Scalable and Robust Algorithms for Cloud Storage and for Sensor Networks

Ittay Eyal

Scalable and Robust Algorithms for Cloud Storage and for Sensor Networks

Research thesis

In Partial Fulfillment of the
Requirement for the
Degree of Doctor of Philosophy

Ittay Eyal

Submitted to the Senate of
the Technion - Israel Institute of Technology

Sivan 5773 — Haifa — May 2013

To my grandparents, Lize, Ya'akov, Bracha and Ephraim.

The research thesis was done under the supervision of Prof. Raphael Rom and Prof. Idit Keidar in the Department of Electrical Engineering.

THE GENEROUS FINANCIAL HELP OF THE TECHNION — ISRAEL INSITUTE OF TECHNOLOGY AND THE HASSO PLATTNER INSTITUTE FOR SOFTWARE SYSTEMS ENGINEERING (HPI) IS GRATEFULLY ACKNOWLEDGED

Acknowledgments

I would like to extend my gratitude to the following people.

To my advisors, Idit and Raphi. Your knowledge, professionalism, and passion for research are an inspiration. You have taught me a lot on research and beyond. Thanks for a combination of great freedom, close guidance and support. It was a privilege to work with you, both professionally and personally.

To Christian Cachin, Robert Haas, Alessandro Sorniotti, Marko Vukolic, Ido Zachevsky, Cristina Basescu, Flavio Junqueira, Ken Birman, and Robbert Van Renesse for fruitful collaborations.

To Muli Ben-Yehuda, Isaac Keslassy, Yoram Moses, and Eddie Bortnikov for their insights and advice.

To my friends in the department for advice, help, and many coffee breaks. Special thanks to Dima, Eddie, Alex, Nathaniel, Yaniv, Eyal, Stacy, Zvika, Liat, Oved, Elad, and Ori.

To my parents Michael and Tzippy, for providing me with the tools to overcome all difficulties during my studies (and in general), and to my brother Ophir, for his support.

Special thanks to my wife for her encouragement, and for helping me balance research and life. This couldn't have happened without your support. And to my son Nadav for helping, mostly by going to sleep on time.

List of Publications

I. Eyal, F. Junqueira., I. Keidar, Thinner Clouds with Preallocation. The 5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'13).

I. Eyal, I. Keidar, S. Patterson, and R. Rom. Global Estimation with Local Communication. Poster in the 5th International Systems and Storage Conference (SYSTOR'12).

C. Basescu, C. Cachin, I. Eyal, R. Haas, Alessandro Sorniotti, M. Vukolic and Ido Zachevsky. Robust Data Sharing with Key-Value Stores. The 42nd annual IEEE/IFIP international conference on Dependable Systems and Networks (DSN'12).

I. Eyal, I. Keidar, and R. Rom: LiMoSense Live Monitoring in Dynamic Sensor Networks, The 7th International Symposium on Algorithms for Sensor Systems, Wireless Ad Hoc Networks and Autonomous Mobile Entities (ALGOSENSOR'11).

I. Eyal, I. Keidar, and R. Rom: Distributed Data Clustering in Sensor Networks. Distributed Computing 24:5, pages 207-222, November 2011, the 29th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'10), and the 5th workshop on Hot Topics in System Dependability (HotDep '09).

Contents

List of Figures

Abstract

Fast advancements in the production and construction of computer systems have led to the proliferation of highly distributed systems, at a scale that was unimaginable not many years ago. We address here two types of such systems — sensor networks and cloud storage. We construct scalable and robust distributed algorithms for these environments, prove their correctness, and analyze their behavior through simulation.

To perform monitoring of large environments, we can expect to see in years to come sensor networks with thousands of light-weight nodes monitoring conditions like seismic activity, humidity or temperature. Each of these nodes is comprised of a sensor, a wireless communication module to connect with close-by nodes, a processing unit and some storage. The nature of these widely-spread networks prohibits a centralized solution in which the raw monitored data is accumulated at a single location. Fortunately, often the raw data is not necessary. Rather, an aggregate that can be computed inside the network.

In the first part of this work, we address two aggregation challenges in the field of sensor networks. First, we present LiMoSense, a fault-tolerant live monitoring algorithm for dynamic sensor networks. This is the first asynchronous robust average aggregation algorithm that performs live monitoring, i.e., it constantly obtains a timely and accurate picture of dynamically changing data. Second, we address the distributed clustering problem, where the computed aggregate is a clustering of the sensor data, i.e., the goal is to partition these values into multiple clusters, and describe each cluster concisely. We present a generic algorithm that solves the distributed clustering problem and may be implemented in various topologies, using different clustering types.

In the second part of this work, we address two challenges relating to consistency in large scale cloud storage systems. Advances in datacenter technologies are leading users, both consumers and large companies, to store large volumes of data in managed services, where storage is offered as a service — cloud storage. The users of such systems have increasing expectations

of both efficiency and reliability, leading to various challenges in implementing these data stores.

First, we observe that a single storage provider, large as it may be, might fail, either losing data or just being temporarily unavailable. We provide a storage algorithm that achieves reliable storage using multiple real-world production storage services. A key-value store (KVS) offers functions for storing and retrieving values associated with unique keys. KVSs have become the most popular way to access Internet-scale "cloud" storage systems. We present an efficient wait-free algorithm that emulates multi-reader multi-writer storage from a set of potentially faulty KVS replicas in an asynchronous environment.

Second, we introduce ACID-RAIN: ACID[1] transactions in a Resilient Archive with Independent Nodes. ACID-RAIN is a novel architecture for efficiently implementing transactions in a distributed data store. ACID-RAIN uses logs in a novel way, limiting reliability to a single tier of the system: a large and scalable set of independent nodes form an outer layer that caches the data, backed by a set of independent reliable log services. If concurrent transactions conflict with one another, one or more of them must abort. ACID-RAIN avoids such conflicts by using prediction to order transactions before they take actions that would lead to an abort.

[1] "ACID transactions", stands for Atomic, Consistent, Isolated and Durable transactions. These are commonly known in the distributed systems literature as atomic transactions with persistent storage.

Chapter 1

Introduction

The field of distributed computing has been dealing for a long while with challenges of designing robust and scalable algorithms for different purposes, from data acquisition, through processing, to storage. Recently, advancements in the production and construction of large scale networked systems has made such algorithms a practical necessity. The design of a distributed algorithm has to follow the following principles.

Robustness A system with thousands of nodes naturally suffers from occasional node failures, and consequently new node additions. Similarly, message loss and link failure are unavoidable in large systems, and the system must be resilient to all of these.

Asynchrony Synchronous algorithms make assumptions on the maximal delay in the system. This allows a node to deduce, by waiting this delay, that a message it has sent was received, or that if it does not receive a message, that message was never sent, or was lost. Using a conservative assumption (large delay) causes the system to progress slowly, as nodes have to wait long to make these deduction. However, choosing a shorter delay is not an option — in a large system occasional occurrences of long delays are the norm. A single node may suffer from a temporal malfunction slowing it down considerably, and messages may be delayed in a multi-hop network. A system should therefore avoid these assumptions, and use an asynchronous algorithm that makes no assumptions on message delay.

Scalability To be truly scalable, a system must refrain from depending on any single element in its critical path, since this element will become a bottleneck.

This dissertation describes four scalable algorithms that follow these principles. They are all asynchronous, allow for node and link failure, and depend on no single element for operation.

1.0.0..1 Aggregation in sensor networks In Part I we address two aggregation challenges in the field of sensor networks. Sensor networks are ad-hoc networks of light-weight nodes monitoring environmental conditions, each communicating via radio with close-by nodes. We briefly introduce aggregation in sensor networks in Chapter 2.

In Chapter 3, we present LiMoSense, a fault-tolerant live monitoring algorithm for dynamic sensor networks. This is the first asynchronous robust average aggregation algorithm that performs live monitoring, i.e., it constantly obtains a timely and accurate average of dynamically changing data.

However, more elaborate data summaries are sometimes required. For example, outliers caused by erroneous samples may divert the average, or the data may be partitioned into several clusters with different averages. In Chapter 4 we address the distributed clustering problem, where the computed aggregate is a clustering of the sensor data, i.e., the goal is to partition the values into multiple clusters, and describe each cluster concisely. We present a generic algorithm that solves the distributed clustering problem and may be implemented in various topologies, using different clustering types. As an example, we implement Gaussian mixture clustering and evaluate its accuracy and convergence speed through simulation.

