other words any type of failure causing this reachability to no longer be true, AllToAll breaks [12,23,52], which in turn reduces the system availability [4,16,20,26,52]. Considering the stability and lifetime of typical hardware, this level of redundancy is mostly unnecessary anyway [3]. We also know that the required coordination is costly [29,49], limiting the system performance. Sharding is a technique for storing each data tuple on a predefined subset of the nodes. This indeed decreases the network traffic, but it also lowers the availability, as it limits the number of nodes where each particular data tuple can be stored.

Among others, Helland and Campbell in 2009 [27] and Hellerstein and Alvaro in 2019 [28], argued for shifting the focus from storage to application semantics. For a specific application, there may exist solutions to the problem of geo-distributed re-

[1] Please note that AllToAll concerns the behavior on the application level. This may or may not use the lower level construct *Total Order Broadcast* [17].

[2] `https://redis.io`

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

silience which are more effective than AllToAll, and can therefore provide higher performance.

In this work, we consider an application providing a message queue for mobile text messages (SMS). The messages are added to the queue by senders, normally companies sending information to their customers, and are then pushed by the queue itself to the network operators for final delivery to the mobile phones. After being forwarded to the operators, the messages are removed from the queue. Due to the queue's push construct, there is no external reader pulling messages from it, and therefore we do not need all nodes to receive the same set of data and operations on that data. The messages are short lived within the queue, and could even be removed from the queue before they have had time to be replicated everywhere according to the AllToAll strategy. As each SMS is independent,[4] we do not need the same operation ordering on all nodes. Therefore, we need no mechanism for enforcing this order [55], and are even free to store different subsets of the queue on each node [52]. Instead of the "all nodes are equal"-consistency provided by AllToAll, we only need the system to be confluent, as expressed by Alvaro [6]: "same set of outputs for all orderings of its inputs", specifically from the perspective of each recipient of the text messages. This strategy goes back at least to an idea by Alsberg [5] in 1976: "Non-interaction may also occur with writers if the data modified by each is disjoint".

Using the scalability equation (1) by Gunther *et al.* [24], where n is the number of nodes in the system, this intentional inconsistency between the nodes would remove the otherwise necessary saturation, resulting in $\kappa = 0$.

$$S_n = \frac{n}{1 + \sigma n + \kappa n(n-1)} \tag{1}$$

With all the potential possibilities for higher performance mentioned above, the **purpose** of this work is to design a replication protocol for a resilient message queue with high efficiency, allowing disaster-resistant processing of 1000 or more messages per second (MPS) per server.

The resulting design was evaluated using a proof-of-concept implementation, tested on servers on multiple continents. Even on servers with modest performance, we reached about 3500 MPS per node in the geo-diverse case, replicating all data tuples to a random other server in the world. By allowing replication to the nearest server, e.g., between New York and Toronto, we instead got 5735 MPS per node.

We claim the following contributions in relation with this protocol.

1. A high level description of its functionality.
2. An analysis of the reliability in terms of availability, potential data loss, and potential data duplication.
3. A method to verify the failover mechanism.
4. A performance analysis on throughput, both when deployed within a local network and for a geo-distributed system configuration.

[4] Messages to different recipients can of course be delivered in any order. Long messages consisting of multiple parts can even be sent in any order to the recipient's phone, as the parts are tagged with a sequence number and will therefore always be merged together correctly in the phone.

5. An open-sourced implementation.

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. Next follows evaluations of the protocol from three different perspectives. First, Section 3 contains a theoretical analysis of the reliability. Then, Section 4 describes the verification of the failover mechanism, and finally Section 5 describes the setup for the experiments conducted to evaluate its behaviour in a real-world configuration, focusing on the quality attribute throughput. The results are discussed in Section 6, and related work in Section 7. Section 8 holds conclusions and some ideas for future work.

The main differences between the previously published conference paper [10] presenting this protocol and this article is this new Introduction section focusing more on resilience , the extension of the "Duplication Analysis" subsection into a more complete Reliability Analysis in Section 3, the failover verification in Section 4, the added experiment in Section 5.4, and an extended list of references.

1.1 System Model

Our system model is a classic store-and-forward queue [18], with external sets of producers and consumers [19]. Data tuples, described in more detail below, are received from the producers and stored in the queue. As soon as possible after they are received, each data tuple is forwarded by the queue to one of the consumers. The data tuples are therefore managed by the queue for a relatively short period of time, normally less than 1 second.

The ownership and responsibility of each data tuple travels along with the data. This is in contrast to other store-and-forward based communication protocols such as TCP, where the data stays with the original producer until the final consumer has acknowledged its arrival. Our mechanism increases the burden on the queue to avoid data loss, but lowers the amount of data the producer and the queue need to store. It also removes the need for a reliable delivery notification from the consumer. Therefore, when our queue acknowledges having received a data tuple, the producer removes their copy. Likewise, when the consumer has received the data, it can be removed from the queue.

The part of the system we can control and manipulate in this model is just the queue itself, which comprises a collection of n nodes, named node_1, node_2, …, 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 rejoin after crashing.

Nodes which are physically close, such as nodes running in the same data center, may share a single point of failure, becoming unavailable due to the same power failure, failed internet connection, etc. Replicating data to several such nodes therefore provide no extra protection, but still requires additional costs in terms of power consumption, network communication, CPU and disk usage. However, with geographically separated nodes, the probability for some event killing multiple nodes during the processing of a particular data tuple is effectively zero. Due to this, the communication between the queue nodes is asynchronous.

Each producer and consumer is a third party system connected to one or more queue nodes. We assume the producers can maintain a list of addresses to multiple nodes they can use when sending their data tuples. However, we cannot change the communication protocol used with these parties, nor anything else in their system.

In addition to n, the number of nodes in the system, we will use f for the number of nodes which are allowed to fail at the same time *without data being lost*. This value is typically 1 or 2.

Closely related to n and f is what we call "majority replication", which we use for all data replication protocols based on inequality (2) below. AllToAll normally uses $\textit{number of writers} = n$ and $\textit{number of readers} = 1$, which trivially satisfies this condition [2].

$$\textit{number of readers} + \textit{number of writers} > n \tag{2}$$

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 failover nodes for this specific data tuple.

Security concerns such as authentication and encryption are not part of the model. There are also no byzantine failures [40], with nodes sending arbitrarily erroneous data.

1.2 Example Application

One of the application areas 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 neither the client companies nor 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, which is in the same order as in the evaluations of both S-Paxos [9] and PaRiS [54]. 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. The target throughput of 1000 MPS was selected to satisfy most existing SMS brokers and SMS sending companies we have been in contact with.

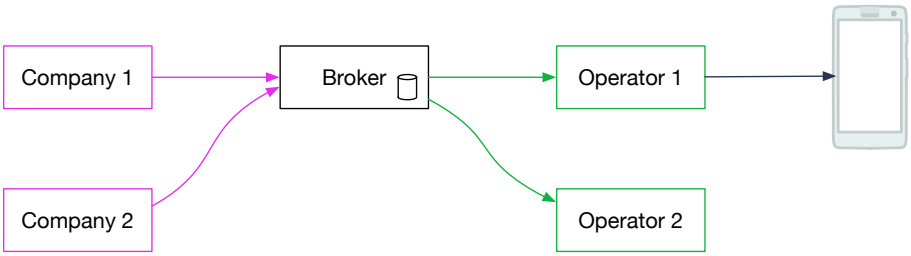

Figure 1: Companies sending text messages, an SMS broker, and mobile network operators.

The network operators implement their own message queues, making the person using the mobile phone the final consumer. This affects the delivery guarantees we must support, as it is important that all messages are delivered as soon as possible, but it is not a big problem if an occasional message is delivered twice. Similar to the established terms “at most once” and “at least once”, we call this “once plus epsilon” delivery. The term “at least once” allows any number of repetitions of each message, but we want to minimize these.

Some operators even has a small filter, automatically ignoring duplicate messages. They still incur a cost for the sender, again making it important to keep the number of duplications as small as possible.

2 Proposed Solution

In this section we describe our proposed replication protocol, named GeoRep. It is designed to be used on n nodes, of which f nodes can fail without data being lost. A program, named ExampleApp, is running on each node, using a context independent subsystem implementing the replication protocol.

The main data flows for a configuration 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. The producers here correspond to the companies in Figure 1. ExampleApp then tells GeoRep to store the data in its local persistent storage, and replicate it to the other node. When ExampleApp has forwarded the data to a consumer, corresponding to one of the operators in Figure 1, 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. So, ExampleApp does not know anything about replication, and GeoRep knows neither of the producers nor the consumers.

