IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

GeoRep – Resilient Storage for Wide Area Networks

DANIEL BRAHNEBORG[1], ROMARIC DUVIGNAU[2], WASIF AFZAL[3], SAAD MUBEEN[3]

[1]Braxo AB, Stockholm, Sweden (email: first@braxo.se)
[2]Chalmers Tekniska Högskola, Göteborg, Sweden (email: first.last@chalmers.se)
[3]Mälardalens Universitet, Västerås, Sweden (email: first.last@mdu.se)

Corresponding author: Daniel Brahneborg.

This research was funded by Braxo AB and the research school ITS ESS-H.

ABSTRACT Embedded systems typically have limited processing and storage capabilities, and may only intermittently be powered on. After sending data from its sensors upstream, the system must therefore be able to trust that the data, once acknowledged, is not lost. The purpose of this work is to propose a novel solution for replicating data between the upstream nodes in such systems, with a minimal effect on the software architecture. On the assumption that there is no relative order between replicated data tuples, we designed a new replication protocol based on partial replication. Our protocol uses only 2 communication steps per data tuple, instead of the 3 to 12 used by other solutions. We verified its failover mechanism in a proof-of-concept implementation of the protocol using simulated network failures, and evaluated the implementation on throughput and latency in several controlled experiments using up to 7 nodes in up to 5 geographically separated areas, with up to 1000 data producers per node. The recorded system throughput increased linearly relative to both the number of nodes and the number of data producers. For comparison, Paxos showed a performance similar to our protocol when using 3 nodes, but got slower as more nodes were added. The lack of a relative order, in combination with partial replication, enables our system to continue working during network partitions, not only in the part containing the majority of the nodes, but also in any sufficiently large minority partitions.

INDEX TERMS store-and-forward, replication, distributed computing, resilience, availability

I. INTRODUCTION

ALL over the world, various types of disasters happen with both regular and irregular intervals [1]–[4]. These disasters, which could be caused by natural, technical, political or other kinds of events, affect network and power equipment, and might therefore lead to outages for internet services [5]–[7]. Such infrastructure failures have been showed to be about twice as likely the cause for services being unavailable to clients, as compared to failures in the servers themselves [8]. Oftentimes, these infrastructure failures can be mitigated by using multiple geographically separated servers [1], [9]–[11], conveniently offering protection from failures in both infrastructure and individual servers. The servers exchange data with each other as necessary, allowing clients to connect to any one of them. If the system uses different cloud providers for each data center to mitigate the risk of failures due to software or configuration upgrades [12], the probability for some event killing multiple nodes during the processing of a particular data tuple is effectively zero.

Maintaining the same data on multiple servers is not a new problem. A common solution is to use *full replication*, which sends all information regarding the processed data to all other servers [13]. This is often managed via a master server as in Paxos [14], [15] or Raft [16], ensuring both that all data and its operations are communicated to all servers, and that the operations are processed in the same order [17].

Full replication is easy to understand and reason about, and is implemented in various concrete tools and libraries, e.g., Redis[1] and Spread[2]. It forms the basis for eventual consistency [18], and for Convergent and Commutative Replicated Data Types (CRDTs) [19]. 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. Because the system can freely select one or

[1]https://redis.io
[2]http://www.spread.org

more remaining servers to take over the duties of a failed server [20], this also makes resilience, as described by the ResiliNets project [3], [21], straightforward. Resilience is then the degree of how well a system can recover from failures. This differs from robustness, which is how well the system behaves during normal operations.

However, full replication also has a number of shortcomings. It wastes network traffic [2], [22], as the amount of transmitted data grows at least linearly by the number of servers in the system. It requires all servers to be able to reach each other, possibly going via one or more other servers. When there is a network partition, by which we mean any type of failure breaking full reachability, system availability [10], [23]–[26] is reduced as clients can then only perform updates on the nodes in the remaining majority part, if any. The required coordination can be costly [27], [28] and limit system performance.

In this work, we envision an application providing a message queue for event data sent from sophisticated sensors or IoT devices. The data tuples are added to the queue by the devices, and then one by one pushed by the queue itself to the service responsible for that particular type of data. After being successfully forwarded, each data tuple is removed from the queue.

The queue's push construct has a few important implications, making previous state-of-the-art non-optimal. One of the explicit goals in current work on replication is that the data tuples should be delivered and thereby be visible to all other nodes. An alternative to this full replication is to use the more resource conservative partial replication, which only sends data tuples to a subset of the servers [10]. In our use case, each message needs to be visible to just one single server, to ensure that it is delivered only once. It is not until a server fails that the application layer on the other servers should be made aware of its messages, again only on a single server per message.

As each data tuple is independent, we have no need for consistent operation ordering, and therefore do not need any mechanism for enforcing this order [29]. As there are no external readers pulling messages from the queue, we also do not need all nodes to receive the same set of data and its operations, and thus have no use for the consistency guarantees provided by full replication.

Partial replication saves both network and other resources compared to full replication, but makes it difficult to maintain a consistent, global order between data tuples. Previous works in this area [30]–[34] solve this by using some variant of atomic broadcast [35]–[39]. Unfortunately that solution requires additional network traffic (between 1 and 10 communication steps, depending on the protocol) and relatively complex algorithms. This creates a problem with scalability, which can be observed in the literature on this topic by noticing that the system throughput does not always increase when new nodes are added. The throughput typically falls relatively quickly when the number of nodes to replicate to increases. This can, for example, be seen in the evaluation of GentleRain [40], where the throughput increases significantly slower when there are more than about 10 servers.

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, with better scalability than in state-of-the-art. The resulting design was evaluated using a proof-of-concept implementation, tested on servers scattered across multiple continents. Even on servers with modest performance, we achieved up to 3440 MPS per node in the geo-diverse case, replicating each data tuple to a random other server in the world. By always using the nearest server, e.g., from New York to Toronto, we instead reached 5661 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 its reliability in terms of availability, potential data loss, and potential data duplication.
3) A method to verify its failover mechanism.
4) A performance analysis on throughput, both when deployed within a local network and for a geo-distributed system configuration.
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 II describes the proposed protocol. Next follows evaluations of the protocol from three different perspectives. First, Section III contains a theoretical analysis of the reliability. Then, Section IV describes the verification of the failover mechanism, and finally Section V describes the setup for the experiments conducted to evaluate the behaviour in a real-world configuration, focusing on the quality attribute throughput. The results are discussed in Section VI, and related work in Section VII. Section VIII holds conclusions and some ideas for future work.