1.0.0..2 Consistency in Cloud Storage In Part II, we present two algorithms that deal with distributed cloud storage. Cloud storage allows many users to concurrently access replicated data stored on multiple machines residing in datacenters. We provide the relevant background in Chapter 5.

Cloud storage providers typically offer several storage interfaces, each implementing different interfaces. However, practically all providers offer a key-value store (KVS), with functions for storing and retrieving values associated with unique keys. Cloud storage providers promise high availability, but even the largest of them sometimes fail, either losing data or just being temporarily unavailable.

In Chapter 6 we present a storage algorithm that provides reliable storage using multiple real-world production storage services. We present an efficient wait-free algorithm that provides multi-reader multi-writer storage from a set of potentially faulty KVS replicas in an asynchronous environment.

In cloud-scale data centers, it is common to shard data across many nodes, each maintaining a small subset of the data. Although ACID transactions are

desirable, architects of such systems often tradeoff efficiency for consistency, and do not support them. In Chapter 7 we present a novel architecture for support of low-latency high-throughput ACID transactions in a Resilient Archive with Independent Nodes (ACID-RAIN). ACID-RAIN uses logs in a novel way, limiting the requirement for reliability to a single scalable tier: A set of independent highly-available logs is accessed by large set of independent fault-prone nodes that caches the sharded data. This structure allows for rapid data access through the cache, and simple and fast restoration in case of node failure. ACID-RAIN dramatically reduces concurrency conflicts by using prediction to order transactions before they take actions that would lead to an abort. We compare ACID-RAIN with contemporary architectures for the support of ACID transactions, and demonstrate effective contention handling and linear scalability, whereas other approaches reach a bottleneck.

Part I

Aggregation in Sensor Networks

Chapter 2

Background

To perform monitoring of large environments, we can expect to see in years to come sensor networks with thousands of light-weight nodes monitoring conditions like seismic activity, humidity or temperature [10, 105]. Each of these nodes is comprised of a sensor, a wireless communication module to connect with close-by nodes, a processing unit and some storage. The nature of these widely spread networks prohibits a centralized approach in which the raw monitored data is accumulated at a single location. Specifically, all sensors cannot directly communicate with a central unit.

Fortunately, often the raw data itself is not the goal. Rather, an *aggregate* that can be computed *inside the network*, such as the sum or average of sensor reads, is of interest. For example, when measuring rainfall, one is interested only in the total amount of rain, and not in the individual reads at each of the sensors. Similarly, one may be interested in the average humidity or temperature rather than minor local irregularities.

Several works have dealt with the single-shot version of this problem [72, 21, 87, 83]. In the single-shot case, each sensor takes a single sample, and then the nodes communicate and learn the average of these read-values. However, to perform *live monitoring*, we need to constantly obtain a timely and accurate picture of the ever-changing data. Running multiple iterations of a single-shot algorithm is either inefficient (starting iterations with high frequency) or inaccurate (starting them with a low frequency, or not allowing the runs to converge). In Chapter 3 we tackle the problem of live monitoring in a dynamic sensor network. This problem is particularly challenging due to the dynamic nature of sensor networks, where nodes may fail and may be added on the fly (churn), and the network topology may change due to battery decay or weather change.

While average aggregation is useful in many scenarios, there are applications that call for a more elaborate summarization of the sensors' readings. In

the *distributed clustering problem*, numerous interconnected nodes compute a *clustering* of their data, i.e., partition these values into multiple clusters, and describe each cluster concisely.

We present in Chapter 4 a *generic algorithm* that solves the distributed clustering problem and may be used in any connected topology, using different clustering types. For example, the generic algorithm can be instantiated to cluster values according to distance, targeting the same problem as the famous k-means clustering algorithm. Since the distance criterion is often not sufficient to provide good clustering results, we present an instantiation of the generic algorithm that describes the values as a *Gaussian Mixture* (a set of weighted normal distributions), and uses machine learning tools for clustering decisions. Simulations show the robustness, speed and scalability of this algorithm. We prove that any implementation of the generic algorithm converges over any connected topology, clustering criterion and cluster representation, in fully asynchronous settings.

Chapter 3

LiMoSense

The subject of environmental monitoring is gaining increasing interest in recent years. *Live monitoring* is necessary for research, and it is critical for protecting the environment by quickly discovering fire outbreaks in distant areas, cutting off electricity in the event of an earthquake, etc. In order to perform these tasks, it is necessary to perform constant measurements in wide areas, and collect this data quickly.

However, most previous solutions have focused on a static (single-shot) version of the problem, where the average of a single input-set is calculated [72, 21, 87, 83]. Though it is in principle possible to perform live monitoring using multiple iterations of such algorithms, this approach is not adequate, due to the inherent tradeoff it induces between accuracy and speed of detection. For further details on previous work, see Section 3.1. In this chapter we tackle the problem of live monitoring in a dynamic sensor network. This problem is particularly challenging due to the dynamic nature of sensor networks, where nodes may fail and may be added on the fly (*churn*), and the network topology may change due to battery decay or weather change. The formal model and problem definition appear in Section 3.2.

In Section 3.3 we present our new **Li**ve **Mo**nitoring for **Sens**or networks algorithm, LiMoSense. Our algorithm computes the average over a dynamically changing collection of sensor reads. The algorithm has each node calculate an estimate of the average, which continuously converges to the current average. The space complexity at each node is linear in the number of its neighbors, and message complexity is that of the sensed values plus a constant. At its core, LiMoSense employs gossip-based aggregation [72, 87], with a new approach to accommodate data changes while the aggregation is ongoing. This is tricky, because when a sensor read value changes, its old value should be removed from the system after it has propagated to other nodes. LiMoSense further employs a new technique to accommodate message loss,

failures, and dynamic network behavior in asynchronous settings. This is again difficult, since a node cannot know whether a previous message it had sent over a faulty link has arrived or not.

In Section 3.4, we prove the correctness of the algorithm, showing that once the network stabilizes, in the sense that no more value or topology changes occur, LiMoSense eventually converges to the correct average, despite message loss. Since the algorithm cannot tell if and when the network has stabilized, it constantly converges to the current average.

To demonstrate the effectiveness of LiMoSense in various dynamic scenarios, we present in Section 3.5 results of extensive simulations, showing its quick reaction to dynamic data read changes and fault tolerance. In order to preserve energy, communication rates may be decreased, and nodes may switch to sleep mode for limited periods. These issues are outside the scope of this work.

In summary, this chapter makes the following contributions:

1. It presents LiMoSense, a live monitoring algorithm for highly dynamic and error-prone environments.

2. It proves correctness of the algorithm, namely robustness and eventual convergence.

3. It shows by simulation that LiMoSense converges exponentially fast in well connected topologies, and demonstrates its efficiency and fault-tolerance in dynamic scenarios.

A preliminary version of the work presented in this chapter appears in the proceedings of the 7$^{\text{th}}$ International Symposium on Algorithms for Sensor Systems, Wireless Ad Hoc Networks and Autonomous Mobile Entities (ALGOSENSOR'11) [48].

3.1 Related Work

To gather information in a sensor network, one typically relies on in-network *aggregation* of sensor reads. The vast majority of the literature on aggregation has focused on obtaining a *single* summary of sensed data, assuming these reads do not change while the aggregation protocol is running [83, 72, 21, 87].

For obtaining a single aggregate, two main approaches were employed. The first is hierarchical gathering to a single base station [83]. The hierarchical method incurs considerable resource waste for tree maintenance, and results in aggregation errors in dynamic environments, as shown in [64].

The second approach is gossip-based aggregation at all nodes. To avoid counting the same data multiple times, Nath et al. [88] employ order and duplicate insensitive (ODI) functions to aggregate inputs in the face of message loss and a dynamic topology. However, these functions do not support dynamic inputs or node failures. Moreover, due to the nature of the ODI functions used, the algorithms' accuracy is inherently limited – they do not converge to an accurate value [50].

An alternative approach to gossip-based aggregation is presented by Kempe et al. [72]. They introduce Push-Sum, an average aggregation algorithm, and bound its convergence rate, showing that it converges exponentially fast in fully connected networks where nodes operate in lock-step. Fangani and Zampieri [49] analyze the exact convergence rate for a fully connected network, and Boyd et al. [22] analyze this algorithm in an arbitrary topology. Jelasity et al. [67] periodically restart the push-sum algorithm to handle dynamic settings, trading off accuracy and bandwidth. Although these algorithms do not deal with dynamic inputs and topology as we do, we borrow some techniques from them. In particular, our algorithm is inspired by the Push-Sum construct, and operates in a similar manner in static settings. The aforementioned analyses therefore apply to our algorithm if and when the system stabilizes.

