Towards a More Reliable Store-and-forward Protocol for Mobile Text Messages

Daniel Brahneborg
Infoflex Connect AB
Stockholm, Sweden
daniel.brahneborg@infoflexconnect.se

Wasif Afzal
Mälardalen University
Västerås, Sweden
wasif.afzal@mdh.se

Adnan Čaušević
Mälardalen University
Västerås, Sweden
adnan.causevic@mdh.se

Mats Björkman
Mälardalen University
Västerås, Sweden
mats.bjorkman@mdh.se

ABSTRACT

Businesses often use mobile text messages (SMS) as a cost effective and universal way of communicating concise information to their customers. Today, these messages are usually sent via SMS brokers, which forward them further to the next stakeholder, typically the various mobile operators, and then the messages eventually reach the intended recipients. Infoflex Connect AB delivers an SMS gateway application to the brokers with the main responsibility of reliable message delivery within set quality thresholds. However, the protocols used for SMS communication are not designed for reliability and thus messages may be lost.

In this position paper we deduce requirements for a new protocol for routing messages through the SMS gateway application running at a set of broker nodes, in order to increase the reliability. The requirements cover important topics for the required communication protocol such as event ordering, message handling and system membership. The specification of such requirements sets the foundation for the forthcoming design and implementation of such a protocol and its evaluation.

CCS CONCEPTS

• **Networks → Network protocols**; **Network protocol design**; **Network reliability**; **Mobile networks**; • **Computer systems organization → Embedded software**; **Reliability**;

KEYWORDS

store-and-forward; replication; SMS

ACM Reference Format:
Daniel Brahneborg, Wasif Afzal, Adnan Čaušević, and Mats Björkman. 2018. Towards a More Reliable Store-and-forward Protocol for Mobile Text Messages. In *ApPLIED'18: Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems, July 27, 2018, Egham, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3231104.3231108

ApPLIED'18, July 27, 2018, Egham, United Kingdom

ACM ISBN 978-1-4503-5775-3/18/07…$$15.00
https://doi.org/10.1145/3231104.3231108

1 INTRODUCTION

Today, mobile text messages (a.k.a. SMS) are often used in business-to-consumer communication, e.g., two-factor authentication and booking reminders. Text messages provide quick and cost effective communication with world-wide coverage, making them a natural choice in many situations. However, the concrete implementation for processing these messages reveals a rather complex system. There are many mobile network operators, even within the same country, and each message must be sent to the correct operator in order to reach the intended recipient. In the 1990s it was sufficient to examine the first few digits in the destination number to find the correct operator. That is no longer enough, as number portability allows customers to keep their number while switching operators. Additionally, the operators use a plethora of communication protocols (e.g. SMPP, UCP, CIMD2, HTTP), of different versions (SMPP 3.4 allows bi-directional traffic while SMPP 3.3 does not) or with unique implementation issues and requirements (primarily phone number formats and character encodings).

Businesses commonly send text messages via services provided by SMS brokers instead of handling this complexity themselves. These brokers charge a fee for providing a single protocol to their clients (i.e. the businesses), and handle both technical and financial communications with the operators. Our context is the software used by these SMS brokers to forward the text messages from their clients to different operators. We call this the SMS gateway application. One such application is the Enterprise Messaging Gateway (EMG) from Infoflex Connect AB.

SMS messaging is based on a store-and-forward architecture, similarly to TCP/IP. Incoming messages are stored in a local queue, and from that queue they are extracted and forwarded as soon as possible. Figure 1 shows the data flow for a message sent from the client to a node run by an SMS broker, and then forwarded to a mobile operator. When the client receives the response at point D, the SMS broker has assumed full responsibility for the message, so the client will delete their copy of the message. A similar response is sent at point G. When the recipient finally has received the message from the operator, an acknowledgement is returned to the sender, indicating whether the message needs to be resent. For

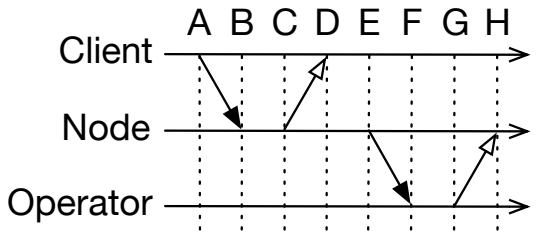

Figure 1: Traffic between clients, nodes, and operators. Filled arrows represent messages, and hollow arrows represent responses.