This paper is an extension to the previously published conference paper [41] presenting this protocol. The main differences between that version and this updated article, are Section I-C discussing our requirements, the extension of the "Duplication Analysis" subsection into a more complete Reliability Analysis in Section III, the failover verification in Section IV, and an extended list of references.

A. SYSTEM MODEL

Our system model is a classic store-and-forward queue [42], with external sets of producers and consumers [43]. 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. When acknowledged by the consumer, the data tuple is removed from our system. The data tuples are therefore managed by the queue for a relatively short time period, normally less than 1 second. There are no end-to-end acknowledgements.

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 knows about all other nodes, 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. 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 that each producer maintains 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 systems. Due to this, a server cannot inform clients about the other servers, unless that is already part of the protocol between clients and servers.

The data tuples contain the following fields.

id A globally unique id.

payload
Opaque application specific payload.

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*. The value of f is typically 1 or 2.

We use the term "majority replication" for all data replication protocols based on inequality (1) below. Full replication normally uses *number of nodes to send write operations to (w)* = n and *number of nodes to read data from (r)* = 1, which trivially satisfies this condition [44]. Another variant is to wait for acknowledgements from at least $\lfloor n/2 \rfloor + 1$ nodes for both write and read operations [45].

$$w + r > n \tag{1}$$

Security concerns such as authentication and encryption are not part of the model. There are also no byzantine failures [46], with nodes sending arbitrarily erroneous data.

B. 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 event data from their IoT devices, authentication codes, meeting reminders and similar information. Using SMS makes it possible to reach all customers without them having to install any additional software on their mobile phones. Fig. 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. In our system model, the clients are the producers, and the operators are the consumers.

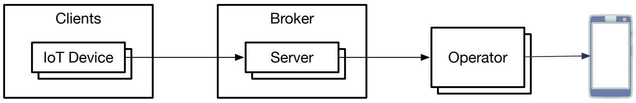

FIGURE 1. Companies sending text messages from multiple IoT devices, an SMS broker with multiple servers, mobile network operators, and customers' mobile phones.

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. 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, in total a few hundred bytes.

The network operators implement their own message queues, making the mobile phone user 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 explicitly minimize these.

C. PROBLEM STATEMENT AND REQUIREMENTS

For our store-and-forward system model in general, and our SMS application in particular, the problem addressed in this paper is to find a way to replicate the forwarded data tuples as effectively as possible, with minimal changes to an existing application. By "effectively" we mean high throughput and low CPU and network usage.

Next, we summarize our requirements, which are based on current industry standards for SMS traffic in general. An overview of the required data flows for a configuration with two nodes is shown in Fig. 2. A program, named ExampleApp, is running on each node, using a context independent subsystem implementing the replication protocol. In the figure this subsystem is called GeoRep, as that is the name of our proposed solution. 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 Fig. 1. ExampleApp then tells the replication subsystem 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 Fig. 1, it tells GeoRep to delete the data on both nodes.

The GeoRep subsystems communicate with each other for replication and failure detection. When a failed node has been detected, GeoRep tells ExampleApp on the working node to forward the data tuples originally received by the failed node. So, ExampleApp does not know anything about replication, and GeoRep knows neither of the producers nor the consumers.

This architecture has several advantages.

1) ExampleApp can maintain its data tuples freely, reordering and delaying them as needed, without any network traffic at all.
2) The API towards the replication system is small and generic, allowing many different solutions.

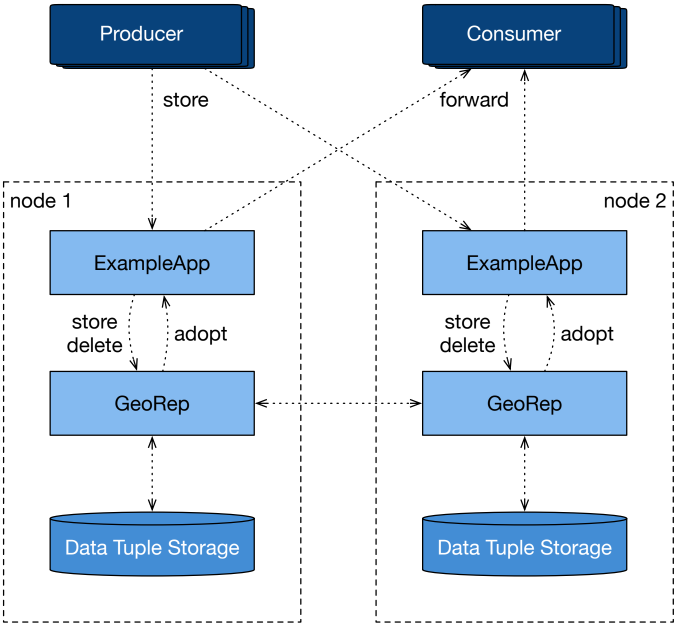

FIGURE 2. Architecture overview for ExampleApp running on two nodes.

3) The replication system does not require any standalone components, which may otherwise add complexity to the installation and maintenance procedures for the full ExampleApp system.

We assume all n nodes receive the same amount of traffic, m messages per second. Using full replication will then lead to the CPU load of $\mathcal{O}(nm)$ on each node, which is undesirable as more system nodes will require a lower m. We therefore need partial replication, giving a load of $\mathcal{O}(fm)$, which is independent of n. We have set a target throughput of 1000 MPS per node.

There are a few potential solutions we need to dismiss for various reasons.

Having the "find the next data tuple" operation in the replication system
If the selection of the next data tuple to forward to the consumer is handled by the replication system, a global consensus must be reached frequently to ensure each data tuple is handled by one single node.

Apache Kafka and other standalone engines
Standalone systems have their advantages, but make the system architecture more complex as they need their own life-cycle management.

Systems requiring modifications in the producers or consumers
For example, ChainReaction [47] uses an API where new data tuples are sent to one node and acknowledged by another. Typically SMS brokers integrate with many different systems developed and maintained by other companies, making any API changes impossible in practice.

II. 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 may fail without data being lost.

A. PROTOCOL DESCRIPTION

Here we describe the activities carried out when GeoRep starts and stops, how data is replicated, and how node failures are handled.

We amend the data tuples with an additional *owners* field, containing 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.

1) Startup

At startup, the application layer in ExampleApp provides its selected value for f to the 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 II-A3 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 [17] 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) Replication

According to our system model described in Section I-A, f nodes are allowed to fail without resulting in data loss. All received data tuples must therefore be replicated to at least f additional nodes before the corresponding acknowledgement can be sent to the producer. We do not need to replicate the data to more than these f nodes, as there is no requirement of keeping all nodes identical. 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 and reachable nodes is less than f, the operation is terminated immediately, and a failure status is returned to the application. If this happens, the producer can send the data to another node.