We are aware of two approaches to aggregate dynamic inputs. The first, by Birk et al. [18], is limited to unrealistic settings, namely a static topology with reliable communication links, failure freedom, and synchronous operation. The second approach, called flow updates [69, 68, 5] also solves aggregation in dynamic settings, overcoming message loss, dynamic topology and churn, albeit in synchronous settings only, running in rounds. Though the technique is illustrated to work in one simulation with dynamic inputs, the correctness proof and analysis [5] cover static inputs only, and the paper does not prove converge to the correct average in the face of message loss. Finally, Flow-Update requires maintaining the aggregate of the messages sent on a link, resulting in an unbounded variable size, a challenge LiMoSense overcomes.

Note that aggregation in sensor networks is distinct from other aggregation problems, such as stream aggregation, where the data in a sliding window is summarized. In the latter, a single system component has the entire data, and the distributed aspects do not exist.

3.2 Model and Problem Definition

3.2.1 Model

The system is comprised of a dynamic set of nodes (sensors), partially connected by dynamic undirected communication links. Two nodes connected by a link are called *neighbors*, and they can send messages to each other. These messages either arrive at some later time, or are lost. Messages that are not lost on each link arrive in FIFO order. Links do not generate or duplicate messages.

The system is asynchronous and progresses in steps, where in each step an event happens and the appropriate node is notified, or a node acts spontaneously. Spontaneous steps occur infinitely often. In a step, a node may change its internal state and send messages to its neighbors.

Nodes can be dynamically added to the system, and may fail or be removed from the system (churn). The set of nodes at time t is denoted $\mathcal{N}^t$ and their number n^t. The *system state* at time t consists of the internal states of all nodes in $\mathcal{N}^t$, and the links among them. When a node is added (`init` event), it is notified, and its internal state becomes a part of the system state. When it is removed (`remove` event), it is not allowed to perform any action, and its internal state is removed from the system state.

Each sensor has a time varying *data read* in $\mathbb{R}$. A node's initial data read is provided as a parameter when it is notified of its `init` event. This value may later change (`change` event) and the node is notified with the newly read value. For a node i in $\mathcal{N}^t$, we denote[1] by r_i^t, the latest data read provided by an `init` or `change` event at that node before time t.

Communication links may be added or removed from the system. A node is notified of link addition (`addNeighbor` event) and removal (`removeNeighbor` event), given the identity of the link that was added or removed. We call these *topology events*[2]. For convenience of presentation, we assume that initially, nodes have no links, and they are notified of their neighbors by a series of `addNeighbor` events. We say that a link (i, j) is *up* at step t if by step t, both nodes i and j had received an appropriate `addNeighbor` notification and no later `removeNeighbor` notification. Note that a link (i, j) may be *half-up* in the sense that the node i was notified of its addition but node j was not, or if node j had failed.

A node may send messages on a link only if the last message it had

[1] For any variable, the node it belongs to is written in subscript and, when relevant, the time is written in superscript.

[2] There is a rich literature dealing with the means of detecting failures, usually with timeouts. This subject is outside the scope of this work.

received regarding the state of the link is `addNeighbor`. If this is the case, the node may also receive a message on the link (`receive` event).

3.2.1..3 Global Stabilization Time We define *global stabilization time*, GST, to be the first time from which onward the following properties hold: (1) The system is *static*, i.e., there are no `change`, `init`, `remove`, `addNeighbor` or `removeNeighbor` events. (2) If the latest topology event a node $i \in \mathcal{N}^{\text{GST}}$ has received for another node j is `addNeighbor`, then node j is alive, and the latest topology event j has received for i is also `addNeighbor` (i.e., there are no half-up links). (3) The network is connected. (4) If a link is up after GST, and infinitely many messages are sent on it, then infinitely many of them arrive.

3.2.2 The Live Average Monitoring Problem

We define the *read average* of the system at time t as $R^t \triangleq \frac{1}{|\mathcal{N}^t|} \sum_{i \in \mathcal{N}^t} r_i^t$. Note that the read average does not change after GST. Our goal is to have all nodes estimate the read average after GST. More formally, an algorithm solves the *Live Average Monitoring Problem* if it gets time-varying data reads as its inputs, and has nodes continuously output their *estimates* of the average, such that at every node in $\mathcal{N}^{\text{GST}}$, the output estimate converges to the read average after GST.

3.2.2..4 Metrics We evaluate live average monitoring algorithms using the following metrics: (1) *Mean square error*, *MSE*, which is the mean of the squares of the distances between the node estimates and the read average; and (2) ε*-inaccuracy*, which is the percentage of nodes whose estimate is off by more than ε.

3.3 The LiMoSense Algorithm

In Section 3.3.1 we describe a simplified version of the algorithm for dynamic inputs but static topology and no failures. This simplified version demonstrates our novel approach for handling dynamic inputs. However, this simplified version is unable to accommodate topology changes, churn, and message loss. To overcome these, we present in Section 3.3.2 a robust algorithm, in which each node maintains for each of its links a summary of the data communicated over that link thereby enabling it to recover after these changes. These summaries, however, are aggregates of all exchanges on the links, and their size grows unboundedly. In Section 3.3.3, we describe

Algorithm 1: Failure-Free Dynamic Algorithm

state
  $(est_i, w_i) \in \mathbb{R}^2$
  $prevRead_i \in \mathbb{R}$

on $\texttt{init}_i(initVal)$
  $(est_i, w_i) \leftarrow (initVal, 1)$
  $prevRead_i \leftarrow initVal$

on $\texttt{receive}_i((v_{\text{in}}, w_{\text{in}}))$ from j
  $(est_i, w_i) \leftarrow (est_i, w_i) \oplus (v_{\text{in}}, w_{\text{in}})$

periodically $\texttt{send}_i()$
  Choose a neighbor j uniformly at random.
  $w_i \leftarrow w_i/2$
  send $((est_i, w_i))$ to j

on $\texttt{change}_i(newRead)$
  $est_i \leftarrow est_i + \frac{1}{w_i} \cdot (newRead - prevRead_i)$
  $prevRead_i \leftarrow newRead$

the complete LiMoSense algorithm, which also implements a clearing mechanism that results in bounded sizes of all its variables and messages, without resorting to atomicity or synchrony assumptions.

3.3.1 Failure-Free Dynamic Algorithm

We begin by describing a version of the algorithm that handles dynamically changing inputs, but assumes no message loss or link or node failures. The pseudocode is shown in Algorithm 1.

The base of the algorithm operates like Push-Sum [72, 21]: Each node maintains a weighted estimate of the read average (a pair containing the estimate and a weight), which is updated as a result of the node's communication with its neighbors. As the algorithm progresses, the estimate converges to the read average.

In order to accommodate dynamic reads, a node whose read value changes must notify the other nodes. It not only needs to introduce the new value, but also needs to undo the effect of its previous read value, which by now has partially propagated through the network.

The algorithm often requires nodes to merge two weighted values into one. They do so using the *weighted value sum* operation, which we define below and concisely denote by $\oplus$. Subtraction operations will be used later, they are denoted by $\ominus$ and are also defined below. The $\oplus$ and $\ominus$ operations are undefined when the sum (resp. difference) between the weights of the operands is zero. We note that the $\oplus$ operation is commutative and both

operations are associative.

$$(v_a, w_a) \oplus (v_b, w_b) \triangleq (\frac{v_a w_a + v_b w_b}{w_a + w_b}, w_a + w_b) \ . \tag{3.1}$$

$$(v_a, w_a) \ominus (v_b, w_b) \triangleq (v_a, w_a) \oplus (v_b, -w_b) \ . \tag{3.2}$$

The state of a node (lines 2–3) consists of a weighted value, (est_i, w_i), where est_i is an output variable holding the node's estimate of the read average, and the value $prevRead_i$ of the latest data read. We assume at this stage that each node knows its set of neighbors. We shall remove this assumption later, in the robust LiMoSense algorithm.

Node i initializes its state on its `init` event. The data read is initialized to the given value $initVal$, and the estimate is $(initVal, 1)$ (lines 5–6).

The algorithm is implemented with the functions `receive` and `change`, which are called in response to events, and the function `send`, which is called periodically.

Periodically, a node i shares its estimate with a neighbor j chosen uniformly at random (line 10). It transfers half of its estimate to node j by halving the weight w_i of its locally stored estimate and sending the same weighted value to that neighbor (lines 11-12). When the neighbor receives the message, it merges the accepted weighted value with its own (line 8). Nodes keep their weights larger than some small arbitrary size q, by performing a `send` only if the node's weight is larger than $2q$. A small value of q therefore increases the weight sending frequency among nodes, but it does not affect the accuracy of estimation.