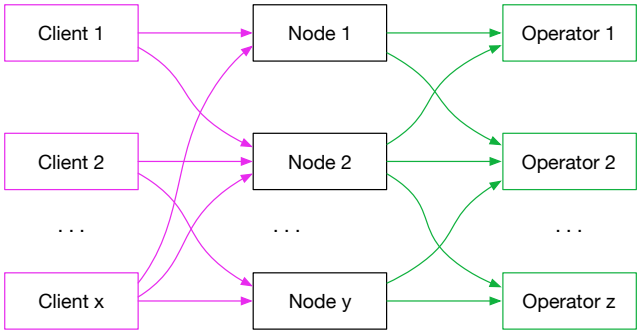

Figure 2: Clients, nodes, and operators.

TCP/IP that acknowledgement is an ACK packet, and for SMS it is a delivery report (usually shortened to "DLR"), albeit with an important difference: the DLR is unreliable. The unreliability of the delivery report is our main issue, as even though the sender can use the response from the SMS broker or operator to know whether the message was accepted, they can not use the DLR to know if the message reached its final destination.

The critical section here is between points C and E in Figure 1, when the message has been received and acknowledged, but not yet forwarded. If the node were to crash in this interval, the message would be lost. To mitigate this risk, the message must be replicated to one or more additional nodes before the response is sent at C. The connectivity to the operator may be (temporarily) broken or too slow in relation to the rate of incoming traffic, resulting in thousands of messages stored on a node, waiting to be forwarded. Without replication, these messages could be lost.

At Infoflex Connect AB, we have experimented with storing messages in SQL and NoSQL databases, but results were discouraging from a performance perspective. Typical throughput was at most a few hundred messages per second, while we could achieve ten times that when using an embedded database, and hundred times that when storing the messages only in memory. There are several reasons for this, apart from the additional I/O. For a system with a single node, the message queues are kept in memory in self-balancing sorted trees to keep the messages in order. For persistence, messages are written and removed using their unique identifier. In a system with multiple nodes, the ordering must be handled by the database, which is not only costly in itself, but also requires synchronization between the database nodes, which in turn must be done for every message.

In this position paper we discuss the requirements for, and the effects of, adding a replication protocol in this store-and-forward context. We do this by first describing our system model (Section 2), followed by the requirements to be met by the protocol in our context (Section 3). Solution candidates are discussed in Section 4, while Section 5 contains a more general review of related work. Section 6 summarizes the requirements and the considered approaches, and concludes the paper.

2 SYSTEM MODEL

We assume the perspective of an SMS broker, using a system consisting of a collection of nodes, some of which are geographically distant. Each node has a unique and constant identifier, can connect to all other nodes via Internet, and may join and leave the system at any time. Furthermore, the nodes are crash-recovery, so they may rejoin after crashing.

Each node runs a store-and-forward application in a configuration as shown in Figure 2. Messages are sent by clients, stored in local queues on one of the nodes managed by an SMS broker, and then forwarded to one of the operators after which they are removed from the queue.

Security issues such as authentication and encryption are handled separately, and there are no byzantine failures [19] with nodes sending arbitrarily erroneous data.

3 REQUIREMENTS

We group our requirements into the following categories: 1) ordering, 2) message handling, 3) system membership, 4) metadata handling, 5) message ownership, and 6) third party effects. Each one is described next. All requirements are considered critical except for the moving of messages between nodes described in Section 3.5.2, as this is just a performance optimization.

3.1 Ordering

The most important requirement in this context, as it has the most far-reaching effects on the solution space, and to the best of our knowledge is the most novel one, is an anti-requirement: the order in which messages are forwarded does not matter. Considering the uses of mobile text message, it is easy to see why. If, e.g., two users request a new authentication code within a few seconds of each other, the exact order of delivery of the codes to the users' mobile phones is not important. For the same reason, the global order of messages received on different nodes, is not important either [24]. However, for the sake of fairness and to ensure liveness, messages should be forwarded in approximately the same order as they were received.

The most commonly used protocols for SMS communication, e.g., SMPP and UCP, support sliding windows with transaction numbers. These numbers are unique values set by the sender and duplicated in the responses, making the reception order of the responses irrelevant. Combined with the insignificance of the message ordering, the protocol can decide to reorder events in different ways if necessary.