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.

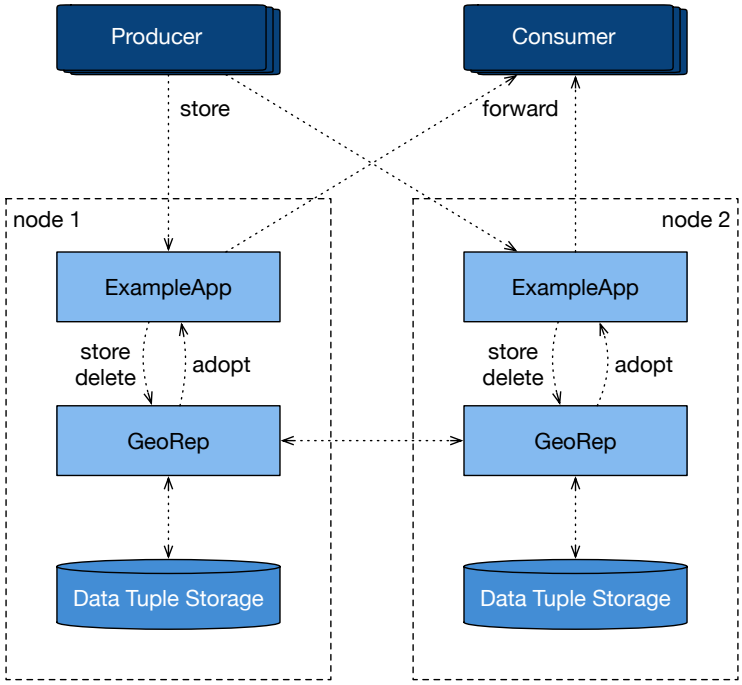

Figure 2: Architecture overview for ExampleApp running on two nodes.

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

In response to a contact request from node_x, GeoRep on the contacted node 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, while 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. Eventually, this will converge, from which point all nodes send periodic heartbeats [5] 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 reduce the number of duplications.

2.1.2 Replication

According to our system model described in Section 1.1, f nodes are allowed to fail without causing data loss. At the very least, all received data tuples must therefore be replicated to f additional nodes before the producer can get the corresponding acknowledgement. However, as there is no requirement of keeping all nodes identical, we do not need to replicate the data to more than f nodes. The replication algorithm therefore becomes as follows.

1. The application layer in ExampleApp requests some opaque data to be replicated.
2. GeoRep creates a list of f other nodes known to be alive out of the other $n-1$ ones it knows about, putting this list in the owners field of the data tuple. If the number of alive nodes is less than f, the operation is terminated immediately, and a failure status is returned to the application. If this happens, the producer is free to select another node to send the data tuple to.
3. The data, plus the owners field, is replicated to the f selected nodes.
4. Once all those nodes have responded, control returns to the application.

If multiple threads request entries to be replicated sufficiently close in time to the same node, 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. In particular, none of the received messages are forwarded at this point. Figure 3 shows the replication when $n = 5$ and $f = 2$, for a message received by node_1, and the f other nodes being node_3 and node_4.

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 [41]. After this, no more entries are replicated from node_1 to node_2 until node_2 sends something to node_1 again.

The reason for this lost connection may be a network outage, resulting in multiple isolated subsets of the original n nodes still in contact with each other. Each network partition with such a subset of at least $f+1$ nodes can continue to run as before.

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 running on node_1 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, and only to the nodes originally stored in that list.

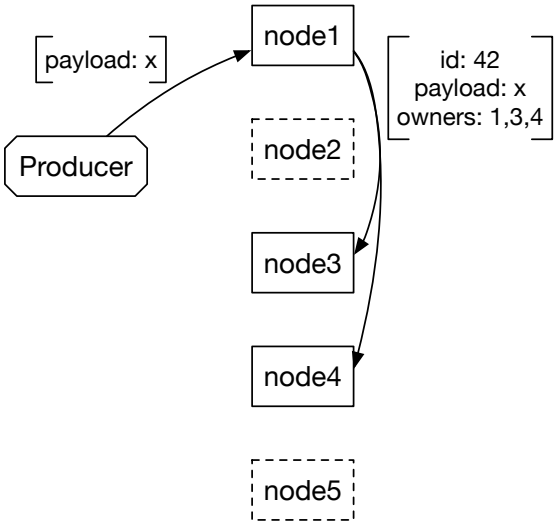

Figure 3: Replicate a payload to a subset of size 2 of the 5 known nodes, here nodes 3 and 4. This payload is sent neither to node 2 nor node 5.

2.1.4 Exiting

When ExampleApp exits and tells GeoRep to shut down, this event 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 Peer Life Cycle

Figure 4 shows the states and transitions used by each node for each one of the other nodes. Each node maintains its own list of states for these peer nodes, so all nodes can take different decisions on which other nodes to replicate data to. This is intentional, and an important feature of this replication protocol as it both avoids having to reach consensus on this, and allows the protocol to continue to work even in case of partial failures. As our model has crash-recovery nodes, there is no end state.

When a node is informed about the existence of a new peer, the new peer starts in the *Prospect* state, causing the node to send it a greeting. When the peer replies with some data, regardless of the current state, it is moved to the *Active* state. This is the only state where it can receive new data tuples, and is marked with **boldface**.

When no data has been received for some time, the peer first moves to the state *Schrödinger*, and after an additional time to the state *Terminated*. The timeouts when moving to the *Schrödinger* and *Terminated* states are configurable, letting the application select its sensitivity to timeouts. When a node knows it will be away for just a short while, making any failover adoptions unnecessary, it can send a goodbye message to

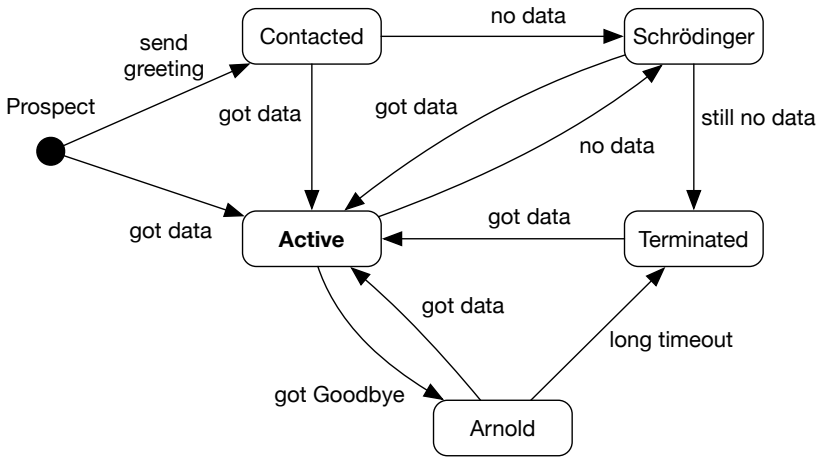

Figure 4: The life cycle of each peer.

the other nodes which puts it in the *Arnold*[5] state. The failover logic is triggered when moving to the *Terminated* state.

2.3 Data Tuple Life Cycle

Figures 5 and 6 illustrate the replication and failover from the perspective of a single 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 represent state changes on the first node, and dashed 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 node_1 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. Also in Figure 3, these are the arrows on the right, from node_1 to node_3 and node_4. When the failover nodes have confirmed this operation, the data tuple on node_1 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., node_3 and node_4 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 later shown as B and C in Figure 8, when a failover node discovers that all earlier nodes in the owners field no longer respond to its heartbeat requests within the stipulated timeout. It then moves the data tuple from state *Inactive* to *Stored*, and informs the application about this change. The life cycle then

[5]It will be back.

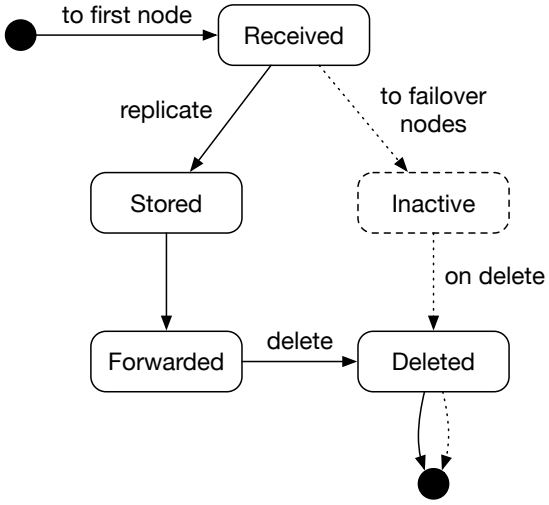

Figure 5: The life cycle of each data tuple on the first node.