3) The data tuple is replicated to the f selected nodes.
4) GeoRep returns a condition variable to the application. This variable is signalled when all nodes have responded. The application can therefore be as synchronous as it wants to be, while GeoRep remains asynchronous.

If multiple producers request entries to be replicated sufficiently close in time to the same node, these are all sent together. 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. Fig. 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$_4$.

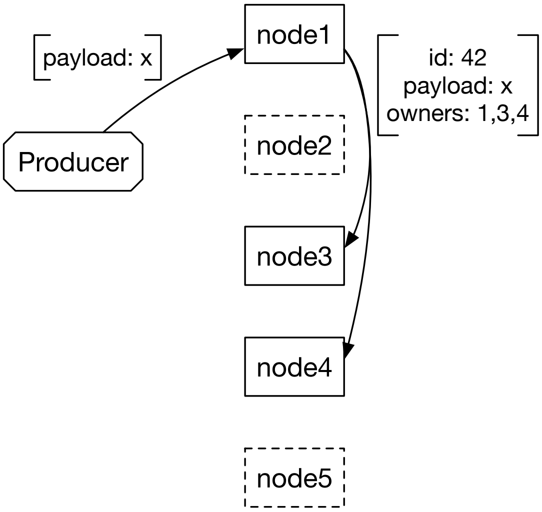

FIGURE 3. Replicate a payload to a subset of size 2 of the 5 known nodes, here nodes 3 and 4. This payload is sent neither to node2 nor node5.

3) Failover

If node$_1$ does not receive anything from node$_2$ for some time, node$_1$ suspects that node$_2$ is down and stops replicating entries to it [48]. It resumes replication to node$_2$ only after node$_2$ has sent proof-of-life by means of new data.

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. This is in contrast to replication protocols using majority quorums, as they only allow the nodes in the majority to accept new data.

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. For these entries, node$_1$ is now the only node allowed to forward them to the consumer. We call this transfer of ownership *adoption*. 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.

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.

B. PEER LIFE CYCLE

Fig. 4 shows the states and transitions used by each node for each one of the other nodes. A 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 the set of reachable servers, 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.

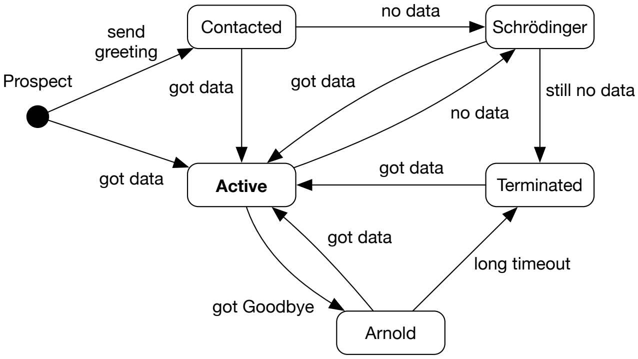

FIGURE 4. The life cycle of each peer.

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 the other nodes which puts it in the *Arnold*[3] state. The failover logic is triggered when moving to the *Terminated* state. To allow partitions to heal, all nodes send occasional heartbeats even to *Terminated* nodes.

C. DATA TUPLE LIFE CYCLE

Fig. 5 and Fig. 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.

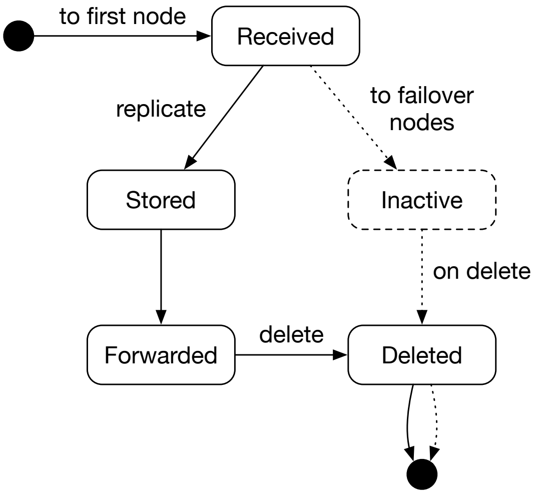

FIGURE 5. The life cycle of each data tuple on the first node.

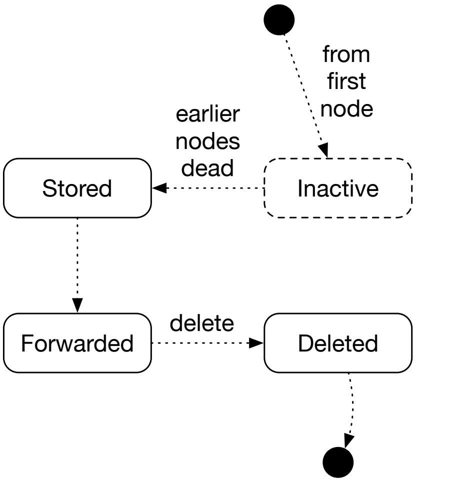

FIGURE 6. The life cycle of a data tuple in case of failover.

First, in Fig. 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 Fig. 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 Fig. 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 Fig. 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.

Fig. 6 illustrates the cases later shown as B and C in Fig. 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 proceeds as above, causing the data tuple to be forwarded and then deleted on any remaining failover nodes. As described in Section III-C, 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.

D. 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 evaluations in Sections IV and V. ZeroMQ[5] is used for the networking layer.

E. 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[6] 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.

III. RELIABILITY ANALYSIS

The design of our protocol has some immediate consequences on its reliability. We will discuss these consequences next, based on the quality model ISO 25010 [49]. This model 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 recov-

[3] It will be back.

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

[5] https://zeromq.org

[6] https://digitalocean.com

erability in terms of how GeoRep handles a lost node was already discussed in Section II-A3.

For the evaluations of the availability and fault tolerance of the proposed protocol, we will use the concepts *yield* and *harvest*, respectively, suggested by Fox and Brewer [50]. In Section III-A 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 III-B, 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 III-C, 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.

A. AVAILABILITY – YIELD

The *yield* [50] 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, as discussed above) correctly functioning nodes. Here we assume that the client knows about all n nodes in the system.

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 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. For example, consider the configuration in Fig. 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 $n = 5$ and $f = 1$ there are $10 + 10 + 5 + 1 = 26$. In contrast, replication protocols which requires a majority of the nodes to work [51] 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 [52], [53] 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 Fig. 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.