3.2 Message handling

Several of the requirements relate to replicating incoming messages and forwarding the messages to an operator or another SMS broker.

3.2.1 Replicate incoming messages. The first step towards high reliability is ensuring the incoming client messages are replicated to the other nodes. The set of nodes required to confirm receiving the messages before the response is sent constitute a quorum [11], which for n nodes in a normal majority system needs a size of at least $\lfloor n/2 \rfloor + 1$. Other quorum systems [34] use other sizes. Thus, we assume this quorum size to be configurable.

3.2.2 Forward messages. Each message received by a node should, ideally, be forwarded to an operator exactly once, regardless of the number of nodes in the system and how many nodes the message has been replicated to. However, this "exactly once" requirement is not absolute, as it is sometimes violated by the usage of sliding windows mentioned in Section 3.1. Sliding windows can lead to duplicated messages if the connection breaks after the message has been received by the operator but before the response comes back to the node (i.e. the interval between points G and H in Figure 1).

In a typical configuration with a single node, the sender uses a window size, denoted w, between 1 and 10. When using UCP, w is limited to 99. With a window size of w, they can send w messages before requiring a response, so the number of messages with an unknown status in case of a broken connection is at most w. The maximum number of duplicated messages, per broken connection, is therefore w. In a system with multiple nodes, the number of duplicated messages should still not exceed w. This would be possible to verify by using a model checker, e.g. Spin[1] or Uppaal[2].

3.3 System membership

The set of nodes in the system should be able to both grow and shrink dynamically.

3.3.1 Accept a new or returning node to the system. A node should be able to join the system at any time, simply by connecting to one of the existing nodes. As described in our system model in Section 2, the nodes are crash-recovery, meaning that nodes can reconnect to the rest of the system, in particular if they were just temporarily unavailable due to a network partition. Even a short-lived network partition may last longer than the lifetime of the messages in the queues, so the difference between "new node" and "returning node" is expected to be minuscule.

3.3.2 Remove a node from the system. We want the current set of nodes in the system to be known to all nodes as soon as possible, in order to know which nodes to replicate messages to. In case the application must be manually stopped,[3] the protocol should propagate the information about a node's impending death.

[1] `http://spinroot.com`
[2] `http://www.uppaal.org`
[3] There are many possible reasons for this, e.g., it should be replaced with a newer version or its system configuration may have changed in a way that requires a full restart.

3.4 Metadata handling

For financial reasons, we must keep track of all received and forwarded messages. We do this by updating the client's credit, and keeping the current state of all messages in a global database.

3.4.1 Manage Credits. Before the client is allowed to send a message, the client's current credit value should be examined. A possible overdraft of this credit is acceptable if it lowers the round-trip time and increases the throughput, but this overdraft must be limited and configurable. When the real cost of forwarding the message is know, this credit value is updated.

3.4.2 Message State Database. For audit purposes, there must exist a mechanism for retrieving the state of all received and forwarded messages. This information does not have to be exact at every point in time, as long as each message is only counted once. Therefore, eventual consistency is sufficient.

3.5 Message Ownership

Each message can only have a single node as its owner, so when the message is replicated to the other nodes it should be stored there in a dormant state, preventing it from being forwarded by those other nodes. This replication is only useful if there also exists a mechanism for changing the owner, making it possible for the replicated messages to be forwarded by another node than the one which received them. Two of the situations where the protocol requires this mechanism are described next.

3.5.1 Adopt messages from a presumed dead node. When an eventually perfect failure detector reports a failed node, the other nodes should quickly take ownership of any messages currently in the queue of this node, so its messages can be forwarded. The delivery requirement is still not "exactly once", but the duplication rate should not change significantly.

3.5.2 Move Messages Between Nodes If Required. For applications such as SMS voting, the message can be sent "upstream" from an operator via one of our nodes, destined for one of the clients. The arrows in Figure 2 just indicates who is connecting to who, the actual network traffic is bidirectional. If the client is not connected to all nodes, a mechanism is needed to automatically move messages to one of the nodes where the client is connected.

Referring to Figure 3, consider the case when Operator z has a message destined for Client 1. The operator does not know about the connections on the left side, so it is sent to a randomly selected node, which in this case could be Node y. It would have been better to send it to Node 2, as it could then be sent directly to Client 1, but we have no control over that. Client 1 is not connected to Node y, so the message must be moved to Node 1 or Node 2 before it can be forwarded to the client. In this scenario the message would need to travel according to the dashed blue lines. While certainly useful, we consider this requirement to be of low priority.