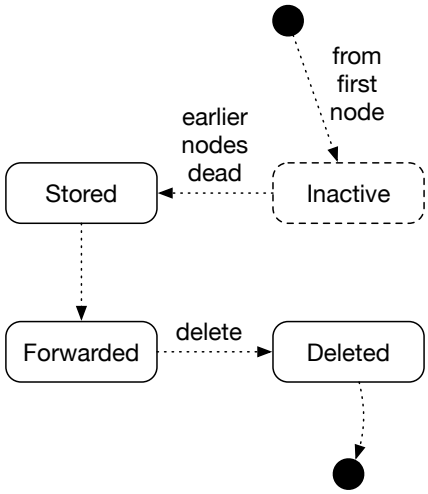

Figure 6: The life cycle of a data tuple in case of failover.

proceeds as above, causing the data tuple to be forwarded and then deleted on any remaining failover nodes. As described in Section 3.3, 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.

In both cases, the flow is monotonic [28] as there are no loops.

2.4 Source Code

The source code, consisting of about 3500 lines of C, is publicly available[6]. This includes both the proof-of-concept implementation of the replication protocol and the test application and scripts used in the evaluations in Sections 4 and 5. ZeroMQ[7] is used for the networking code.

2.5 Evaluation Environment

For the evaluations later in this paper, we used a total of thirteen servers in 2021, all of them being the smallest ones offered by DigitalOcean[8] at that time: 1 GB memory, 25 GB disk, and 1 virtual x64 CPU. They all ran CentOS 7.9, with the working directory on the filesystem XFS. The code was compiled using gcc 4.8.5.

3 Reliability Analysis

The quality model ISO 25010 [33] defines several characteristics for the evaluation of a software product, each one separated into several sub-characteristics. In this section we will focus on the Reliability characteristic, which contains the sub-characteristics Maturity, Availability, Fault Tolerance and Recoverability. Discussing the maturity of a new protocol does not seem meaningful, and the recoverability in terms of how GeoRep handles a lost node was already discussed in Section 2.1.3.

For the evaluations of the availability and fault tolerance of the proposed protocol, we will use the concepts *yield* and *harvest*, respectively, by Fox and Brewer [21]. In Section 3.1 we discuss the availability in terms of the yield, i.e., how likely it is for a producer to be able to find a node in the GeoRep system which accepts a new data tuple. Next, in Section 3.2, we discuss the fault tolerance in terms of the harvest, seen as the probability that the consumer will receive at *least* one copy of each data tuple. Finally, the fault tolerance is again discussed in Section 3.3, now from the perspective of what happens when the communication between two or more nodes fail for some reason, and under which conditions the consumer will get at *most* one copy of a particular data tuple.

3.1 Availability – Yield

The *yield* [21] for GeoRep is the probability for a client to be able to find a set of at least $f+1$ (where f represents the number of nodes that are allowed to fail after data has been received and acknowledged [41], as discussed above) mutually connected and correctly functioning nodes.

To calculate this yield, we define a *node-set* as a set of nodes that can communicate with each other. Each one of n nodes is either part of, or not part of, each such set, giving a total of 2^n sets. If a node has failed, it is put in its own node-set. As we

[6] `https://bitbucket.org/infoflexconnect/leaderlessreplication`
[7] `https://zeromq.org`
[8] `https://digitalocean.com`

only care about sets with a size of at least 2 (i.e. $f+1$, where $f>0$), failed nodes are automatically ignored in our calculations below. There are $\binom{n}{k}$ sets with size k, as that is the number of ways to pick k nodes out of n. For example, consider the configuration in Figure 3, where $n=5$. The number of sets with sizes between 2 and 5 are then 10, 10, 5, and 1, respectively.

GeoRep can use all sets with a size of at least $f+1$, which for $f=1$ there are $10+10+5+1=26$. In contrast, replication protocols which requires a majority of the nodes to work [57] can only use those with a size of at least $(n+1)/2$, which for $n=5$ becomes $(5+1)/2=3$. There are $\binom{5}{3}+\binom{5}{4}+\binom{5}{5}=10+5+1=16$ such sets. The protocols requiring fewer nodes than a majority [38, 42] for a write operation to succeed, achieve this by only allowing predefined node sets, so for n nodes there are typically only n usable node sets. For protocols replicating all data to all other nodes, only a single node set is allowed.

We illustrate the general case in Figure 7, using Pascal's triangle, where the row (starting at 0, shown to the left) is the number of nodes in the system, and the values in the triangle are the number of node-sets with a particular size. The list of 1's along the left side represents the single situation where all nodes are unavailable. The next column on each row, where the value is the same as the number of nodes, represents the cases where only a single node is available. Each following column represents the cases with an increasing number of available nodes. Along the rightmost side are finally the single cases where all nodes are available.

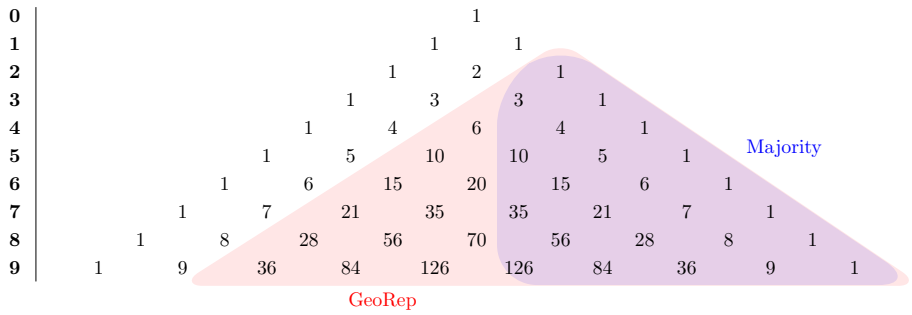

Figure 7: Number of node-sets usable by majority replication and GeoRep, for $f=1$.

The node-sets usable by majority replication are the ones on the right part of Figure 7. As described above, GeoRep can use not only these node-sets, but also the ones to the left except the ones in the first $f+1$ columns.

The total number of node-sets for a particular value of n, is shown in Equation (3) below. The ones usable by GeoRep are then shown by Equation (4). The number of node-sets usable by majority replication are given by in Equations (5) and (6) for odd and even values of n, respectively. For example, going from right to left on row 3, we see that for 3 nodes we can use the single case where all nodes are available, and the 3 cases where 2 out of 3 nodes are available: $2^{(n-1)}=2^{(3-1)}=2^2=4=1+3$.

The ratio between the number of sets usable by GeoRep and the ones usable by majority replication in the best case, is then given by the expression (7), which simplifies

to Equation (8). This value converges to 2 as n increases.

$$\texttt{total} = 2^n \tag{3}$$

$$\texttt{georep} = 2^n - (n+1) \tag{4}$$

$$\texttt{majority\_odd} = 2^{(n-1)} \tag{5}$$

$$\texttt{majority\_even} = 2^{(n-1)} - \binom{n}{n/2} < \texttt{majority\_odd} \tag{6}$$

$$\texttt{ratio} \geq \frac{\texttt{georep}}{\texttt{majority\_odd}} = \frac{2^n - (n+1)}{2^{(n-1)}} \tag{7}$$

$$= 2 - \frac{n+1}{2^{(n-1)}} \tag{8}$$

Generally, we get:

$$\texttt{georep} = 2^n - \sum_{k=0}^{f-1} \binom{n}{k} \tag{9}$$

As the second term in Equation (9) is a polynomial, the second term in Equation (8) will always converge to 0, making the ratio converge to 2 for all values of f. Assuming the producer can connect to any of the system nodes, the availability is therefore about twice as high as for other systems.

There are multiple strategies to use when selecting which of the node-sets to use, for the situations when there are more than 1. The effect the selected strategy has on the system throughput is examined in Experiment 2 in Section 5.4.

3.2 Fault Tolerance – Harvest

The *harvest* [21] is the probability that each data tuple inserted into the system still exists to be output when needed. When this condition is true, the consumer will receive at least one copy of the data tuple. Whether the output is the response to an incoming query or forwarded by the system itself, as in our case, is not really important. For GeoRep we can therefore define the harvest as the probability that at least one of the nodes in the particular subset used for storing an individual data tuple is alive until the data has been forwarded to the consumer (as shown in Figure 2).

This interval from when a data tuple is stored to when it is forwarded is typically less than one second. If a node fails exactly once every 3 years the probability that it happens in any particular second, which we denote as d_{1s}, is

$$d_{1s} = \frac{1}{3 \cdot 365 \cdot 24 \cdot 60 \cdot 60} \approx 10^{-8}$$

(assuming each second is equiprobable[9]). When the node has been repaired or replaced and then restarted, we reset the clock and assume it will run for up to 3 more years.