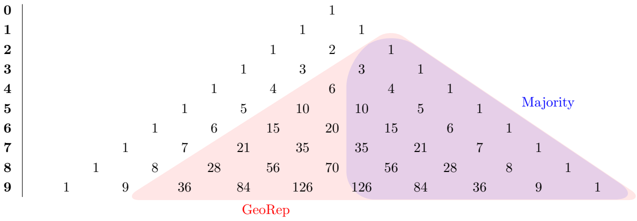

FIGURE 7. Number of node-sets usable by majority replication and GeoRep, for $f = 1$.

The node-sets usable by majority replication are the ones on the right part of Fig. 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 is shown in Equation (2) below. The ones usable by GeoRep are then shown by Equation (3). The number of node-sets usable by majority replication are given by in Equations (4) and (5) 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 (6), which simplifies to Equation (7). As the second term in Equation (8) is a polynomial, the second term in Equation (7) 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 up to twice as high as for other systems.

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

$$\texttt{georep} = 2^n - (n+1) \tag{3}$$

$$\texttt{majority\_odd} = 2^{n-1} \tag{4}$$

$$\texttt{majority\_even} = 2^{n-1} - \binom{n}{n/2} < \texttt{majority\_odd} \tag{5}$$

$$\texttt{ratio} \geq \frac{\texttt{georep}}{\texttt{majority\_odd}} = \frac{2^n - (n+1)}{2^{n-1}} \tag{6}$$

$$= 2 - \frac{n+1}{2^{n-1}} \tag{7}$$

Generally, we get:

$$\texttt{georep} = 2^n - \sum_{k=0}^{f-1} \binom{n}{k} \tag{8}$$

There are multiple strategies to use when selecting which node-set to use, for the situations when there are more than 1 available. The effect the selected strategy has on the system throughput is examined in Section V-D.

B. FAULT TOLERANCE – HARVEST

The *harvest* [50] 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. For GeoRep we 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 Fig. 2). Again, we use concrete values, for e.g., queue and recovery times, in accordance with industry standards. According to Sahoo et al. [54], the typical lifetime of a computer system is in the order of 3–10 years. The actual mean time between failures (MTBF) for a specific system may of course be both lower and higher than this, but in the calculations below we have assumed it to be 3 years. We make no assumptions on the MTBF for other equipment in the data-center, the power grid, etc, even though those are also relevant for a full analysis.

The 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[7]). 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.

In our use case, an embedded system or an IoT device may send a large batch of data tuples faster than they can be fully processed. The resulting queues are typically cleared within a few hours, as the incoming traffic eventually 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 $f = 2$. 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 IV, where only one of the nine test cases required a fourth node to be available to avoid data loss.

For replication protocols using full replication, we get a harvest of $1 - d^n$. As n grows, this of course converges more rapidly towards 1, but at the cost of significantly more data traffic and higher CPU load. We want to emphasize that as there is a possibility that all nodes fail at the same time, the harvest is never exactly 1, so data loss is always possible.

C. FAULT TOLERANCE – DUPLICATION ANALYSIS

We now consider the cases that can occur in the same situation as in Section II-A2, 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 Fig. 8. Neither node$_2$ nor node$_5$ have seen this message, so whether they remain in contact with the other nodes has no effect here. For our SMS gateway application, the consumer here is the mobile network operator handling SMS to the recipient of each particular SMS.

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.

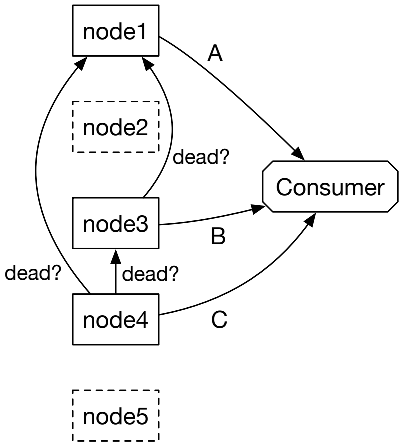

FIGURE 8. Possible duplications.

There is no way for a node to know if any of the other nodes are dead or are unreachable for another reason, e.g., being unusually slow [23], [48]. In case multiple nodes can communicate with the consumer but not with each other, messages could therefore be duplicated. We assume that 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.

IV. 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

[7] This is of course a simplification, but we consider it to be an acceptable compromise in the interest of understandability [55].

tuple should only be adopted by the first node in its *owners* list where all preceding nodes have become unreachable.

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 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.

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

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

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. 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.

Finally, we mapped the test cases listed in Table 2 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.

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

TABLE 2. Relevant tests cases.

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

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

A. EXPERIMENT DESIGN

The critical point for a data tuple is the transfer from *Inactive* to *Stored*, shown in Fig. 6 in Section II-C, 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. This simulates the node having failed.
4) Wait some time to allow the blocked nodes to reach the state *Terminated* in Fig. 4 in Section II-B, triggering the data tuple adoptions.
5) Examine the log files created on each node, to see which node or nodes adopted the data tuple.

B. 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. The factors and variables are summarized in Table 4 for easy overview.

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

TABLE 4. Experiment factors for the failover evaluation.

C. 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. The 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.

D. 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.

No	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**

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

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 III-C, this would however rarely lead to any data duplications.

V. THROUGHPUT EVALUATION

For an evaluation of the proposed protocol primarily focused on quality attributes, we designed a controlled experiment [56]. The overall goal was to evaluate the throughput in a few different configurations.

A. 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 I-A, resulting in realistic experiments. We created a test application which itself created the messages, and discarded them when all tasks described below were completed.

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) [57]. 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.

B. 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.

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

TABLE 6. Experiment factors.

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.

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 possible. Only when using 6 or 7 servers did we use data centers relatively close to each other.

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

TABLE 7. Data centers used for the Geographical cases.

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 Fig. 7 in Section III-A, 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.

2) Constants

We motivate setting the protection f to 1 by recalling the discussion about reliability in Section III-B. 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 $1 - 10^{-16}$.

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.

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 (MPS), according to the sequence of tasks described in Section V-A.

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 Section I, 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.

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.

C. 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.

D. 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.

In a local network, the total system throughput increased with the number of nodes up to $40\,437\,\mathrm{MPS}$ on 7 nodes with 300 clients, shown in Fig. 9. The minimum RTT varied between 143 μs and 420 μs.

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. Fig. 10 shows how the throughput reached $9048\,\mathrm{MPS}$ for 2 nodes and $24\,085\,\mathrm{MPS}$ for 7 nodes.