Correctness of the algorithm in static settings follows from two key observations. First, *safety* of the algorithm is preserved, because the system-wide weighted average over all weighted-value estimate pairs at all nodes and all communication links is always the correct read average; this invariant is preserved by send and receive operations. Thus, no information is ever "lost". Second, the algorithm's *convergence* follows from the fact that when a node merges its estimate with that received from a neighbor, the result is closer to the read average.

We proceed to discuss the dynamic operation of the algorithm. When a node's data read changes, the read average changes, and so the estimate should change as well. Let us denote the previous read of node i by r_i^{t-1} and the new read at step t by r_i^t. In essence, the new read, r_i^t, should be added to the system-wide estimate with weight 1, while the old read, r_i^{t-1}, ought to be deducted from it, also with weight 1. But since the old value has been distributed to an unknown set of nodes, we cannot simply "recall" it. Instead, we make the appropriate adjustment locally, allowing the natural flow of the algorithm to propagate it.

We now explain how we compute the local adjustment. The system-wide estimate should shift by the difference between the read values, factored by the relative influence of a single sensor, i.e., $1/n$. So an increase of x increases the system-wide estimate by x/n. However, when a node's read value changes, its estimate has an arbitrary weight of w, so we need to factor the change of its value by $1/w$ to obtain the required system-wide shift. Therefore, in response to a `change` event at time t, if the node's estimate before the change was est_i^{t-1} and its weight was w_i^{t-1}, then the estimate is updated to (lines 14-15)

$$est_i^t = est_i^{t-1} + (r_i^t - r_i^{t-1})/w_i^{t-1} \ .$$

Note that the value of n^t does not appear in the equation, as it is unknown to any of the nodes.

3.3.2 Adding Robustness

Overcoming failures is challenging in an asynchronous system, where a node cannot determine whether a message it had sent was successfully received. In order to overcome message loss and link and node failure, each node maintains a summary of its conversations with each of its neighbors. Each node i maintains the aggregates (as weighted sums) of the messages received from and sent to node j in the variables $receivedTotal_i(j)$ and $sentTotal_i(j)$, respectively. Nodes interact by sending and receiving these summaries, rather than weighted values as in the failure-free algorithm. The data in each message subsumes all previous value exchanges on the same link. Thus, if a message is lost, the lost data is recovered once an ensuing message arrives. When a link fails, the nodes at both of its ends use the summaries to retroactively cancel the effect of all the messages ever transferred over it. A node failure is treated as the failure of all its links. The resulting algorithm, whose pseudocode is given in Algorithm 2, is robust to message loss, link failure, and churn.

To send, node i adds to $sentTotal_i(j)$ the weighted value it wants to send, and sends $sentTotal_i(j)$ to j (lines 18–20). When receiving this message, node j calculates the newly received weighted value by subtracting its $receivedTotal_i(j)$ variable from the newly received aggregate (line 22). After acting on the received message (line 23), node j replaces its $receivedTotal$ variable with the new weighted value (line 24). Thus, if a message is lost, the next received message compensates for the loss and brings the receiving neighbor to the same state it would have reached had it received the lost messages as well. Whenever the most recent message on a link (i, j) is correctly received and there are no messages in transit, the value of $sentTotal_i^j$

Algorithm 2: Robust Dynamic Algorithm with Unbounded State

```
1  state
2      (est_i, w_i) ∈ ℝ²
3      prevRead_i ∈ ℝ
4      neighbors_i ⊂ ℕ, initially ∅
5      sentTotal_i : ℕ → ℝ², initially ∀j : sentTotal_i(j) = (0,0)
6      receivedTotal_i : ℕ → ℝ², initially ∀j : receivedTotal_i(j) = (0,0)
7      (unrecvVal_i, unrecvWeight_i) ∈ ℝ², initially (0,0)
8  on init_i(initVal)
9      (est_i, w_i) ← (initVal, 1)
10     prevRead_i ← initVal
11 periodically send_i()
12     Choose a neighbor j uniformly at random.
13     if w_i >= 2q then
14         w_toUnsend ← max(unrecvWeight_i, w_i − q)
15         (est_i, w_i) ← (est_i, w_i) ⊖ (unrecvVal_i, w_toUnsend)
16         unrecvWeight_i ← unrecvWeight_i − w_toUnsend
17     if w_i >= 2q then
18         sentTotal_i(j) ← sentTotal_i(j) ⊕ (est_i, w_i/2)
19         (est_i, w_i) ← (est_i, w_i/2)
20     send sentTotal_i(j) to j
21 on receive_i(v_in, w_in) from j
22     diff ← (v_in, w_in) ⊖ receivedTotal_i(j)
23     (est_i, w_i) ← (est_i, w_i) ⊕ diff
24     receivedTotal_i(j) ← (v_in, w_in)
25 on change_i(r_new)
26     est_i ← est_i + (1/w_i) · (r_new − prevRead_i)
27     prevRead_i ← r_new
28 on addNeighbor_i(j)
29     neighbors_i ← neighbors_i ∪ {j}
30 on removeNeighbor_i(j)
31     (est_i, w_i) ← (est_i, w_i) ⊕ sentTotal_i(j)
32     (unrecvVal, unrecvWeight) ← (unrecvVal, unrecvWeight) ⊕ receivedTotal_i(j)
33     neighbors_i ← neighbors_i \ {j}
34     sentTotal_i(j) ← (0,0)
35     receivedTotal_i(j) ← (0,0)
```

is identical to the value of $receivedTotal_j^i$. In order to overcome message loss, a node i sends its summary to its neighbor j even if its current weight is smaller than $2q$, and the message carries no new information.

Upon notification of topology events, nodes act as follows. When notified of an `addNeighbor` event, a node simply adds the new neighbor to its *neighbors* list (line 29). When notified of a `removeNeighbor` event, a node reacts by nullifying the effect of this link, clearing the state variables, removing the neighbor from its *neighbors* list, and discarding its link records (lines 31–35). The effects of sent and received messages are summarized in the respective *sentTotal* and *receivedTotal* variables. When a node i discovers that link (i, j) failed, it adds the outgoing link summary $sentTotal_i^j$ to its estimate, thus cancelling the effect of ever having sent anything on the link. In

order to prevent its estimate weight from becoming negative, $receivedTotal_i^j$ is not immediately subtracted from the estimate. Instead, the node adds the incoming link summary $receivedTotal_i^j$ to a buffer, the weighted value ($unrecvVal$, $unrecvWeight$) and lazily subtracts it from its estimate, preserving the estimate weight positive (lines 13–16). The node thus cancels the effect of everything it has received from that neighbor.

After a node joins the system or leaves it, its neighbors are notified of the appropriate topology events, adding links to the new node, or removing links to the failed one. Thus, when a node fails, any part of its read value that had propagated through the system is annulled, and it no longer contributes to the system-wide estimate.

3.3.3 LiMoSense

The summary approach of Algorithm 2 causes summary sizes, namely the weights of $receivedTotal_i(j)$ and $sentTotal_i(j)$, to increase unboundedly as the algorithm progresses. To avoid that, we devise a channel reset mechanism that prevents this without resorting to synchronization assumptions. Instead of storing the aggregates of all received and sent weights, we store only their difference, which can be bounded, and we store the received and sent aggregates only for limited epochs, thereby bounding them as well.

The result is the full LiMoSense algorithm, shown as Algorithms 3–4, where the state information of Algorithm 2 is replaced with a more elaborate scheme. Messages are aggregated in epochs, and the aggregate is reset on epoch change. Epochs are defined per node, per link, and per direction, and are identified by binary serial numbers, so each node maintains an incoming and an outgoing serial number per link. Node i maintains for its link with node j the serial numbers $inSN_i(j)$ and $outSN_i(j)$ for the incoming and outgoing weights, respectively. Epochs on different directed links are independent of each other. Neighboring nodes reset their aggregates for their connecting directed link and proceed to the next epoch after reaching consensus on the aggregate values sent in the current epoch. This approach is similar to the classical stop-and-wait message exchange protocol [100]. However, here the receiving end of the link initiates the transition to the next epoch, after receiving multiple messages. Intuitively, the stop-and-wait is performed for the ACKs, each of which acknowledges a set of weight transfers.