[9]This is of course a simplification, but we consider it to be an acceptable compromise in the interest of understandability [3].

In the SMS use case, a client may send a large batch of messages faster than the operator(s) can receive them. An operator may also be temporarily unavailable. The resulting queues are typically cleared within a few hours, as the incoming traffic eventually[10] slows down. The probability that the node that received the messages dies within this time, say 3 hours, is

$$d_{3h} = 1 - (1 - d_{1s})^{3 \cdot 60 \cdot 60} \approx 10^{-4}.$$

As the nodes are geographically distant from each other, we can further assume their failures are independent. The formula for the harvest as defined above, then simply becomes $1 - d^{f+1}$, for the relevant value of d. For the normal case when data is forwarded within a second, we get a harvest for $f = 1$ of about $1 - 10^{-8(f+1)} = 1 - 10^{-16}$, a.k.a. "16 nines". For data that stays in the system for 3 hours, we instead get a reliability of $1 - 10^{-4(f+1)} = 1 - 10^{-8}$ for $f = 1$ and $1 - 10^{-12}$ for $f = 2$. SMS brokers with systems where queues are frequent might therefore want to replicate to two other nodes, but more than that is mostly just a waste of network bandwidth. Please also see Table 3 in Section 4, where only one of the nine test cases required a forth node to be available to avoid data loss.

For replication protocols using the AllToAll strategy, sending all data to all nodes, we instead get a harvest of $1 - d^n$. As n grows, this of course converges to even closer to 1, but at the cost of significantly more data traffic.

3.3 Fault Tolerance – 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 8. Neither node_2 nor node_5 have ever heard of this message, so whether they remain in contact with the other nodes has no effect here.

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. Here, the status of node_4 does not matter.

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

There is no way for a node to know if any of the other nodes are dead or are unreachable for another reason, e.g., being unusually slow [4, 41]. 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 of these two latter cases the message is delivered only once.

[10] There is a limited number of SMS recipients in the world, so barring client-side software faults, there will never be an infinite stream of messages.

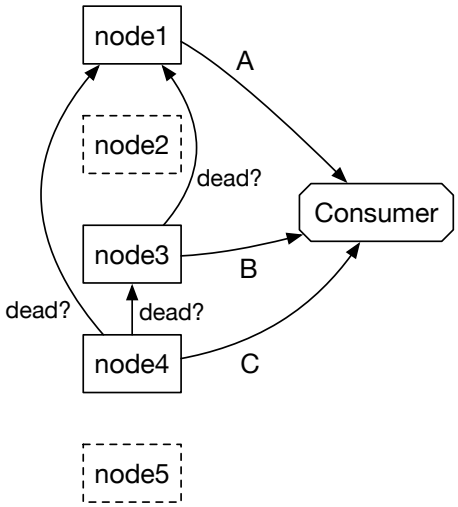

Figure 8: Possible duplications.

4 Failover Verification

As we see it, the most important functionality that needs verification is that data tuples inserted into the system are adopted and subsequently forwarded by another node if the original node becomes unreachable. More specifically, a data tuple should only be adopted by the first node in its *owners* list where all preceding nodes have become unreachable.

For the verification of this functionality, we will construct a small set of test cases, when taken together, provide good coverage of the various combinations of node failures that may occur. The idea, described in more detail in Section 4.1, is for each test to start a set of nodes, insert a single data tuple, and then simulate the failure of zero, one, or more of the nodes. Finally we will examine which node or nodes considers themselves selected to forward this data tuple to the consumer, as shown in Figure 6 in Section 2.3.

For the test case construction, we defined five different categories of nodes. At the top level we had the nodes in the *owners* list plus the *rest* of the nodes. Of the owners, we had one *originator* and a list of failover nodes, here called *peers*. Of those peers, we distinguished between the *first* one, the ones in the *middle*, and the *last* one. These three peer groups allowed at least one peer to have other peers before it in the *owners* list, after it, and both.

Next, we assigned a number to each category as follows, and as shown in Table 1: originator=1, first=2, middle=4, last=8, rest=16. Finally we created a sum of the values representing nodes that had become unavailable. As the selected values are powers of 2, this sum can be seen as a bitmask, where the bit value 0 meant the nodes in this category were still reachable, and 1 that they were not. For example, the bitmask value $00001 = 1$ meant only the originator was unreachable, and $01100 = 12$ that the

originator and the first failover peers was still reachable, as well as the non-peer nodes (in the *rest* group), but not any of the other failover peers. This way we got a set of 32 unique test cases, numbered from 0 to 31, providing a reasonable coverage of possible server and network outages as each test case represented the situation where zero or more nodes in each of these categories became unavailable to all other nodes.

Table 1: The five different node categories, and their assigned bitmask values.

owners	**originator**		1
	failover peers	**first**	2
		middle	4
		last	8
rest			16

Of the total set of 32 possible test cases, all even numbered ones mean the originating node is still alive and reachable. Therefore no adoption should occur in any of these cases. Next, the test cases 16–31 are the same as the cases 0–15, as the reachability of nodes not in the *owners* list have no effect, regardless of how many they are. This leaves us with just 9 distinct test cases, listed in Table 2. We note that in cases 0 and 15, no adoption is made. In case 0, as there is no need for it, and in case 15, as there is no owner left alive to do the adoption. Our system model, described in Section 1.1, only guarantees that data will not be lost if at most f nodes become unavailable, not that at most f nodes actually will. In case 15 there is simply an unfortunate subset of $f+1$ nodes being unavailable, corresponding exactly to the nodes storing the tested data tuple, i.e., both the original node and all failover peers.

Table 2: Relevant tests cases.

Number	**Unreachable**	**Adopter**	**Minimum** f
$0 = 00000$	none	none	1
1	originator	first	1
3	originator and first	middle	2
5	originator and middle	first	3
7	originator, first and middle	last	3
9	originator and last	first	3
11	originator, first, and last	middle	3
13	originator, middle and last	first	2
$15 = 01111$	all owners	none	1

Finally, the test cases listed in Table 2 was mapped to concrete servers. This mapping is shown in Table 3, where nodes that should become unreachable are marked with *italics* and nodes that should adopt the message(s) are marked with **boldface**.

The rest of this section contains the details regarding the implementation and execution of these test cases, as well as the results.

Table 3: Mapping test cases to servers, marking which ones should become *unreachable* and which ones should **adopt** the replicated data tuples.

Number	originator	first	middle	last
0	node_1	node_2	node_3	node_4
1	$node_1$	$\mathbf{node}_2$	node_3	node_4
3	$node_1$	$node_2$	$\mathbf{node}_3$	node_4
5	$node_1$	$\mathbf{node}_2$	$node_3$	node_4
7	$node_1$	$node_2$	$node_3$	$\mathbf{node}_4$
9	$node_1$	$\mathbf{node}_2$	node_3	$node_4$
11	$node_1$	$node_2$	$\mathbf{node}_3$	$node_4$
13	$node_1$	$\mathbf{node}_2$	$node_3$	$node_4$
15	$node_1$	$node_2$	$node_3$	$node_4$

4.1 Experiment Design

The critical point for a data tuple is the transfer from *Inactive* to *Stored*, shown in Figure 6 in Section 2.3, which in turn will trigger at least one of the nodes in the *owners* list to hand the data tuple over to the application so it can ultimately be forwarded to the consumer. To simulate this sequence of events, we created a test application that performed the following steps.

1. Create a single data tuple.
2. Replicate the data tuple to all other nodes, and wait for confirmation.
3. Block all outgoing traffic from a selected subset of nodes, as specified in Table 3.
4. Wait some time to allow the blocked nodes to reach the state *Terminated* in Figure 4 in Section 2.2, triggering the data tuple adoptions.
5. Examine the log files created on each node, to see which node or nodes adopted the data tuple.

4.2 Factors and Variables

For this evaluation, the only independent factor was the set of nodes which should be made unavailable, and the only dependent variable was the set of nodes adopting the data. Based on Table 3, all test cases in this section used $n = 4$ and $f = 3$. We also used a fixed peer order to ensure the roles of each node was predictable. Preliminary tests showed that the number of clients and messages had no effect on the behaviour, so we set both of these parameters to 1. As the adoptions were performed based entirely on local information, the concepts of recovery time, time to elect a new leader and so on, commonly evaluated for other replication protocols, were not relevant to us. This also made it possible to run all tests in a local environment. The factors and variables are summarized in Table 4 for an easy overview.