In Fig. 11 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. We also ran a few tests using $f = 2$. 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 $105\,\mathrm{ms}$. For 7 nodes, the relatively distant nodes in Bangalore and Singapore resulted in an increase to $138\,\mathrm{ms}$. Fig. 12 shows the RTT between a few selected pairs of nodes. For example, the RTT from Toronto (in column 3) is quite low to New York, almost the same to San Francisco and Amsterdam, and quite high to Bangalore. The profiles for nodes geographically close to each other, e.g., New York and Toronto, are notably similar.

Based on Fig. 12, 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 message was received, ignoring nodes with an RTT lower than some predefined limit, say $10\,\mathrm{ms}$. This minimum value ensures messages are always replicated outside of the critical region mentioned in Section I.

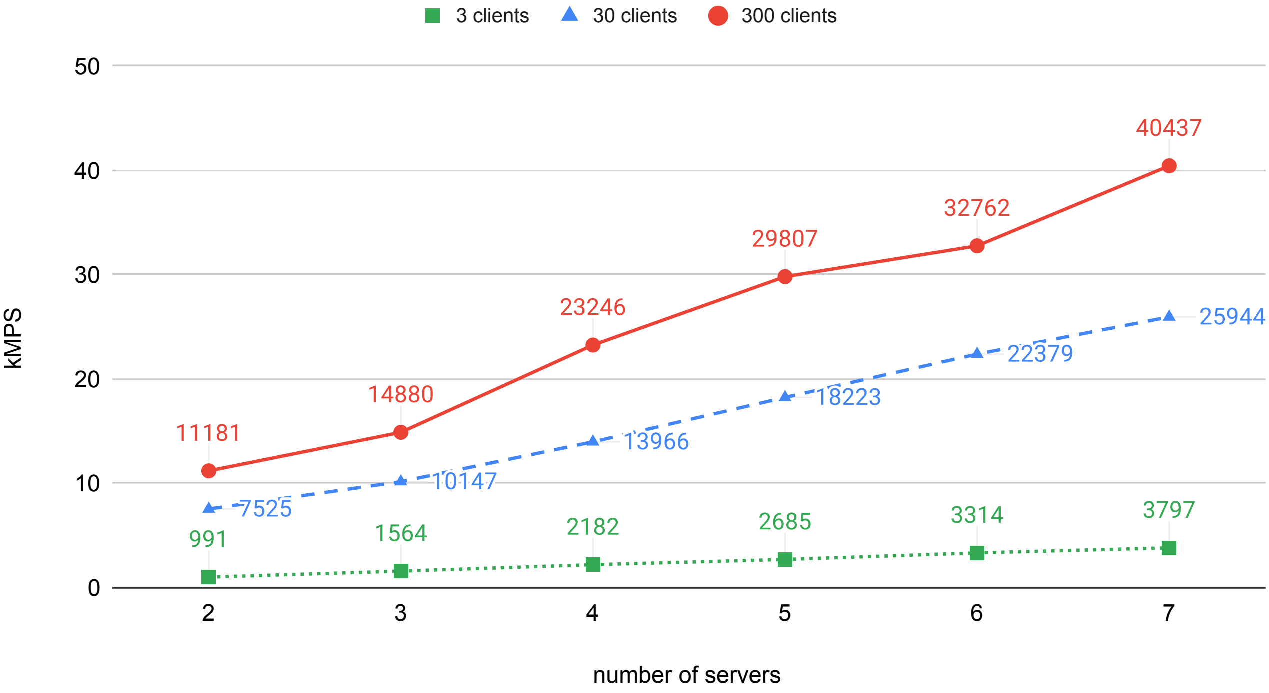

FIGURE 9. System throughput as a function of the number of servers, all running in the same data center.

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 Fig. 13.

E. COMPARATIVE EVALUATION

To get a performance comparison between GeoRep and Paxos, we used the C implementation LibPaxos3[8]. Based on the requirements described in Section I-C, we assumed that a full implementation based on Paxos would need to do at least two operations per message. First, the data would be added to the replicated event list, including the *owners* field described in Section II-A. As only the node first in the *owners* field would be allowed to forward the message, we avoid duplications. When the correct node has forwarded the message, the message id would be replicated again, with a flag marking it as being delivered. We can therefore get the number of messages that could be processed by a Paxos based solution per second, by simply counting the number of events we can submit and divide by 2.

The set of reachable nodes would be stored within the event log as well, providing a consensus on when the failover logic should be activated. There still exists at least one sequence of events where a message may be duplicated, described below. To the best of our knowledge, this situation can not be completely avoided, as any process may crash between promising to do something and then doing it, or between doing something and then informing that it has been done. However, we already stated in Section I-B that a limited number of message duplications, caused by situations like these, are acceptable.

1) A node N finds itself being the owner of a particular message m.
2) Node N sends m.
3) Node N replicates the event that m has been forwarded. Before this event has been sent, N crashes.
4) The remaining nodes discover that N no longer responds, and after a consensus round m is adopted by the