For a link (i, j), node i maintains in $sent_i(j)$ and $received_i(j)$ the aggregate sent and received values in the current epoch (rather than the entire history as in the failure-free algorithm). In addition, it maintains in $totalDiff_i(j)$ the difference between the sent and received aggregates over the entire history.

A channel reset for a link (i, j) is initiated by the receiver j when it notices

Algorithm 3: LiMoSense – part 1

```
1  state
2      (est_i, w_i) ∈ ℝ²
3      prevRead_i ∈ ℝ
4      neighbors_i ⊂ ℕ, initially ∅
5      totalDiff_i : ℕ → ℝ², initially ∀j : totalDiff_i(j) = (0,0)
6      (unrecvVal_i, unrecvWeight_i) ∈ ℝ², initially (0,0)
7      sent_i : ℕ → ℝ², initially ∀j : sent_i(j) = (0,0)
8      outSN_i : ℕ → {0,1}, initially ∀j : outSN_i(j) = (0,0)   (Serial number of outgoing messages)
9      received_i : ℕ → ℝ², initially ∀j : received_i(j) = (0,0)
10     inSN_i : ℕ → {0,1}, initially ∀j : inSN_i(j) = 0   (Expected serial number of incoming messages)
11     cleared_i : ℕ → ℝ², initially ∀j : cleared_i(j) = (0,0)   (Weight received with previous serial number)
12 on init_i(initVal)
13     (est_i, w_i) ← (initVal, 1)
14     prevRead_i ← initVal
15 periodically send_i()
16     Choose a neighbor j uniformly at random.
17     if w_i >= 2q then
18         w_toUnsend ← max(unrecvWeight_i, w_i − q)
19         (est_i, w_i) ← (est_i, w_i) ⊖ (unrecvVal_i, w_toUnsend)
20         unrecvWeight_i ← unrecvWeight_i − w_toUnsend
21     if w_i >= 2q and weight of totalDiff_i(j) > −2 · bound and weight of sent_i(j) < 2 · bound then
22         sent_i(j) ← sent_i(j) ⊕ (est_i, w_i/2)   (sentTotal_i(j) ← sentTotal_i(j) + (est_i, w_i/2))
23         totalDiff_i(j) ← totalDiff_i(j) ⊖ (est_i, w_i/2)
24         (est_i, w_i) ← (est_i, w_i/2)
25     send (sent_i(j), outSN_i(j), inSN_i(j) − 1 mod 2, cleared_i(j)) to j   (Ack cleared vals and serial of previous epoch)
26 on receive_i((v_in, w_in), msgSN, clearSN, clearVal) from j
27     if clearSN = outSN_i(j) then   (Relevant clear)
28         outSN_i(j) ← outSN_i(j) + 1 mod 2
29         sent_i(j) ← sent_i(j) ⊖ clearVal   (sentCleared_i(j) ← sentCleared_i(j) + clearVal)
30     if msgSN = inSN_i(j) then   (Relevant message)
31         diff ← (v_in, w_in) ⊖ received_i(j)
32         (est_i, w_i) ← (est_i, w_i) ⊕ diff
33         totalDiff_i(j) ← totalDiff_i(j) ⊕ diff
34         received_i(j) ← (v_in, w_in)   (receivedTotal_i(j) ← receivedTotal_i(j) + diff)
35         if (weight of received_i(j)) > bound then   (Reset the channel)
36             inSN_i(j) ← inSN_i(j) + 1 mod 2
37             cleared_i(j) ← received_i(j)
               (receivedCleared_i(j) ← receivedCleared_i(j) + received_i(j))
38             received_i(j) ← (0,0)
```

that the weight in $received_i(j)$ reaches a certain threshold (line 35). Node j then (1) increments modulo 2 the serial number on that link, (2) adds the aggregate received values in the completed epoch to its $totalDiff_i(j)$ summary of the link, and (3) clears the aggregate by storing $received_i(j)$ in $cleared_i(j)$ and setting $received_i(j)$ to zero (lines 36–38). Node j will not accept future messages for the previous serial number — it will simply ignore them. On its next send to i (in the inverse direction), node j's message will update i about

Algorithm 4: LiMoSense – part 2

```
39 on change_i(r_new)
40     est_i ← est_i + (1/w_i) · (r_new − prevRead_i)
41     prevRead_i ← r_new
42 on addNeighbor_i(j)
43     neighbors_i ← neighbors_i ∪ {j}
44     inSN_i(j) ← 0
45     outSN_i(j) ← 0
46 on removeNeighbor_i(j)
47     if Weight of totalDiff_i(j) < 0 then
48         (est_i, w_i) ← (est_i, w_i) ⊖ totalDiff_i(j)
49     else
50         (unrecvVal_i, unrecvWeight_i) ← (unrecvVal_i, unrecvWeight_i) ⊕ totalDiff_i(j)
51     neighbors_i ← neighbors_i \ {j}
52     totalDiff_i(j) ← (0, 0)        (receivedCleared_i(j) ← (0, 0), sentCleared_i(j) ← (0, 0))
53     sent_i(j) ← (0, 0)
54     received_i(j) ← (0, 0)
55     cleared_i(j) ← (0, 0)
```

the epoch reset by sending the index and final aggregate of the completed epoch (line 25).

When notified of the channel reset, the sender resets the aggregate for that channel, and increases modulo 2 the serial number as well. Note that i may have sent messages with the old serial number after the receiver reset the link, but these messages are ignored by j. To prevent this weight from being lost, node i does not reset its aggregate to zero, but rather to the aggregate of messages sent with the old serial number but not cleared (line 29).

Upon notification of topology events, nodes act as follows. When notified of an `addNeighbor` event, a node adds the new neighbor to its *neighbors* list and resets the epoch serial numbers for the link (lines 43–45). When notified of a `removeNeighbor` event, a node removes the neighbor from its *neighbors* list and discards its link records. Additionally, it subtracts *totalDiff* from its estimate, thus cancelling the effect of ever having communicated over the link. Unlike Algorithm 2, we cannot separate here the sent from the received weights. To prevent the estimate weight from being negative, we check if the weight in *totalDiff* is positive. If it is, we add to an aggregate buffer (*unrecvVal*, *unrecvWeight*), which is later subtracted in stages from *totalDiff* (on send events), as before.

We follow in comments the behavior of four virtual variables, the total sent and received aggregates in $sentTotal_i(j)$ and $receivedTotal_i(j)$, respectively, and the aggregates of everything that was ever cleared from $sent_i(j)$ and $received_i(j)$ in $sentCleared_i(j)$ and $receivedCleared_i(j)$, respectively. These virtual variables all grow unboundedly as the algorithm progresses and we will use them for proving correctness in Section 3.4.

3.4 Correctness

In this section, we show that the LiMoSense algorithm (Algorithms 3–4) adapts to network topology as well as value changes and converges to the correct average. We start in Section 3.4.1 by proving that when there are no half-up links, a combination of the system's variables equals the read sum. Then, in Section 3.4.2, we prove that after GST the estimates at all nodes eventually converge to the average of the latest read values.

3.4.1 Invariant

We denote by (R^t, n^t) the *read sum* at time t, as shown in Equation 3.3.

$$(R^t, n^t) = \bigoplus_{i=1}^{n} (r_i^t, 1) \tag{3.3}$$

We denote by (E^t, n) the weighted sum over all nodes at time t of their (1) weighted values, (2) outgoing link summaries in their *sent* variables, (3) the inverse of their incoming summaries in their *received* variables, and (4) the latest *cleared* received aggregate, if their neighbor has not yet received the reset message. The sum is shown in Equation 3.4

$$(E^t, n^t) = \bigoplus_{i=1}^{n} \left(\begin{array}{l} (est_i^t, w_i^t) \ominus (unrecvVal_i^t, unrecvWeight_i^t) \oplus \\ \oplus \bigoplus_{j \in neighbors_i^t} \left(sent_i^t(j) \ominus received_i^t(j) \right) \ominus \bigoplus_{\substack{j \in neighbors_i^t \text{ s.t.} \\ inSN_i^t(j) \neq \\ outSN_j^t(i)}} cleared_i^t(j) \end{array} \right) \tag{3.4}$$

We show that if there are no half-up links in the system (each link is known to be up or down by both its nodes), then $R^t = E^t$.

Lemma 1. *For any time t, if for any nodes i and j, either $j \in neighbors_i^t \wedge i \in neighbors_j^t$ or $j \notin neighbors_i^t \wedge i \notin neighbors_j^t$, then $R^t = E^t$.*

We begin by analyzing the effect of communication steps, then of dynamic steps, and then conclude by proving the statement.