3.6 Third Party Effects

We must also consider the perspective of the clients, operators and SMS brokers connected to our system.

3.6.1 Transparent to third party software. The communication with both clients and operators follow well defined protocols (e.g.

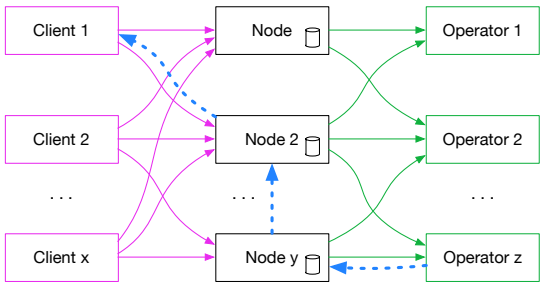

Figure 3: Routing a message back to the client.

SMPP or HTTP), involving thousands of clients and hundreds of operators. Any solution must therefore be completely transparent to these third parties. However, we can assume clients can be given the connection details to multiple nodes and that they can switch freely between them, in particular if the selected node becomes unreachable. HTTP includes response codes to request such a switch, but we can not depend on that being supported by the clients.

As more work needs to be performed per message in order to perform the replication, and additional round-trips between the nodes must be performed before the client can get their response back, the round-trip time seen from the client's perspective (Figure 1, the interval between points A and D) will increase with any replication strategy.

3.6.2 High throughput. The usage of local queues gives mostly independent nodes, which should allow the throughput to remain high while still getting increased reliability from the replication. Previous experiments using a shared database have failed to achieve more than a few hundred messages per second even within the same data center, while local queues reach several thousand. The interval between 1000 and 10 000 operations per second is what is achieved by WanKeeper [1] when it focuses on reading data, as it can take advantage of local processing and data ownership. With equal parts reading and writing in a configuration with geographically distant servers [1], WanKeeper achieves 100 operations per second, while the commonly used ZooKeeper achieves 10. Our requirement, based on discussions with various SMS brokers, is 1000 or more operations per second.

3.7 Limitations

A system fulfilling the described requirements would have some limitations we need to be aware of.

System reliability & performance evaluation
To verify the reliability and performance of the solution, extensive testing and verification is needed.

Network overhead
The replication will lead to increased network traffic, but we can mitigate that in several ways. First, we can replicate multiple messages in the same packet, using the network bandwidth as effectively as possible. Second, we can replicate the messages to just a subset of the nodes. Third, messages that have been forwarded before being replicated to the most distant nodes (with the longest round-trip time), does not have to replicated there at all.

Increased complexity
The complexity of the system would increase, primarily caused by the coordination between the nodes. This could to be addressed by making the solution architecture as simple as possible.

Protocol adoption
Despite being called a protocol, the requirements actually concern an internal architecture. It is not intended for interoperability between separate systems, rather between different broker nodes within a single system.

4 SOLUTION SPACE

This section discusses various available solutions to address the elicited requirements as well as their suitability in our context.

4.1 SQL and NoSQL database clusters

In a multi-node environment a clustered database might be considered for managing the credit values, as described in Section 3.4.1.

Section 3.2 described the low consistency requirements in this domain, which is lower than what either ACID (Atomicity, Consistency, Isolation, and Durability) [12] or BASE (Basically Available, Soft state, Eventual consistency) [10] can offer. Similarly to how an optimistic consistency model can lead to better performance [31], this relaxed delivery guarantee should also enable a more effective architecture.

4.2 Adoption tokens

Our initial idea for solving the changes of message ownership described in Section 3.5 was to use an "adoption token", and passing it around between the nodes. Only the node currently in possession of this token would be allowed to adopt any messages, so there should never exist more than one. If a node dies or there is a network partition, the system may end up with both 0, 1 or 2 such tokens. A lost token is no problem, as that situation is no different from when the system is initially started; simply run a leader election algorithm such as Paxos [18] or Raft [26] to select a node that can create the token. Multiple tokens on the other hand, could lead to thousands of duplicated messages. This can happen if the adoption token ends up in the minority group during a network partition.

Next, we kept the idea of an adoption token, but modified the token passing to use quorum voting instead. To pass the token, the node currently holding it would start an election among all reachable nodes, suggesting the next node. When the new node has seen enough votes, the token is recreated there.