[8] https://bitbucket.org/sciascid/libpaxos

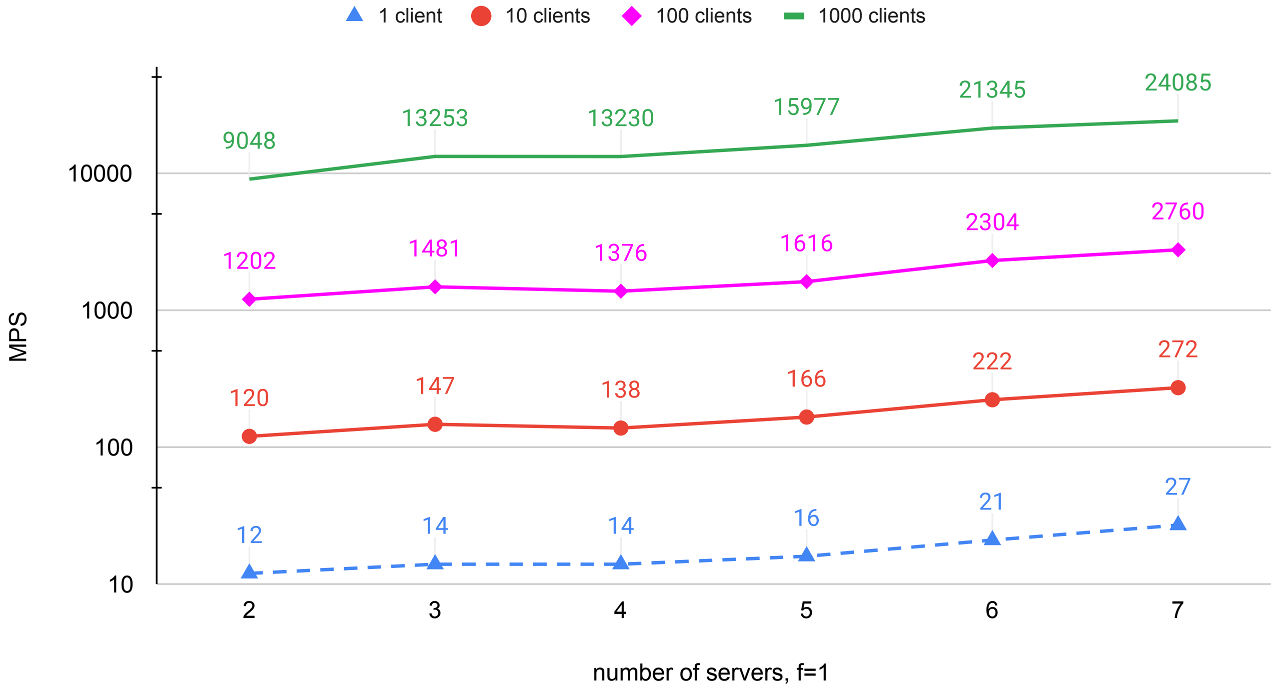

FIGURE 10. System throughput as a function of the number of servers, running in different data centers on multiple continents. Please note that the Y axis is logarithmic, to match the logarithmic increase in the number of clients.

next node in its *owners* list.

We tested the Paxos implementation in the same environments as GeoRep, first with up to 7 servers in the same data-center, and then on up to 7 geo-separated servers. The numbers when all nodes are within the same data-center, in Table 8 on the line marked *Local*, should be compared to the ones for GeoRep in Fig. 9. We see that for 3 nodes Paxos is faster than GeoRep, even when GeoRep has 300 parallel client threads. However, while the system throughput increases when nodes are added in GeoRep, the throughput instead decreases in Paxos. We compare the numbers for the geo-separated configurations to the ones for GeoRep in Fig. 10. Paxos is now more on par with GeoRep for 10 parallel clients. Just as in the previous configuration, the clear performance increase seen for GeoRep is not present with Paxos. The number of clients had no measurable effect in this experiment.

	Number of servers				
	3	**4**	**5**	**6**	**7**
Local/Paxos	22827	13366	16021	13798	9343
Local/GeoRep	14880	23246	29807	32762	40437
Separated/Paxos	756	356	217	211	243
Separated/GeoRep	13253	13230	15977	21345	24085

TABLE 8. LibPaxos3 system throughput, in messages per second (MPS).

The main advantage with a Paxos based solution is that the risk for duplicated messages would be 0, due to the stricter reliance on consensus in Paxos. With up to at least 7 nodes running within the same data-center, we also get at least 1000 MPS per node, our target as specified in Section I-C. Paxos is not as suitable in geo-separated configurations, nor provides the clear scale-up for more servers as seen with GeoRep.

VI. DISCUSSION

In our experiments, the proposed protocol was shown to be 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 Fig. 13 in Section V, the highest recorded throughput for the geo-distributed case was 28 377 MPS when using 7 servers with a minimum RTT of at least 20 ms between each other, or sufficiently far apart to avoid having more than 1 server fail due to a single power or network outage. The independence between the data tuples enables us to reach much more than our target 1000 MPS per node, as long as there are sufficiently many clients.

A. THREATS TO VALIDITY

The identified validity threats are grouped [58], [59] for better overview.

1) Construct

The validity threat "construct" concerns whether the experiment measures the right thing. Differences in hardware,

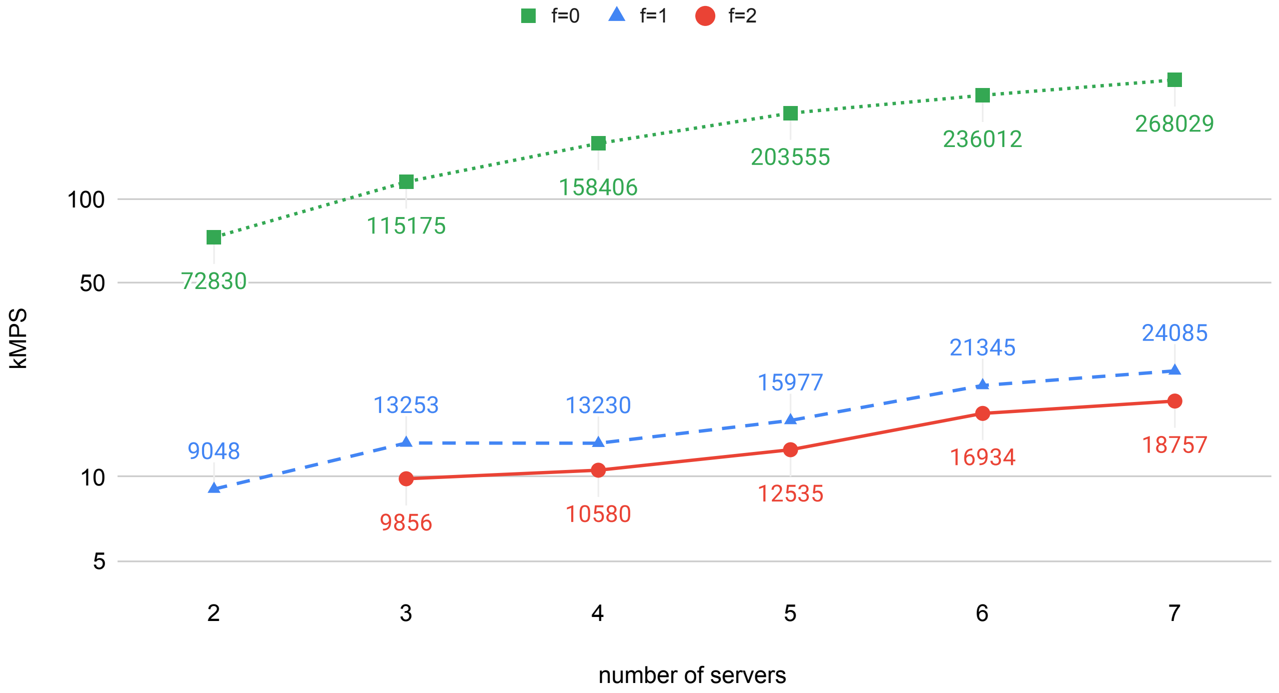

FIGURE 11. System throughput as a function of the number of servers, running in different data centers on multiple continents, when varying f between 0, 1, and 2. The number of clients is 1000.