Static Behavior

First, we consider `send` and `receive` events. Note that message loss is not an event and does not affect the state of the system. In particular, it does not affect the correctness of this lemma or the following ones.

Lemma 2 (Static operations)**.** *If step t is either* `send` *or* `receive` *at some node i, then $R^t - E^t = R^{t-1} - E^{t-1}$.*

Proof. First, consider a `send` step. If the weight in i is below the threshold of $2q$, no variables are changed (lines 17 and 21), so the lemma trivially holds. If the weight is above the threshold, then a certain weight is subtracted from the pairs $(unrecvVal_i^{t-1}, unrecvWeight_i^{t-1})$ and (est_i^{t-1}, w_i^{t-1}) (lines 18– 20). Since the two pairs appear with opposite signs in Equation 3.4, the value of E is unchanged. If the weight in i is still above the threshold of $2q$, then the weighted value $(est_i^{t-1}, \frac{1}{2}w_i^{t-1})$ is subtracted from the weighted value of node i, and added to $sent_i^j$, again leaving (E^t, n) according to Equation 3.4 unchanged.

Next consider a `receive` step. Lines 27–29 handle the outgoing link to j. If the message's $clearSN$ is the same as the current $outSN_i(j)$ (line 27), it causes a reset. Node i reacts by resetting $sent_i(j)$ and incrementing $outSN_i(j)$. Incrementing $outSN_i(j)$ makes it equal to its counterpart $inSN_j(i)$, removing the negative $cleared_j(i)$ element from the sum in Equation 3.4. Decreasing $sent_i(j)$ by the same value, leaves E^t in Equation 3.4 unchanged.

Next, if the incoming message carries the appropriate serial number, the incoming value (deducting the previously received value from the incoming aggregate) is added to the weighted value of node j, and the same weighted value is added to $received_j^i$. Since the latter is subtracted in Equation 3.4, this leaves (E^t, n) unchanged.

Finally, if the weight in the incoming message is too high, the receiver initiates a channel reset. Note that the incoming message serial number equals $outSN_j(i)$. The node increments $inSN_i(j)$, causing $cleared_i(j)$ to be counted in the sum, since after the change it becomes different than $outSN_j(i)$. Then it stores the value of $received_i(j)$ in $cleared_i(j)$, and nullifies $received_i(j)$, both with negative sign in Equation 3.4, leaving (E^t, n) unchanged.

None of these events changes the read sum, therefore, since neither the read sum nor (E^t, n) change, $R^t - E^t = R^{t-1} - E^{t-1}$ □

Dynamic Values

When the value read by node i changes from r_i^{t-1} to r_i^t, the node updates its estimate in a manner that changes (E, n) correctly, as shown in the following lemma.

Lemma 3 (Read value change)**.** *If step t is* `change` *at node i, then $R^t - E^t = R^{t-1} - E^{t-1}$*

Proof. After the change of the read value, the new read average is $R^t = R^{t-1} + \frac{r_i^t - r_i^{t-1}}{n^{t-1}}$, and the weighted value[3] $\left(est_i^{t-1} + \frac{r_i^t - r_i^{t-1}}{w_i}, w_i\right)$ replaces the weighted value of node i. We show that the new (E, n) changes just like the read sum:

$$(E^t, n^t) = (E^{t-1}, n^{t-1}) \ominus \left(est_i^{t-1}, w_i^{t-1}\right) \oplus \\ \oplus \left(est_i^{t-1} + \frac{r_i^t - r_i^{t-1}}{w_i^{t-1}}, w_i^{t-1}\right) = \\ = (E^{t-1} + \frac{r_i^t - r_i^{t-1}}{n^{t-1}}, n^{t-1}) \ ,$$

leaving the difference between R and E unchanged. □

Dynamic Topology

When a link is added, the node adding it starts to keep track of the messages passed on the link. When a link is removed, the node retroactively cancels the messages that passed through this link, as if it never existed. In both cases, both E^t and R^t are unchanged, as we now show.

Lemma 4 (Dynamic Topology)**.** *If step t is* `addNeighbor` *at node i, then $R^t - E^t = R^{t-1} - E^{t-1}$, and if the link between nodes i and j fails and its nodes receive* `removeNeighbor` *at times t_i and t_j (respectively), with $t_i < t_j$, then*

$$(E^{t_i}, n^{t_i}) - (E^{t_i - 1}, n^{t_i - 1}) = (E^{t_j - 1}, n^{t_j - 1}) - (E^{t_j}, n^{t_j}) \ .$$

Proof. The `addNeighbor` function does not affect the read sum or E^t, so the claim holds. We proceed to handle link failure. When the failure is discovered at t_i by i, the weighted value $totalDiff_i^{t-1}(j)$ is subtracted from est_i or added to $(unrecvVal_i, unrecvWeight_i)$ at node i, and the variables $sent_i(j)$, $received_i(j)$ and $cleared_i(j)$ are nullified. The same happens in j at t_j.

We note that $totalDiff_i(j)$ does not directly appear in Equation 3.4. We decompose $totalDiff_i(j)$ to the difference between the virtual variables $receivedTotal_i(j)$ and $sentTotal_i(j)$, defined above.

$$totalDiff_i(j) = receivedTotal_i(j) \ominus sentTotal_i(j) \ .$$

We also note that summing virtual variables $sentCleared_i(j)$ and $receivedCleared_i(j)$, together with the real variables $sent_i(j)$ and $received_i(j)$ (respectively) results in $sentTotal_i(j)$ and $receivedTotal_i(j)$, respectively. Therefore, when

[3]Note that the weight at a node never drops below q, so the expression is valid.

subtracting $totalDiff_i(j)$ in i's side, we subtract

$$\begin{aligned} totalDiff_i(j) = receivedTotal_i(j) \ominus sentTotal_i(j) = \\ = receivedCleared_i(j) \oplus received_i(j) \ominus \\ \ominus sentCleared_i(j) \ominus sent_i(j) \ . \end{aligned}$$

Now, the $received_i(j)$ and $sent_i(j)$ cancel each other on i's side, as they are subtracted and added (respectively) directly (lines 53–54) when clearing $totalDiff_i(j)$ (line 52).

On the other hand, $receivedCleared_i(j)$ and $sentCleared_i(j)$ must be canceled by an inverse change on j's side. Note that if $inSN_i(j) = outSN_j(i)$, we have $receivedCleared_i(j) = sentCleared_j(i)$, and $cleared_i(j)$ is not counted in Equation 3.4, whereas if $inSN_i(j) \neq outSN_j(i)$ then $cleared_i(j)$ is counted, and we have $receivedCleared_i(j) + cleared_i(j) = sentCleared_j(i)$. In both cases, the change is canceled, i.e., inverse weighted values are subtracted from/added to est and $(unrecvVal, unrecvWeight)$ (respectively) at t_i and t_j, and the equation in the lemma holds. □

Dynamic Node Set

When a node is added, its state is added to the system. When it is removed, its state is removed.

Lemma 5 (Dynamic Node Set)**.** *If step t is* `init` *or* `remove`*, then $R^t - E^t = R^{t-1} - E^{t-1}$*

Proof. An addition of a node i with initial estimate r_i^t results in $(R^t, n^t) = (R^{t-1}, n^{t-1}) \oplus (r_i^t, 1)$ and $(E^t, n^t) = (E^{t-1}, n^{t-1}) \oplus (r_i^t, 1)$, so their difference is unchanged at step t.

We model the failure of a node i as the failure of all its links, followed by its removal from the system. The failure of the links leaves i with its most recent read value and a weight of one, $(r_i^{t-1}, 1)$, and all other state variables empty ($totalDiff_i(j)$, $sent_i(j)$, etc.), with (E^t, n) unchanged.

Removing the node thus results in $(R^t, n^t) = (R^{t-1}, n^{t-1}) \ominus (r_i^{t-1}, 1)$ and $(E^t, n^t) = (E^{t-1}, n^{t-1}) \ominus (r_i^{t-1}, 1)$, so their difference is unchanged at step t. □

We are now ready to prove Lemma 1.

Proof. Initially, at $t = 0$, the claim holds, since for any node i, the component of the read sum is identical to that of E^t: $(r_i^t, 1) = (est_i^t, 1)$.

According to Lemmas 2–5, the difference between R^t and E^t changes only due to link failure events. Since there are no half-up links, then if a node i

detected the failure of its link with j before t, then j has also detected the failure of the link before t. Lemma 4 shows that the resulting operations by i and j compensate each other, resulting in the required equality at t. □

3.4.2 Convergence

We show that after GST the estimate at all nodes converges to the read average. Since after GST messages are not lost, we can simplify our proof by abstracting away the fact that messages contain aggregated values; instead, we consider each message to deliver only the delta from the previous one, as translated in the code to the *diff* variable upon receipt (line 31).