Table 4: Experiment factors for the failover evaluation.

Type	**Factor**	**Value(s)/Unit**
Independent	Disabled node(s)	None, 1, 2, 3, and/or 4
Constants	Servers, n	4
	Protection, f	3
	No of clients	1
	No of messages	1
	Separation	local
Dependent	Adopter	node number(s)
Ignored	Recovery time	seconds

4.3 Execution

The tests were implemented by adding a filter between the main GeoRep logic and the ZeroMQ interface, making it possible on the application level to prevent any outgoing traffic to one or more particular other peer nodes. A flag was added to GeoRep to always replicate data tuples to nodes sorted in alphabetical order on their identifier ("node1", "node2", etc), making the behaviour predictable. A small shell script, `run-failover.sh`, was used to ensure all executions used the correct parameters, and that data was collected in the same way for all test cases.

4.4 Results

Table 5 shows the results for each one of the test cases. For test case 0, no node was blocked, and therefore no adoptions by other nodes occurred. For the other test cases, we notice that the correct node, as specified in Table 3, does indeed adopt the replicated data.

Table 5: Failover results, showing *blocked* nodes and the ones **adopting** any data tuples.

Number	**node**$_1$	**node**$_2$	**node**$_3$	**node**$_4$
0				
1	*blocked*	**adopts**		
3	*blocked*	*blocked* / **adopts**	**adopts**	
5	*blocked*	**adopts**	*blocked* / **adopts**	
7	*blocked*	*blocked* / **adopts**	*blocked* / **adopts**	**adopts**
9	*blocked*	**adopts**		*blocked* / **adopts**
11	*blocked*	*blocked* / **adopts**	**adopts**	*blocked* / **adopts**
13	*blocked*	**adopts**	*blocked* / **adopts**	*blocked* / **adopts**
15	*blocked*	*blocked* / **adopts**	*blocked* / **adopts**	*blocked* / **adopts**

Except for node$_1$, all blocked nodes also adopt the replicated data tuples. The reason for this is that as they are blocked, they never get any life signs from the other nodes and therefore must consider these too to be unreachable. As discussed in Section 3.3,

this would however rarely lead to any data duplications. If a node cannot reach the other nodes, it is simply not likely that it could still reach the consumers as shown in Figure 2.

5 Throughput Evaluation

For an evaluation of the proposed protocol primarily focused on quality attributes, we designed a controlled experiment [50], following the guidelines by Basili *et al.* [8] and Pfleeger [48]. The overall goal was to evaluate the throughput in a few different configurations.

5.1 Experiment Design

We used a sequence of tasks corresponding with the queue related operations performed by the type of systems described as our system model in Section 1.1, resulting in realistic experiments. We created a test application which created the messages itself, and discarded them when all tasks described below were completed. Due to the nature of the experiment, we were able to use a fixed design [59].

1. A new message was stored locally and replicated according to the selected configuration. The application waited for acknowledgements from the others servers before returning control to the application.
2. A message was extracted from the queue.
3. The extracted message was deleted from all servers where it was stored.

A benchmark suite commonly used for evaluating replication systems is the Yahoo! Cloud Serving Benchmark (YCSB) [15], which exists in several different variants with varying proportions between writes and reads. Using the same 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.

5.2 Factors and Variables

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 6. The throughput was measured in messages per second (MPS).

5.2.1 Independent Factors

The primary factors in these experiments were selected to give a deeper understanding of the behaviour under different circumstances.

The number of servers was varied from 2 to 7. The number of client connections was varied between 1 and 1000. For clarity, only subsets of these intervals are shown in the diagrams below.

Table 6: Experiment factors.

Type	Factor	Value(s)/Unit
Independent	Servers, n	2…7
	Clients	$1, 3, 10, \ldots, 1000$
	Separation	Local, Geographical
	Protection, f	0, 1, 2
Constant	Transient	5 s
	Steady-state	30 s
Dependent	Throughput	MPS
	Min RTT	Microseconds, μs
Ignored	Recovering	MPS
	Duplications	Ratio

We used servers both within the same data center and in multiple time zones. This way we could examine the effect the physical distances between the servers, and thereby the different round-trip times, had on the system throughput. The data centers used for the different numbers of servers, are shown in Table 7. The idea was to keep the sites as geographically separated as much as possible. Only when using 6 or 7 servers did we use data centers relatively close to each other.

Table 7: Data centers used for the Geographical cases.

Data center	Number of servers					
	2	**3**	**4**	**5**	**6**	**7**
San Francisco		✓	✓	✓	✓	✓
Toronto						✓
New York	✓	✓	✓	✓	✓	✓
London					✓	✓
Amsterdam	✓	✓	✓	✓	✓	✓
Bangalore			✓	✓	✓	✓
Singapore				✓	✓	✓

We motivate setting the protection f to just 1 or 2 by recalling the discussion about reliability in Section 3.2. For normal operations, where messages are forwarded within the same second as they were received, even setting f to such a low value as 1 gives a reliability of about $1 - 10^{-16}$. Replicating to a single other independent node is therefore normally enough. To see the effect of replicating to a larger number of nodes, which may be useful when the queues become long, we also ran the tests with $f = 2$. To see the performance degradation caused by the replication logic, we ran a few tests with $f = 0$.

The reliability of the power and internet infrastructure is also relevant, but these factors mainly affect the availability of the system, not its fault tolerance. We get high availability by having a large number of possible node sets, and as we saw in Figure 7 in Section 3.1, the most effective way to increase the number of such sets is to increase

the number of nodes, n. This value is already selected as one of the independent factors.

5.2.2 Constants

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 application continued to run in the steady-state phase for another 30 seconds.

5.2.3 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 5.1.

We also measured the minimum RTT between each pair of nodes. The median round-trip time would be more relevant for answering the question of what a typical response time would be. However, as discussed in the Introduction, we are more interested in the system resilience, achieved by replicating the data tuples to nodes at some minimum physical distance from each other. A large RTT clearly is no guarantee that the nodes are far apart, but due to the finite speed of light, a small RTT requires the nodes to be near each other.

5.2.4 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.

5.3 Execution

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 as needed. The application was then started on all servers, with the selected values for the independent variables provided 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. Finally, the median of the values for each of the 30 seconds in the steady-state phase was calculated.

5.4 Results

Here we present a summary of the results from our throughput evaluations, made to establish an initial intuition of how this protocol behaves. As mentioned, we varied the number of servers up to 7, and the number of clients up to 1000, even though the diagrams just show the results for representative subsets.

Figure 9: System throughput as a function of the number of servers, all running in the same data center. Here, $f = 1$.

In a local network, the total system throughput increased with the number of nodes up to 33 181 MPS on 7 nodes with 300 clients, shown in Figure 9. The minimum RTT varied between 143 μs and 420 μs. When setting $f = 2$, as shown in Figure 10, the throughput was similar given the same remaining factors, but consistently 20–40% lower.

When GeoRep was deployed in a cluster of geo-separated servers, throughput again increased with the number of nodes. The peak throughput levels were much lower than in the local case, due to the longer round-trip times. For the same reason, the system spent more time waiting for responses, lowering the CPU load. This allowed us to increase the number of clients to 1000. Figure 11 shows how the throughput reached 7609 MPS for 2 nodes and 24 562 MPS for 7 nodes. As can be seen in Figure 12, when setting $f = 2$ the throughput was again consistently lower than for $f = 1$.

In Figure 13 we see the performance hit resulting from the replication logic. The entries for $f = 0$ show the case when not using any replication at all. Other than occasional heartbeat traffic, the executed program code in GeoRep is just a very thin layer on top of LevelDB. As expected, the throughput scales almost linearly by the number of nodes, around 35–40 kMPS per node.

For 3 geo-separated nodes, the minimum RTT averaged 105 ms. For 7 nodes, the relatively distant nodes in Bangalore and Singapore resulted in an increase to 138 ms. Figure 14 shows the RTT between a few selected pairs of nodes. For example, the RTT from Toronto (in column 3) to New York is quite low, almost the same to San Francisco and Amsterdam, and quite long to Bangalore. The profiles for nodes geographically close to each other, e.g., New York and Toronto, are notably similar.