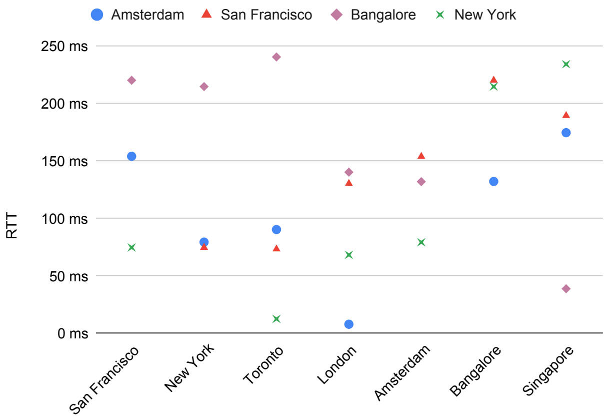

FIGURE 12. Round-trip time (RTT) for various pairs of servers.

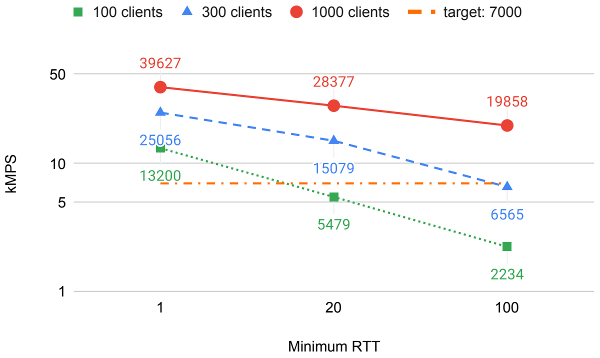

FIGURE 13. System throughput for various minimum RTT limits. In this experiment we use 7 nodes, giving a target throughput of $7 * 1000 = 7000$ MPS.

programming language, the number of clients, servers, and replication groups, as well as selected test scenarios make it difficult to compare absolute numbers to previous work. The failover mechanism uses only local operations, and the rate of this was not measured.

2) Internal

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

3) 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, other than in our embedded systems use case, 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,[9] and each message is important to its recipient. Here, the quality requirements for these systems also mean 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.

VII. RELATED WORK

A. REPLICATION IN PRACTICE

Among others, Helland and Campbell in 2009 [60] and Hellerstein and Alvaro in 2019 [61], argued that shifting the focus from the storage layer up to application semantics may lead to better solutions. In our case, this shift enabled us to not only take advantage of the lower network requirements by partial replication, but also to lower the network usage even further by avoiding the cost of maintaining a total order of the messages. It also made it possible, in case of a network partition, to let other subsets than the one containing the majority of the original nodes continue working, thereby making the system available to the senders in the minority group(s).

B. REPLICATION PROTOCOLS

Other store-and-forward systems are application-to-application message queues, e.g. Apache Kafka [62]. 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. It is quite likely that a Kafka-based solution would perform well in the same environment as used in our tests. It would however not satisfy our "minimal changes to an existing application" requirement from Section I-C.

For systems where a global ordering must be maintained, e.g., fast atomic multicast [36] and white-box atomic multicast [37], the replication protocols are often based on a variant of Paxos [15] or Raft [16]. The Paxos variant Mencius [63] 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. These systems would also require a consensus round among all nodes when each message has been processed and can be deleted, while GeoRep only needs to send this information to the f nodes involved in the replication for that particular message. As is shown in the evaluations of both white-box atomic multicast [37] and Mencius [63], reducing the number of communication steps has a clear and positive effect on the system performance. We do not need the higher consistency these protocols provide, so we can reduce the number of communication steps even further. The experiment in Section V-E showed some of these differences in practice.

Another 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) [64] 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 in our case, as the "find and remove the next data tuple" operation would require a global, synchronous lock. Preliminary tests with such a configuration resulted in a throughput in the order of 1 message per second. Comparing GeoRep with an SQL database in this paper would therefore not be meaningful.

VIII. CONCLUSIONS AND FUTURE WORK

With the purpose of increasing the resilience of a store-and-forward system, we designed a solution based on application semantics instead of lower level storage operations. Several approaches to data replication exist, but we could not find any existing solutions with sufficiently high throughput for geo-separated configurations. Our main contribution in this work is the description and implementation of a new protocol, based on partial replication. When deployed on 7 nodes running on different continents, it provided a total throughput of 24 085 messages per second, almost 100 times higher than a comparable implementation based on Paxos. The primary trade-off is that during a network outage, there is a small risk for message duplication.

Naturally, we welcome replication studies of our protocol. The experiments can be varied along several different dimensions, e.g., a) using other programming languages than

[9] 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.

C, b) using other frameworks than ZeroMQ, c) using a larger number of nodes, d) separating the client applications into separate nodes, and e) 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 IV 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 [4], e.g., hurricanes, floods and tsunamis, it should be possible 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.

REFERENCES

[1] 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.

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

[3] 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.

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

[5] 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.

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

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

[8] Michael Dahlin, Bharat Baddepudi V Chandra, Lei Gao, and Amol Nayate. End-to-end WAN Service Availability. IEEE/ACM transactions on Networking, 11(2):300–313, 2003.

[9] 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.

[10] 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.

[11] 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.

[12] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In Proceedings – SIGCOMM. ACM, 2011.

[13] 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.

[14] 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.

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

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

[17] 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.

[18] 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, Dec 1995.

[19] 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.

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

[21] James P.G. Sterbenz and David Hutchison. ResiliNets Wiki. resilinets.org, 2016 (accessed July 26, 2021).

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

[23] 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.

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

[25] 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.

[26] Coda Hale. You can't sacrifice partition tolerance. https://codahale.com/you-cant-sacrifice-partition-tolerance, 2010 (Retrieved May 2020).

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

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

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

[30] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In Proceedings – Networked Systems Design & Implementation–Volume 3, NSDI. USENIX, 2006.

[31] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. Saturn: A Distributed Metadata Service for Causal Consistency. In Proceedings – European Conference on Computer Systems, 2017.

[32] Paulo Coelho and Fernando Pedone. Geographic State Machine Replication. Technical Report USI-INF-TR-2017-3, Faculty of Informatics Università della Svizzera italiana Lugano, Switzerland, 2017.