First, we prove in Section 3.4.2.a that connected nodes send each other values infinitely often. Then, in Section 3.4.2.b, we define the tracking of the propagation of the weighted value from a node i at time t at any time later time. We proceed to show in Section 3.4.2.c that there exists a time t' after t such that the ratio of the weight propagated to any node j from any node i, relative to the total weight at j, is bounded from below. In Section 3.4.2.d we construct a series of such times, where in each time t_x the values from t_{x-1} have propagated and match this bound. This allows us to prove convergence, as required.

3.4.2.a Fair Scheduling

We begin by proving the following lemma.

Lemma 6. *every node sends weight to each of its neighbors infinitely often.*

Proof. We prove by contradiction. Assume that a node i never sends a message to its neighbor j. Since neighbors are chosen infinitely often (we assume a fair scheduler and neighbors are chosen uniformly at random), this means the condition of line 21 evaluates to false.

The last part of the condition may evaluate to false only if weight was sent to j but not received. Once this weight is received, node j changes epochs (line 36–38), which will reset $sent_i(j)$ once the next message arrives from j to i, and the condition will evaluate to true.

Therefore, either the first or the second part of the condition do not hold. Assume first that the first part does not hold, i.e., the weight in i is always smaller than $2q$. This means that none of i's neighbors ever sends it weight (from some time). Otherwise, eventually $unrecvWeight_i$ would drop to zero, and subsequently est_i would rise above zero. Assume that none of i's neighbors ever sends it weight also due to their weights being smaller than $2q$, and continue similarly, i.e., all nodes in the system hold a weight

smaller than $2q$. Since the entire weight in the *est* variables is at least n (possibly more, if nodes have non-zero *unrecvWeight* variables)), at least one node must hold a weight larger than one, i.e., larger than $2q$, and we reach a contradiction.

Maintaining our initial assumption, we conclude that there exists some node i that never sends weight to a neighbor j since the second part of the condition holds, i.e., it already sent to j much more than it got back. Once a message from i successfully reaches j, the value of j's $totalDiff_j(i)$ is correctly updated to $-totalDiff_i(j)$, so it is positive. Therefore the second part of the condition is true in j. However, if j sends weight to i, the value of $totalDiff_i(j)$ eventually becomes positive, contradicting our assumption, and we conclude that j stops sending weight at some point, since its weight never rises above $2q$. For that to happen, j must not receive weight from any of its neighbors. So each of j's neighbors either has a weight less than $2q$, or has already sent to j more than it got back. If all of j's neighbors (including i) have sent it more than they got back, than j's weight would be more than $2q$, which we already ruled out. Therefore at least one neighbor k received from j more than it sent, but it does not send weight to j because its own (k's) weight is too small. Now, the same logic that held for j holds for k, and we continue this, forming a chain of nodes. Each node in the chain holds a weight smaller than $2q$. At the end of such a chain (and there is an end, since the number of nodes is finite) there is a node z that does not send weight to any of its neighbors, but has received from each of them more weight than it has sent. The weight at node z is therefore larger than one, and its *totalDiff* for all its neighbors is positive, So the condition in line 21 holds, and it should have sent weights to its neighbors, leading to a contradiction. □

After GST, no links failures are detected. Since weights are sent infinitely often between neighbors by Lemma 6, we conclude that there exists a time $\overline{\text{GST}} \geq \text{GST}$ after which the *unrecvWeight* variables at all nodes are zero:

Definition 1 ($\overline{\text{GST}}$)**.** *The time $\overline{GST}$ is a time after which for all $i \in \mathcal{N}^{GST}$ and for all $t > \overline{GST}$: $unrecvWeight_i^t = 0$.*

3.4.2.b Propagation Tracking

We explain how to track the propagation of the weighted value from a node i as of some time $t > \overline{\text{GST}}$. The definition recursively defines two components maintained at each node k: The *prop* component, $(est_i^t, w_{k,prop}^{t'})$, which is the propagation of i's weighted value at t to k at t', and the *agg* component, $(est_{k,agg}^{t'}, w_{k,agg}^{t'})$, which is the aggregation from all nodes but i. The *prop* component is called the *component of est_i^t at node k at time t'*. Though these

definitions depend on i and t, we fix i and t and omit them, to make the expressions cleaner.

Definition 2 (Propagation tracking)**.** *Initially, at t, at all nodes $k \neq i$, agg is the weighted value (est_k^t, w_k^t), and prop is $(0,0)$. At node i, agg is $(0,0)$ and prop is (est_i^t, w_i^t).*

For all steps $t' > t$:

1. *If the operation at t' is a send at node k, then*

$$(est_{k,agg}^{t'}, w_{k,agg}^{t'}) = (est_{k,agg}^{t'-1}, w_{k,agg}^{t'-1}/2)$$

and

$$(est_i^t, w_{k,prop}^{t'}) = (est_i^t, w_{k,prop}^{t'-1}/2)$$

and the message sent is partitioned:

$$(est_{k,agg}^{t'-1}, w_{k,agg}^{t'-1}/2) \oplus (est_i^t, w_{k,prop}^{t'-1}/2) \ .$$

2. *If the operation at t' is a receive at node k of a message (v_{in}, w_{in}) partitioned to $(est_{in,agg}, w_{in,agg})$ and $(est_i^t, w_{in,prop})$ components, then*

$$(est_{k,agg}^{t'}, w_{k,agg}^{t'}) = (est_{k,agg}^{t'-1}, w_{k,agg}^{t'-1}) \oplus (est_{in,agg}, w_{in,agg})$$

and

$$(est_i^t, w_{k,prop}^{t'}) = (est_i^t, w_{k,prop}^{t'-1}) \oplus (est_i^t, w_{in,prop}) \ .$$

It can be readily seen that the *agg* and *prop* components partition the weighted value at the node k at all times $t' \geq t$:

$$(est_k^{t'}, w_k^{t'}) = (est_{k,agg}^{t'}, w_{k,agg}^{t'}) \oplus (est_i^t, w_{k,prop}^{t'}) \ .$$

We define the *component ratio* of node i at a node k to be the ratio between i's *prop* component in k and the total weight at k:

Definition 3 (Component ratio)**.** *The* component ratio *of est_i^t at node k at $t' > t$ is*

$$\frac{w_{k,prop}^{t'}}{w_{k,prop}^{t'} + w_{k,agg}^{t'}} = \frac{w_{k,prop}^{t'}}{w_k^{t'}} \ .$$

3.4.2.c Bounded Ratio

We proceed to prove that for any time t after $\overline{\text{GST}}$, eventually each node has a component of est_i^t with a ratio that is bounded from below.

Denote by M_i^s the set of nodes with an est_i^t component at time $s > t$. Denote by $w_{M_i^s}$ the sum of weights at the nodes in M_i^s and in messages sent from nodes in $M_i^{s'}$ with $s' < s$ and not yet received.

Lemma 7. *Given two times s and t s.t. $s > t > \overline{GST}$, at all nodes in M_s^i, the est_i^t component ratio is at least $\left(\frac{q}{w_{M_i^s}}\right)^{w_{M_i^s}/q}$.*

Proof. We prove by induction on the steps taken from t. We omit the i superscript for M^i hereinafter.

At time t the only node with an est_i^t component is i with a ratio of one, and the invariant holds. Consider the system at time s, assuming the invariant holds at $s-1$. We show that after any of the possible events at s, the invariant continues to hold.

1. Send: No effect on the invariant. The ratio at the sender stays the same, and w_M is unchanged.

2. Receive from $j \notin M_i^{s-1}$ by $k \notin M_i^{s-1}$: No effect on the invariant since no nodes in M are concerned.