Based on Figure 14, we saw that instead of replicating messages to a random selection of nodes, we could select the f ones with the smallest RTT from where the

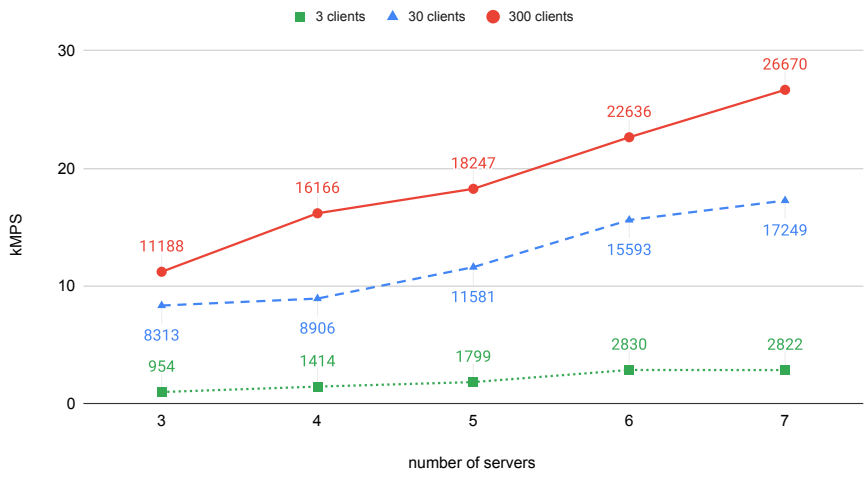

Figure 10: System throughput as a function of the number of servers, all running in the same data center. Here, $f = 2$.

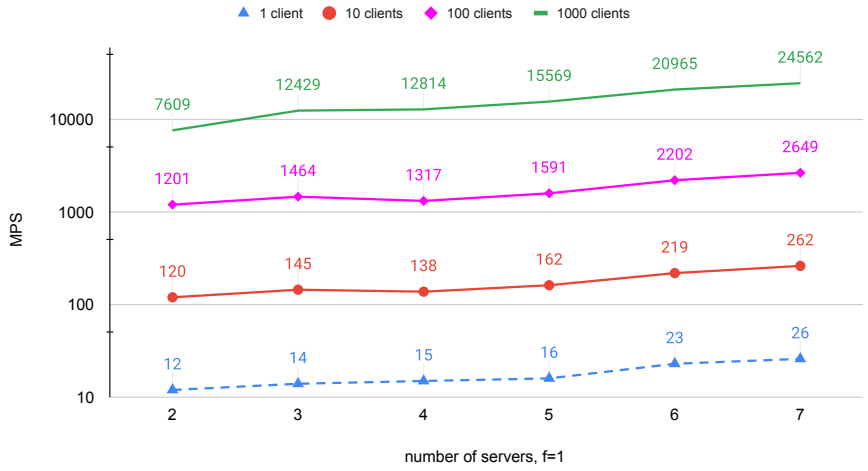

Figure 11: System throughput as a function of the number of servers, running in different data centers on multiple continents. Please note that the Y axis is logarithmic, to match the logarithmic increase in the number of clients. Here, $f = 1$.

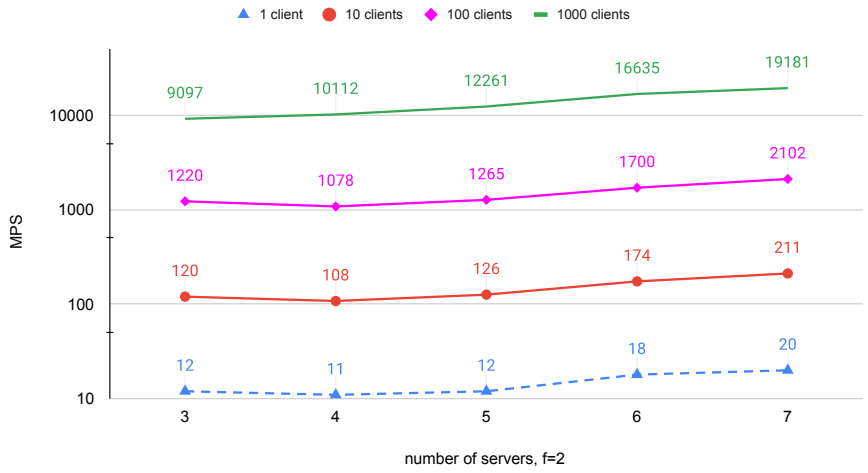

Figure 12: System throughput as a function of the number of servers, running in different data centers on multiple continents. Please note that the Y axis is logarithmic, to match the logarithmic increase in the number of clients. Here, $f = 2$.

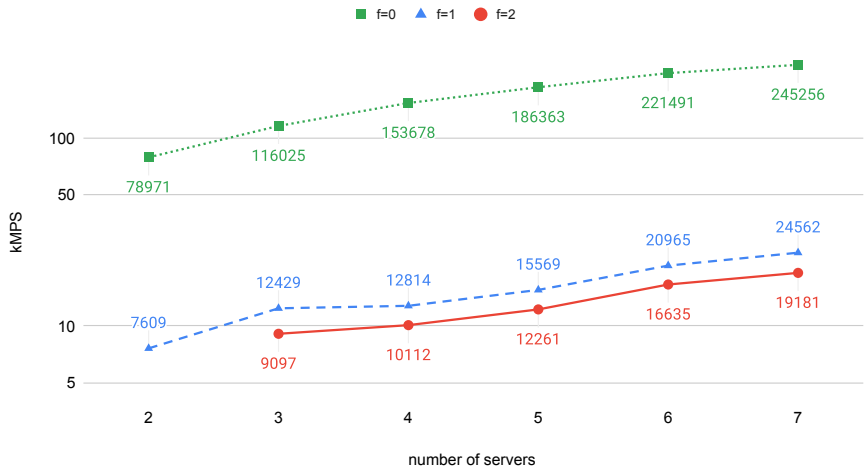

Figure 13: System throughput as a function of the number of servers, running in different data centers on multiple continents, when varying f between 0, 1, and 2. The number of clients is 1000.

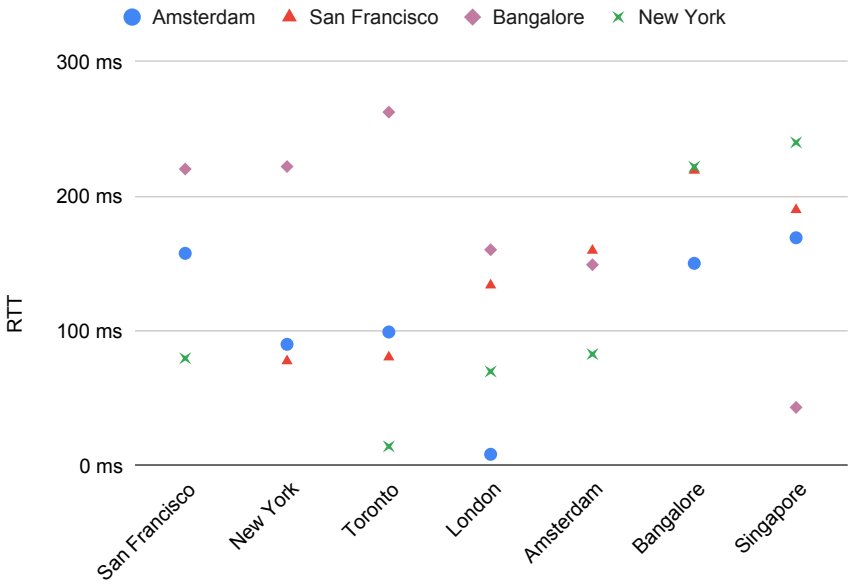

Figure 14: Round-trip time (RTT) for various pairs of servers.

message was received, ignoring nodes with an RTT lower than some predefined limit, say 10 ms. This minimum value ensures messages are always replicated outside of the critical region mentioned in the Introduction.

We set the number of servers to 7, and varied the number of clients between 100 and 1000. We varied the minimum RTT limit between 1, 20, and 100 ms, based on the following reasoning. A minimum of 1 ms prevents a node from replicating to another node within the same data center. This level protects from local internet and power outages. The RTT between New York and Toronto, and between the nodes in Europe, is around 10 ms. By setting a minimum of 20 ms, these nodes must find peers further away, such as the one in California or one across the Atlantic. This level protects from larger outages covering bigger areas. When increasing the limit to 100 ms, we also prevent replication within the American continent and between the American east coast and Europe. The data tuples are then always replicated at least about one third of the total circumference of the earth. Increasing the limit further would not have any practical application. With a larger number of nodes in more parts of the world, other RTT limits would be meaningful, offering a larger number of tradeoff points between throughput and reliability. The achieved throughput for the three tested cases are shown in Figure 15 for $f = 1$, and in Figure 16 for $f = 2$.