However, this can also lead to considerable message duplications. Consider the following sequence of events, in a system of 3 nodes: A, B and C.

(1) Node A has the token, and starts an election for passing it to B.
(2) Node B wins the election, and assumes ownership of the token.
(3) The network splits, leaving B alone.
(4) Node B realizes it can not reach node A, and adopts all its messages. At this point all current messages on node A are duplicated. Node A is still in the majority group, so it keeps processing its messages normally.

(5) Node C realizes it has not seen the adoption token in some time, and starts an election to create one.
(6) Node C gets a vote from A, giving it a majority.
(7) Node C realizes it can not reach node B, and adopts all its messages. Now all messages on node B are also duplicated.

Depending on the relative executions of the nodes, we may still end up with two adoption tokens and thousands of duplicated messages as an unacceptable consequence.

4.3 Replicated logs

In Section 3.2.2 we concluded that the messages have no linearizability requirement. The high independence between separate text messages, combined with the throughput requirement, make solutions that send all events via a single node to ensure all nodes see the events in the same order both unnecessarily strict and unusable. For this reason, we are not able to use neither Raft [26], Viewstamped Replication [20], nor ZooKeeper [14] for the external network traffic.

The externally visible protocols can not be changed, as discussed in Section 3.6.1. This prevents solutions such as Raft [26] and Chubby [4].

5 RELATED WORK

This section discusses the existing products and academic work focused on data replication. In Section 5.1 we describe solutions that are implemented, evaluated and in use, while in Section 5.2 we focus more on theoretical results. In short, we have not been able to find a solution that can take advantage of our low ordering requirements, thus reusing existing solutions would potentially result in suboptimal performance in our context.

5.1 State of practice

In many applications, persisting data to survive application crashes is done in a database. This can be either a classical SQL database such as MySQL or Oracle, or a more modern NoSQL database such as MongoDB or Cassandra.

For message queues, RabbitMQ and Apache Kafka are common solutions. Kafka is a better choice if events need to be persisted to disk and replayed any number of times by clients, while RabbitMQ supports multiple protocols which is good for interoperability, and a possibility to set the time to live (TTL) of messages [5]. However, the disk usage by Kafka can be extensive, and RabbitMQ does not scale well when the queue sizes increase. EMG needs both support for message TTL and large queues.

5.1.1 Message Queues. There are plenty of message queue products of varying complexity (e.g. RabbitMQ[4], Qpid[5], and IBM WebSphere), implementing various well documented protocols (e.g. Java JMS, AMQP (Advanced Message Queuing Protocol)[6], and ZeroMQ [13]), so a program that only forwards messages is trivial. However, systems using these solutions, such as SDQS [38], Andes [35] and EQS [32], are all very strict regarding the ordering of the messages. A common solution for synchronization between nodes to ensure this ordering is ZooKeeper [14], which works fine for local clusters within a single data center, but in a geo-distributed environment stays below 10 transactions per second [1].

Previously, the message queue system Apache Kafka[7] [15] used ZooKeeper [14] for coordination for each entry, making it unusable in our context. In Kafka, enqueued messages can be delivered to one of many nodes, which is exactly what we needed. Messages can be separated into different topics, with ordering only being guaranteed within each topic. This relaxed ordering is still too strict for us.

Closely related to message queues are publish/subscribe systems. Both message queues and publish/subscribe systems allow "space decoupling", i.e. the sender and the recipient need not be aware of each other, "time decoupling", i.e. the message is sent at one point in time and delivered at another, and "synchronization decoupling", i.e. the sender does not have to wait until the message has been delivered to the recipient [8]. This matches our requirements well. The two main ways they differ from what is needed for EMG is that they also assume a "one-to-many" delivery strategy, and that filtering must be used to avoid sending all messages to all recipients.

5.1.2 Storage. An important difference between our message queues and a generic storage system, is that we have no externally initiated reads. Once a message has been received, only the system itself needs to know where it is stored. There will be no requests from other applications to fetch the message. Therefore, all effort spent in handling such operations, are for our purposes wasted. As an example, the "Saturn" system [3] is described as a way to "enforce causal consistency when accessing replicated data", where the critical word here is "accessing". Other systems such as ChainReaction [2], Orbe [6], GentleRain [7], COPS [22] and SwiftCloud [37] use different mechanisms to achieve this accessibility, e.g. vector clocks [17], physical clocks, and caches. MeteorShower [21] explicitly addresses the delays caused by geographically separated servers.