[33] Pedro Fouto, João Leitão, and Nuno Preguiça. Practical and Fast Causal Consistent Partial Geo-replication. In Proceedings – International Symposium on Network Computing and Applications (NCA). IEEE, 2018.

[34] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-store: Genuine Partial Replication in Wide Area Networks. In Proceedings – Symposium on Reliable Distributed Systems. IEEE, 2010.

[35] Marcos Kawazoe Aguilera and Robert E Strom. Efficient Atomic Broadcast Using Deterministic Merge. In Proceedings – Symposium on Principles of Distributed Computing, PODC. ACM, 2000.

[36] Paulo R Coelho, Nicolas Schiper, and Fernando Pedone. Fast Atomic Multicast. In Proceedings – International Conference on Dependable Systems and Networks, DSN. IEEE, 2017.

[37] Alexey Gotsman, Anatole Lefort, and Gregory Chockler. White-box Atomic Multicast. In Proceedings – International Conference on Dependable Systems and Networks, DSN. IEEE, 2019.

[38] Rachid Guerraoui and André Schiper. Genuine Atomic Multicast in Asynchronous Distributed Systems. Theoretical Computer Science, 254(1-2):297–316, 2001.

[39] Nicolas Schiper and Fernando Pedone. On the Inherent Cost of Atomic Broadcast and Multicast in Wide Area Networks. In Proceedings – Distributed Computing and Networking, ICDCN. Springer, 2008.

[40] Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. Proceedings – Symposium on Cloud Computing, 2014.

[41] Daniel Brahneborg, Wasif Afzal, Adnan Causevic, and Mats Björkman. Superlinear and bandwidth friendly geo-replication for store-and-forward systems. In 15th International Conference on Software Technologies, July 2020.
[42] Edsger Wybe Dijkstra. Co-operating sequential processes. In Programming languages. Academic Press Inc, 1968.
[43] 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.
[44] 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.
[45] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. Journal of the ACM, 42(1):124–142, 1995.
[46] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. ACM Transactions on Programming Languages and Systems (TOPLAS), 4(3):382–401, Jul 1982.
[47] Sérgio Almeida, João Leitão, and Luís Rodrigues. ChainReaction: a Causal+ Consistent Datastore based on Chain Replication. In Proceedings – The European Professional Society on Computer Systems, EuroSys. ACM, 2013.
[48] 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.
[49] ISO. ISO/IEC 25010. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010, 2021. Accessed 2021-04-06.
[50] Armando Fox and Eric A. Brewer. Harvest, Yield, and Scalable Tolerant Systems. In Proceedings - Workshop on Hot Topics in Operating Systems, HOTOS. IEEE, 1999.
[51] 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.
[52] Akhil Kumar. Hierarchical Quorum Consensus: A New Algorithm for Managing Replicated Data. IEEE Transactions on Computers, 40(9):996–1004, 1991.
[53] Mamoru Maekawa. A sqrt(N) Algorithm for Mutual Exclusion in Decentralized Systems. ACM Transactions on Computer Systems, 3(2):145–159, 1985.
[54] Siva Satyendra Sahoo, Behnaz Ranjbar, and Akash Kumar. Reliability-aware resource management in multi-/many-core systems: A perspective paper. Journal of Low Power Electronics and Applications, 11(1):7, 2021.
[55] Peter A. Alsberg. Research in Network Data Management and Resource sharing, 1976.
[56] Colin Robson and Kieran McCartan. Real world research. John Wiley & Sons, 2016.
[57] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In Proceedings – Symposium on Cloud Computing, SoCC '10, New York, NY, USA, 2010. ACM.
[58] Thomas D Cook and Donald Thomas Campbell. Quasi-experimentation: Design and analysis for field settings, volume 3. Rand McNally, Chicago, 1979.
[59] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. Reporting experiments in software engineering. In Guide to advanced empirical software engineering, pages 201–228. Springer, 2008.
[60] Pat Helland and Dave Campbell. Building on Quicksand. In Proceedings – Conference on Innovative Data Systems Research, CIDR. ACM, 2009.
[61] Joseph M Hellerstein and Peter Alvaro. Keeping calm: when distributed consistency is easy. arXiv preprint arXiv:1901.01930, 2019.
[62] 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.
[63] 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.
[64] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. ACM Computing Surveys, 15(4):287–317, 1983.

DANIEL BRAHNEBORG received a M.Sc degree in computer science from Umeå University in 2015. In 2017 he joined the ITS ESS-H research school as an industrial doctoral student, with a research focus on distributed systems in general and messaging systems in particular. He received a Licentiate degree from Mälardalen University in 2020, on his research on improving the quality attributes of messaging gateways.

He has worked at Braxo AB in Stockholm, Sweden, for the last 20 years. Most of this time has been spent on further development of the company's flagship product, the SMS gateway EMG.

ROMARIC DUVIGNAU received his Ph.D. degree in computer science from the University of Bordeaux (LaBRI), France, in 2015. He is currently an Assistant Professor with the Networks and Systems Division at Chalmers University of Technology.

He was previously affiliated with Aix-Marseille University (LIF) and the University of Bordeaux (LaBRI). His research interests include data stream processing, edge computing, p2p energy trading, and continuous distributed monitoring.

WASIF AFZAL is a Professor in Computer Science and Software Engineering at Mälardalen University, Sweden, where he also co-leads the software testing laboratory research group.

His research interests include software testing, empirical software engineering and decision-support tools for software verification and validation.

PH.D. SAAD MUBEEN is an Associate Professor at Mälardalen University, Sweden. He has previously worked in the vehicle industry as a Senior Software Engineer at Arcticus Systems and as a Consultant for Volvo Construction Equipment, Sweden. He is a Senior Member of IEEE and a Co-chair of the Subcommittee on In-vehicle Embedded Systems within the IEEE IES Technical Committee on Factory Automation. He is co-leading the Heterogeneous systems - hardware software co-design (HERO) research group at Mälardalen University. His research focus is on model- and component-based development of predictable embedded software, modeling and timing analysis of in-vehicle communication, and end-to-end timing analysis of distributed embedded systems. Within this context, he has co-authored over 150 publications in peer-reviewed international journals, conferences and workshops. He has received several awards, including the IEEE Software Best Paper Award in 2017. He is a PC member and referee for several international conferences and journals respectively. He is a guest editor of IEEE Transactions on Industrial Informatics (TII), Elsevier's Journal of Systems Architecture and Microprocessors and Microsystems, ACM SIGBED Review, and Springer's Computing journal. For more information see http://www.es.mdh.se/staff/280-Saad_Mubeen.