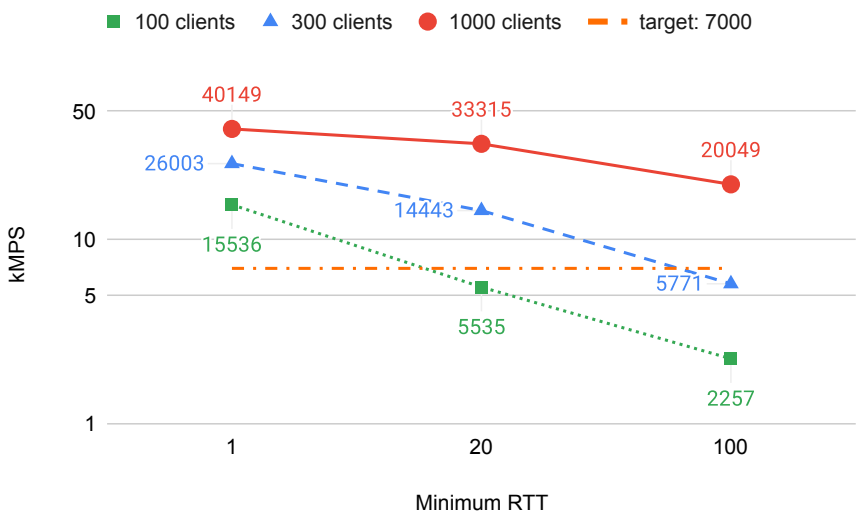

Figure 15: System throughput for various minimum RTT limits, with $f = 1$.

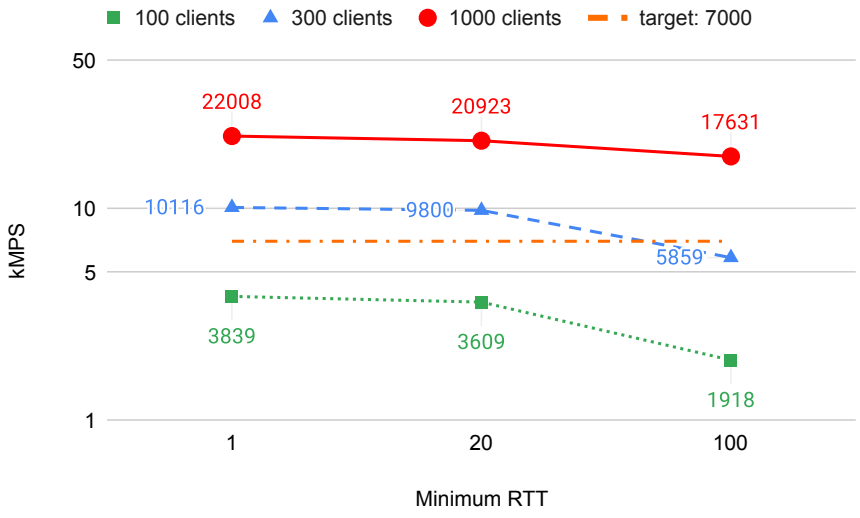

Figure 16: System throughput for various minimum RTT limits, with $f = 2$.

6 Discussion

In our experiments, the proposed protocol was shown to being able to leverage the ordering independence of the data tuples and thereby perform better as the number of clients, and thereby also the number of parallel requests, increased. As shown in Section 5, the highest recorded throughput for the geo-distributed case was a bit over 33 000 MPS when using 7 servers sufficiently far apart to avoid having more than 1 server fail due to a single power or network outage.

Figure 13 provides a way for us to calculate the implementation efficiency, where there is room for improvement. Each node performs the same local LevelDB operations for data tuples replicated to them, so the theoretical maximum throughput are the values for $f = 0$, divided by $f + 1$. The entries for $f = 1$ and $f = 2$ are from Figure 11 and Figure 12, respectively, in both cases around 20–25% of this maximum. For example, for 7 nodes we get an efficiency of 20% for $f = 1$ as shown in Equation 10, and 23% for $f = 2$ as shown in Equation 11.

$$\texttt{efficiency}_1 = \frac{24562}{245256/(1+1)} \approx 20\% \tag{10}$$

$$\texttt{efficiency}_2 = \frac{19181}{245256/(2+1)} \approx 23\% \tag{11}$$

A similar slowdown for higher values of f as is shown in Figure 13, can be seen by comparing Figure 15 and Figure 16. The independence between the data tuples still enables us to reach much more than our target 1000 MPS per node, as long as there are sufficiently many clients.

Our design does not consider network partitions in general to be errors, so we could therefore often avoid steps 2 and 3 of the D2R2 model from ResiliNets: *detect* and *remediate* [44]. As was elaborated in Section 3, this gave us better resilience [22] than what AllToAll systems could provide, as well as up to twice the availability.

6.1 Threats to Validity

The identified validity threats are grouped [14, 34] for better overview.

6.1.1 Internal

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

In a production environment, the client applications will of course not run on the same machine as GeoRep. Separating them will result in more time passing for the client, between submitting a data tuple for replication, and getting the confirmation back. On the other hand, it will leave more CPU to GeoRep, possibly increasing its performance for the CPU bound parts.

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.

6.1.2 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. Applications where this is true besides SMS gateways, are email gateways. These gateways also route messages from companies to their customers, but instead of delivering messages to network operators, they are delivered to email servers and ultimately to the customers' mailboxes. Here too, the relative order between messages does not matter, there are no reliable end-to-end acknowledgements,[11] and each message is important to its recipient. The quality requirements for these systems are indeed similar to the ones for SMS gateways, as the system must provide high availability to the senders, and as messages must not get lost despite temporary failures of both system nodes and recipient systems.

7 Related Work

7.1 Network Reliability

Kleppman presented some critique of the CAP theorem, offering an alternative framework [36] using the terms "Availability", "Delay-sensitive", "Network faults", "Fault tolerance" and "Consistency". This would be more suitable for practitioners, separating meaningful delays which are typically very short, from "eventually", which may be anything less than infinity. Even though we do not use that framework in our work, it supports our claim that there is a need for, and room for, models and frameworks which are closer to the practical reality than what CAP suggests.

7.2 Replication Protocols

Other store-and-forward systems are application-to-application message queues, e.g. Apache Kafka [37]. 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.

[11] A common workaround for emails are tracking pixels, but these are usually possible to disable on the client side. Some email services, e.g., `hey.com`, see them as a threat to privacy and explicitly blocks them.

For systems where a global ordering must be maintained, the replication protocols are often based on a variant of Paxos [39] or Raft [46], recently proven by Howard and Mortier to be functionally identical [30]. The Paxos variant Mencius [43] was designed to perform well even in wide-area networks with high inter-node latency. One of the ways they achieve this is by using 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.

Even in cases where the bandwidth is not the primary issue, such as between the cores within the same CPU, coordination has a significant cost. Qadah and Sadoghi demonstrated this with their key-value store QueCC [49]. The round-trip times between CPU cores is of course several orders of magnitude smaller than between servers in different data centers or time zones, but the coordination between the servers is still so costly that by avoiding it, QueCC is about 5 times faster than comparable systems.

Many of the data replication protocols are based on total order broadcast [17], 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.

A naïve solution would be to store the data tuples in an SQL database, where there are plenty of replication methods. However, as SQL databases must maintain the ACID (Atomicity, Consistency, Isolation, and Durability) [25] properties of the data, those methods work best within a local server cluster. With geo-separated servers, the higher round-trip times cause a significant performance degradation. Comparing GeoRep with an SQL database, given the same operations, would therefore not be fair.

8 Conclusions and Future Work

With the purpose of increasing the reliability of a store-and-forward system, we designed a solution based on application semantics instead of lower level storage operations [27,28]. We took advantage of the lack of a relative order between the data tuples, and that not all data tuples had to be replicated to all servers. This not only allowed us to achieve the desired reliability while maintaining sufficient throughput, but also gave almost twice the system availability compared to other replication protocols.

Naturally, we welcome replication studies of this protocol. The experiments can be varied along several different dimensions, e.g., a) using other programming languages than C, b) using other frameworks than ZeroMQ, c) using a larger number of nodes, d) separating the client applications into separate nodes, e) and considering other use cases and application areas. The source code used in the experiment is open sourced to facilitate such studies.

There is no consensus among the nodes regarding the reachability of the other nodes, so the number of use cases for the failover verification in Section 4 is actually higher than 9, and increases with higher values of f. A deeper analysis to find the exact formula for which of these test cases involving the reachabilities from multiple nodes can actually occur, their expected outcome, and comparing this with the actual behaviour, would be interesting, but is left as future work.