Rarely, if ever, is the lifetime of the stored objects addressed. Typically, objects are created by one of the nodes, and occasionally updated by the same or another node. Most of the operations are then read accesses. For example, the system OCC [31] uses a workload "with a 32:1 GET:PUT ratio". For a message queue, the situation is different. For such a system, there are not only no read accesses at all, we also know that all objects will be removed after usually just a few seconds. To the best of our knowledge, no existing storage mechanism considers this factor as a way of minimizing the amount of data needed for the replication.

5.2 State of the art

The most active areas of interest to us concern commutative functions and leader election algorithms. Commutative functions address the fact that strict ordering of events is unnecessary, and leader election is the base concept when achieving consensus between multiple nodes.

5.2.1 Commutative Functions. Whether reality is ordered has been discussed for some time, at least from 1993 by Queinnec and Padiou [29] regarding flight plans. If this ordering can be ignored, it

[4] https://www.rabbitmq.com
[5] https://qpid.apache.org
[6] http://www.amqp.org
[7] http://kafka.apache.org

is also possible to attain higher availability of data in an unreliable network [9] and lower the number of network round-trips [27]. For example, using the CRDT "PN-counter" [30], it is possible to implement updates and limit checks of user credits in a distributed environment from Section 3.4.1, without any network round-trips.

Using commutative functions in order to get lower response times has been known since 1988 [16] if not earlier. Shapiro et al. [30] described data types based on these commutative functions, calling them "Convergent or Commutative Replicated Data Types (CRDTs)". Zawirski then described some lower limits of the amount of metadata needed for the replication of some of these data types [36], both in terms of the number of nodes and the number of updates.

5.2.2 Leader Election. There are several algorithms for reaching consensus among a set of communicating nodes [33], each one in multiple variants. The most commonly known one is probably Paxos [18]. In recent years, Raft [26] is also used. Both of them, as well as Viewstamped Replication [20, 25] and the lock service Chubby [4], require support from the third party applications that connect to the replicating system. This is in conflict with the requirement in Section 3.6.1 of being transparent to those third parties. For an internal protocol between EMG nodes only, any of these can of course be used.

To avoid overloading a single node with a leadership role through which all requests much pass, the mechanism used by Paxos, Raft and others, some alternatives exist. Mencius [23] is derived from Paxos, but lets the leader role rotate periodically. AllConcur [28] goes further, being entirely leaderless. With AllConcur, each message is forwarded several times between different pairs of nodes, resulting in more network traffic than the leader-based methods.

6 SUMMARY

6.1 Approaches

Table 1 summarizes the requirements and approaches to be considered for fulfilling them. Essentially, databases are too strict on ordering, and replicated logs are unsuitable in a geo-distributed environment as they rely on all events being serialized by a single node.

Requirement	Approach	Problems
System membership	Per node[8]	None known
Message storage	Database	Strict ordering
	Message queue	Strict ordering
Message state	Database	Round-trip times
	Replicated log	Single node
Message ownership	Replicated log	Single node
Client credits	PN-counter	Possible overdrafts

Table 1: Considered approaches for each set of requirements, and expected new problems.

[8]Information about new nodes are broadcast among all nodes, but each node maintains its own list of active peers.

6.2 Conclusions

The communication protocols for mobile text messages are unreliable, as there is no dependable end-to-end acknowledgement packet. There is, on the other hand, no strict ordering requirements in this domain, allowing the use of more effective solutions than off-the-shelf message queue products. We can not avoid the inevitable round-trip time between data centers, which may very well be geographically distant, but by replicating messages from multiple clients in the same update, the total system throughput should be satisfactory. As contributions, this position paper sets the requirements for a reliable communication protocol for SMS in place, and reviews the state of the art as well as practice for considered solutions. While partial solutions suitable in our context are available, a complete solution satisfying the requirements specified in this paper would require a new bespoke protocol that can effectively take advantage of the possibility of event reordering and short lifetime of the messages.

ACKNOWLEDGMENTS

This work was sponsored by The Knowledge Foundation industrial PhD school ITS ESS-H, 20160139 (TestMine), 20130085 (TOCSYC) and Infoflex Connect AB.