For predictable disasters [45], e.g., hurricanes, floods and tsunamis, we should be able to temporarily disable some servers beforehand as replication targets, to minimize data loss. The same strategy could even be used for more unpredictable disasters causing power failures, in those cases triggered by the affected nodes switching to battery power.

Acknowledgements

This work was sponsored by The Knowledge Foundation industrial PhD school ITS ESS-H, H2020 project ADEPTNESS (871319) and Braxo AB. Thanks to Per Erik Strandberg and Daniel Flemström for assistance with the mathematics.

References

[1] Giuseppe Aceto, Alessio Botta, Pietro Marchetta, Valerio Persico, and Antonio Pescapé. A comprehensive survey on internet outages. *Journal of Network and Computer Applications*, 113(2018):36–63, jul 2018.

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

[3] Peter A. Alsberg. Research in Network Data Management and Resource sharing, 1976.

[4] Peter A. Alsberg, Geneva G. Belford, Steve R. Bunch, John D. Day, Enrique Grapa, David C. Healy, Edwin J. McCauley, and David A. Willcox. Research in Network Data Management and Resource Sharing, Synchronization and Deadlock. Technical report, Center for Advanced Computation, University of Illinois, 1977.

[5] Peter A. Alsberg and John D. Day. A Principle for Resilient Sharing of Distributed Resources. In *Proceedings - International Conference on Software Engineering*, ICSE. IEEE Comput. Soc. Press, 1976.

[6] Peter Alvaro. *Data-centric Programming for Distributed Systems*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2015.

[7] Peter Bailis and Kyle Kingsbury. The Network is Reliable. *Communications of the ACM*, 57(9):48–55, sep 2014.

[8] Victor R. Basili, Richard W. Selby, and David H. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE-12(7):733–743, 1986.

[9] Martin Biely, Zarko Milosevic, Nuno Santos, and André Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 2012.

[10] Daniel Brahneborg, Wasif Afzal, Adnan Causevic, and Mats Björkman. Super-linear and bandwidth friendly geo-replication for store-and-forward systems. In *15th International Conference on Software Technologies*, July 2020.

[11] Susanne Braun and Stefan Desloch. A Classification of Replicated Data for the Design of Eventually Consistent Domain Models. In *International Conference on Software Architecture Companion*, ICSA-C. IEEE, 2020.

[12] Eric A Brewer. Towards Robust Distributed Systems. In *Principles Of Distributed Computing*. ACM, 2000.

[13] Yufei Cheng, M Todd Gardner, Junyan Li, Rebecca May, Deep Medhi, and James PG Sterbenz. Analysing GeoPath diversity and improving routing performance in optical networks. *Computer Networks*, 82:50–67, 2015.

[14] Thomas D Cook and Donald Thomas Campbell. *Quasi-experimentation: Design and analysis for field settings*, volume 3. Rand McNally, Chicago, 1979.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM symposium on Cloud computing*, SoCC '10, New York, NY, USA, 2010. ACM.

[16] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in Partitioned Networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[17] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms. *ACM Computing Surveys*, 36(4):372–421, 2004.

[18] Edsger Wybe Dijkstra. Co-operating sequential processes. In *Programming languages*. Academic Press Inc, 1968.

[19] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[20] Michael J. Fischer and Alan Michael. Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network. In *Proceedings - ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS, 1982.

[21] Armando Fox and Eric A. Brewer. Harvest, Yield, and Scalable Tolerant Systems. In *Proceedings - Workshop on Hot Topics in Operating Systems*, HOTOS. IEEE, 1999.

[22] Ivan Ganchev, Jacek Rak, Tibor Cinkler, and Máirtín O'droma. Taxonomy of schemes for resilient routing. In *Guide to Disaster-Resilient Communication Networks*, pages 455–482. Springer, 2020.

[23] Seth Gilbert and Nancy A. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. In *Principles Of Distributed Computing*, PODC, 2004.

[24] Neil Gunther, Paul Puglia, and Kristofer Tomasette. Hadoop superlinear scalability. *Queue*, 13:20–42, 5 2015.

[25] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[26] Coda Hale. You can't sacrifice partition tolerance. *Retrieved May 2020*, 2010.

[27] Pat Helland and Dave Campbell. Building on quicksand. In *Proceedings - Conference on Innovative Data Systems Research*, CIDR. ACM, 2009.

[28] Joseph M Hellerstein and Peter Alvaro. Keeping calm: when distributed consistency is easy. *arXiv preprint arXiv:1901.01930*, 2019.

[29] Joseph M. Hellerstein and Peter Alvaro. Keeping calm. *Communications of the ACM*, 63, 8 2020.

[30] Heidi Howard and Richard Mortier. Paxos vs raft: Have we reached consensus on distributed consensus? In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, 2020.

[31] David Hutchison and James P.G. Sterbenz. Architecture and design for resilient networked systems. *Computer Communications*, 131:13–21, 10 2018.

[32] Farabi Iqbal and Fernando A Kuipers. Disjoint paths in networks. *Wiley Encyclopedia of Electrical and Electronics Engineering*, pages 1–11, 1999.

[33] ISO. ISO/IEC 25010. `https://iso25000.com/index.php/en/iso-25000-standards/iso-25010`, 2021. Accessed 2021-04-06.

[34] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. Reporting experiments in software engineering. In *Guide to advanced empirical software engineering*, pages 201–228. Springer, 2008.

[35] Paul R Johnson and Robert H Thomas. RFC 677: The Maintenance of Duplicate Databases, 1975.

[36] Martin Kleppmann. A Critique of the CAP Theorem, 2015.

[37] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a Distributed Messaging System for Log Processing. In *Proceedings of the SIGMOD Workshop on Networking Meets Databases*, NetDB, Athens, Greece, 2011.

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

[39] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[40] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, July 1982.

[41] Bruce G Lindsay, Patricia G Selinger, Cesare Galtieri, James N Gray, Raymond A Lorie, Thomas G Price, Franco Putzolu, Irving L Traiger, and Bradford W Wade. *Notes on Distributed Databases*. IBM Thomas J. Watson Research Division, 1979.

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

[43] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI, Berkeley, CA, USA, 2008.

[44] Andreas Mauthe, David Hutchison, Egemen K Cetinkaya, Ivan Ganchev, Jacek Rak, James PG Sterbenz, Matthias Gunkelk, Paul Smith, and Teresa Gomes. Disaster-resilient communication networks: Principles and best practices. In *International Workshop on Resilient Networks Design and Modeling*, RNDM. IEEE, 2016.

[45] B. Mukherjee, M. F. Habib, and F. Dikbiyik. Network adaptability from disaster disruptions and cascading failures. *IEEE Communications Magazine*, 52(5):230–238, 2014.

[46] Diego Ongaro and John K Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*, 2014.

[47] Roberto Percacci and Alessandro Vespignani. Scale-free behavior of the internet global performance. *The European Physical Journal B*, 32(4):411–414, Apr 2003.

[48] Shari Lawrence Pfleeger. Experimental design and analysis in software engineering, part 2. *ACM SIGSOFT Software Engineering Notes*, 20(1), 1995.

[49] Thamir M. Qadah and Mohammad Sadoghi. QueCC: A Queue-oriented, Control-free Concurrency Architecture. In *Proceedings of the International Middleware Conference*, Middleware '18, New York, NY, USA, 2018. ACM.

[50] Colin Robson and Kieran McCartan. *Real world research*. John Wiley & Sons, 2016.

[51] Justin P Rohrer, Abdul Jabbar, and James PG Sterbenz. Path diversification for future internet end-to-end resilience and survivability. *Telecommunication Systems*, 56(1):49–67, 2014.

[52] James B Rothnie and Nathan Goodman. A Survey of Research and Development in Distributed Database Management. In *Proceedings - International Conference on Very Large Data Bases*, 1977.

[53] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, Inria – Centre Paris-Rocquencourt, 2011.

[54] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. PaRiS: Causally Consistent Transactions with Non-blocking Reads and Partial Replication. In *IEEE International Conference on Distributed Computing Systems*, ICDCS. IEEE, jul 2019.

[55] Michael Stonebraker and Eric Neuhold. A Distributed Data Base Version of Ingres. Technical report, California University, Berkeley., 1976.

[56] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29(5):172–182, December 1995.

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

[58] Balázs Vass, János Tapolcai, David Hay, Jorik Oostenbrink, and Fernando Kuipers. How to model and enumerate geographically correlated failure events in communication networks. In *Guide to Disaster-Resilient Communication Networks*, pages 87–115. Springer, 2020.

[59] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[60] Mazin Yousif. Cloud Computing Reliability—Failure is an Option. *IEEE Cloud Computing*, 5(3):4–5, may 2018.