REFERENCES

[1] A. Ailijiang, A. Charapko, M. Demirbas, B. O. Turkkan, and T. Kosar. Efficient distributed coordination at WAN-scale. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2017.
[2] S. Almeida, J. Leitão, and L. Rodrigues. Chainreaction: a causal+ consistent datastore based on chain replication. In *Proceedings of The European Professional Society on Computer Systems (EuroSys)*, pages 85–98. ACM, 2013.
[3] M. Bravo, L. Rodrigues, and P. Van Roy. Towards a Scalable, Distributed Metadata Service for Causal Consistency under Partial Geo-replication. *Proceedings of the Doctoral Symposium of the International Middleware Conference (Middleware)*, pages 1–4, 2015.
[4] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–350. USENIX Association, 2006.
[5] P. Dobbelaere and K. S. Esmaili. Kafka versus RabbitMQ. In *Proceedings of the Distributed Event-Based Systems (DEBS)*, 2017.
[6] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the Symposium on Cloud Computing (SOCC)*, page 11. ACM, 2013.
[7] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. *Proc. of the ACM Symposium on Cloud Computing (SOCC)*, pages 1–13, 2014.
[8] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.
[9] M. J. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, 1982.
[10] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. *ACM SIGOPS Operating Systems Review*, 31(5), 1997.
[11] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*. ACM, 1979.
[12] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.
[13] P. Hintjens. *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.
[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 2010.
[15] J. Kreps, N. Narkhede, and J. Rao. Kafka: a Distributed Messaging System for Log Processing. In *Proceedings of the SIGMOD Workshop on Networking Meets Databases (NetDB)*, Athens, Greece, 2011.
[16] A. Kumar and M. Stonebraker. Semantics based transaction management techniques for replicated data. *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 117–125, 1988.
[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[18] L. Lamport. The part-time parliament. *ACM Trans. on Computer Systems*, 16(2):133–169, 1998.
[19] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. on Programming Languages and Systems*, 4(3):382–401, 1982.
[20] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, Massachusetts Institute of Technology, 2012.
[21] Y. Liu, X. Guan, V. Vlassov, and S. Haridi. MeteorShower: Minimizing Request Latency for Majority Quorum-Based Data Consistency Algorithms in Multiple Data Centers. *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 57–67, 2017.
[22] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. *Proceedings of the Symposium on Operating Systems Principles (SOPS)*, pages 1–16, 2011.
[23] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, pages 369–384, 2008.
[24] C. S. Meiklejohn. A certain tendency of the database community. In *Companion to the International Conference on the Art, Science and Engineering of Programming*. ACM, 2017.
[25] B. M. Oki and B. H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, volume 62, pages 8–17, 1988.
[26] D. Ongaro. *Consensus: Bridging Theory And Practice*. PhD thesis, Stanford University, 2014.
[27] S. J. Park and J. Ousterhout. Exploiting commutativity for practical fast replication. 2017.
[28] M. Poke, T. Hoefler, and C. W. Glass. AllConcur: Leaderless Concurrent Atomic Broadcast Marius. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 205–218, 2017.
[29] P. Queinnec and G. Padiou. Flight plan management in a distributed air traffic control system. *Proceedings of the International Symposium on Autonomous Decentralized Systems (ISAD)*, 1993.
[30] M. Shapiro, N. Pregui, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, 2011.
[31] K. Spirovska, D. Didona, and W. Zwaenepoel. Optimistic Causal Consistency for Geo-Replicated Key-Value Stores. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 2626–2629, 2017.
[32] N. L. Tran, S. Skhiri, and E. Zimányi. EQS: An elastic and scalable message queue for the cloud. In *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science, (CloudCom)*, pages 391–398, 2011.
[33] J. Turek and D. Shasha. The many faces of consensus in distributed systems. *Computer*, 25(6):8–17, 1992.
[34] M. Vukolić. *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012.
[35] C. Wickramarachchi, S. Perera, S. Jayasinghe, and S. Weerawarana. Andes: A highly scalable persistent messaging system. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, pages 504–511, 2012.
[36] M. Zawirski. *Dependable Eventual Consistency with Replicated Data Types*. PhD thesis, Université Pierre et Marie Curie - Paris, 2015.
[37] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write Fast, Read in the Past: Casual Consistency for Client-side Applications. Technical report, 2015.
[38] Z. Zhang, Y. Wang, H. Chen, M. Kim, J. M. Xu, and H. Lei. A cloud queuing service with strong consistency and high availability. *IBM Journal of Research and Development*, 55(6):10:1–10:12, 2011.