3. Receive from $j \in M_i^{s-1}$ by $k \notin M_i^{s-1}$: Two things change: (1) $w_{M_i^s} = w_{M_i^{s-1}} + w_k^{s-1}$ and (2) k becomes a part of M_i. The first change decreases the lower bound, therefore the assumption holds at s for all nodes in M^{s-1}. Denote by α the ratio at j when it sent the message. According to the induction assumption, $\alpha \geq \left(\frac{q}{w_{M_i^{s-1}}}\right)^{w_{M_i^{s-1}}/q}$. The new ratio at k is minimal when the weight of the received message is minimal (i.e.,

q). Therefore, the ratio at k, which is now also in M_i, is at least

$$\alpha \cdot \frac{q}{w_k^{s-1}} \overset{\text{induction hypothesis}}{\geq}$$

$$\geq \left(\frac{q}{w_{M_i^{s-1}}}\right)^{w_{M_i^{s-1}}/q} \frac{q}{w_k^{s-1}}^{\frac{w_k^{s-1}}{q}>1} >$$

$$> \left(\frac{q}{w_{M_i^{s-1}}}\right)^{w_{M_i^{s-1}}/q} \left(\frac{q}{w_k^{s-1}}\right)^{w_k^{s-1}/q} >$$

$$> \frac{q^{\frac{w_{M_i^{s-1}}+w_k^{s-1}}{q}}}{(w_{M_i^{s-1}}+w_k^{s-1})^{\frac{w_{M_i^{s-1}}}{q}}(w_{M_i^{s-1}}+w_k^{s-1})^{\frac{w_k^{s-1}}{q}}} =$$

$$= \left(\frac{q}{w_{M_i^{s-1}}+w_k^{s-1}}\right)^{\frac{w_{M_i^{s-1}}+w_k^{s-1}}{q}} =$$

$$= \left(\frac{q}{w_{M_i^s}}\right)^{w_{M_i^s}/q}.$$

We conclude that the ratio at all the nodes in M_s is larger than the bound at s.

4. Receive from $j \notin M_i^{s-1}$ by $k \in M_i^{s-1}$: Denote the weight of the message by w_{in}. Two things change: (1) $w_{M_i^s} = w_{M_i^{s-1}} + w_{\text{in}}$ and (2) the ratio at k. The change of w_{M_i} decreases the bound, therefore the assumption holds at s for all nodes other than k. The relative weight at k before receiving is at least $\left(\frac{q}{w_{M_i^{s-1}}}\right)^{w_{M_i^{s-1}}/q}$. Therefore, after receiving the message, it is at least

$$\left(\frac{q}{w_{M_i^{s-1}}}\right)^{w_{M_i^{s-1}}/q} \cdot \frac{q}{q+w_{\text{in}}} >$$

$$> \left(\frac{q}{w_{M_i^{s-1}}+w_{\text{in}}}\right)^{\frac{w_{M_i^{s-1}}+w_{\text{in}}}{q}} =$$

$$= \left(\frac{q}{w_{M_i^s}}\right)^{w_{M_i^s}/q}.$$

We conclude that the ratio at all the nodes in M_s is larger than the bound at s.

5. Receive from $j \in M_i^{s-1}$ by $k \in M_i^{s-1}$: The ratio does not decrease below the minimum between the ratios in j and k, therefore the invariant's correctness follows directly from the induction hypothesis.

□

Lemma 8. *For any time $t > \overline{GST}$ and node i, there exists a time $t' > t$ after which every node j has an est_i^t component with ratio larger than $\left(\frac{q}{n}\right)^{n/q}$.*

Proof. Once a node has an est_i^t component, it will always have an est_i^t component (no operation removes it), and eventually it will succeed sending a message to all of its neighbors (lemma 6). Therefore, due to the connectivity of the network after $\overline{\text{GST}}$, and according to Lemma 7, eventually every node has an est_i^t component. Then we have $M_{\overline{\text{GST}}} = \mathcal{N}$, so $w_{M^{=}_{\overline{\text{GST}}}} n$, and the ratio is not smaller than $\left(\frac{q}{n}\right)^{n/q}$. □

3.4.2.d Convergence

Theorem 1 (Liveness)**.** *After $\overline{GST}$, the estimate at all nodes converges to the read average.*

Proof. We construct a series of times $t_0, t_1, t_2, \ldots$ recursively, where the initial time is $t_0 = \overline{\text{GST}}$. For every t_{p-1} we define t_p to be a time from which each node $k \in \mathcal{N}^{\overline{\text{GST}}}$ has an est_i^{p-1} component with ratio at least $\left(\frac{q}{n}\right)^{n/q}$ for each $i \in \mathcal{N}^{\overline{\text{GST}}}$. Such a t_p exists according to Lemma 8.

Denote by $e_{\max}^{p-1}$ the largest estimate at a node at time t_{p-1}, i.e., $e_{\max}^{p-1} = \max_i\{est_i^{p-1}\}$. Assume without loss of generality that the average is zero. If all node estimates are the exact average, then the estimate is zero at all nodes and it does not change. Otherwise, $e_{\max}^{p-1}$ is strictly positive, and there exists some node j whose estimate is negative. At t_p, a node i has a component of $est_j^{t_{p-1}}$ with weight at least $\left(\frac{q}{n}\right)^{n/q}$ (lemma 8). The weight of the rest of its components is smaller than n, and their values are at most $e_{\max}^{p-1}$. Therefore, the estimate of i at t_p is bounded:

$$est_i^{t_p} < \left(n \cdot e_{\max}^{p-1} + \left(\frac{q}{n}\right)^{n/q} \cdot est_j^{t_{p-1}}\right) \cdot \frac{1}{n + \left(\frac{q}{n}\right)^{n/q}} \overset{est_j^{t_{p-1}} < 0}{<} \frac{n}{n + \left(\frac{q}{n}\right)^{n/q}} e_{\max}^{p-1} \quad .$$

The estimate at i is similarly bounded from below with respect to the minimal value at t_{p-1}. The maximal error (absolute distance from average) at t_p is therefore bounded by $\frac{n}{n+\left(\frac{q}{n}\right)^{n/q}}$ the maximal error at t_{p-1}. We conclude that the maximal error decreases at least exponentially with p, and therefore the estimates converge to some value x.

Now, the values in *sent*, *received* and *cleared* are occasionally reset to the *est* value of their node (*sent*) or the neighbor's (*received* and *cleared*), and since we are after $\overline{\text{GST}}$, the weight *unrecvWeight* is zero. Since these all converge to x, their weighted sum E converges to the same value, i.e., $x = E$, and we have already shown (Lemma 1) that E is equal to the value of the read sum. We conclude that *est*, converges to the average read value. □

We note that while *est* variables converge, the weights in the w variables fluctuate continuously, as nodes lose half their weight on send and similarly receive considerable weights.

3.4.2.e Bounded State Variables

To conclude, we show that if the rate of dynamism allows the algorithm to converge between events, all state variables maintained by the nodes do not grow unboundedly.

Theorem 2 (Bounded variables)**.** *If* `change` *and* `removeNeighbor` *events occur only when all* est_i *variables are in a* 2Δ *neighborhood of* R*, then all state variables are bounded.*

Proof. Consider first the value component of the aggregates. A `change` event may significantly change the estimate value at a node if holds a small weight. Denote the maximal read value change by d_{change}, the maximal read value by M and the minimal read value by m. Since the minimal weight at a node is bounded by q, the change in bounded by d_{change}/q, and the estimate value is bounded in the range $[m-\Delta/q, M+\Delta/q]$. Since the variables $received_i(j)$, $sent_i(j)$, $totalDiff_i(j)$, and $unrecvVal_i$ are aggregates of *est*'s, their values are also bounded in the same range.

As for weights, the weight in $totalDiff_i(j)$ is explicitly bounded by having the sending node stop sending if its weight is too negative, and hence it is too positive on the opposite side (line 21). The weight of $received_i(j)$ is also explicitly bounded by resetting and changing epoch if its weight is too high (line 35). Bounding $received_i(j)$ automatically bounds $cleared_i(j)$, and $sent_j(i)$ is bounded explicitly (line 21). Finally, we bound the weight of est_i. Initially, the sum of weights in all *est* variables is n, and this sum changes on weight send/receive and on `removeNeighbor`. Send and receive

cannot increase the sum of weights, since weight received was previously sent. On the other hand, `removeNeighbor` events change est_i by subtracting $totalDiff_i(j)$, possibly increasing/decreasing the sum of weights. Since the *totalDiff* weights are bounded by $2 \times \mathsf{bound}$, and each link could (temporarily) increment the sum of *est* weights in the system (if one side increases its estimate, and the other postpones the decrease to avoid negative weight), we conclude that the sum of *est* weights, and hence each *est* weight, is bounded by $n + 2 \times \mathsf{bound} \times n^2$. The *unrecvWeight* variables are similarly bounded, as they increase only on link failure by the difference of weights transferred on that link. □

3.5 Evaluation

In order to evaluate LiMoSense in the dynamic settings it was designed for, we have conducted simulations of various scenarios. Our goal is to asses how fast the algorithm reacts to changes, and succeeds to provide accurate information.

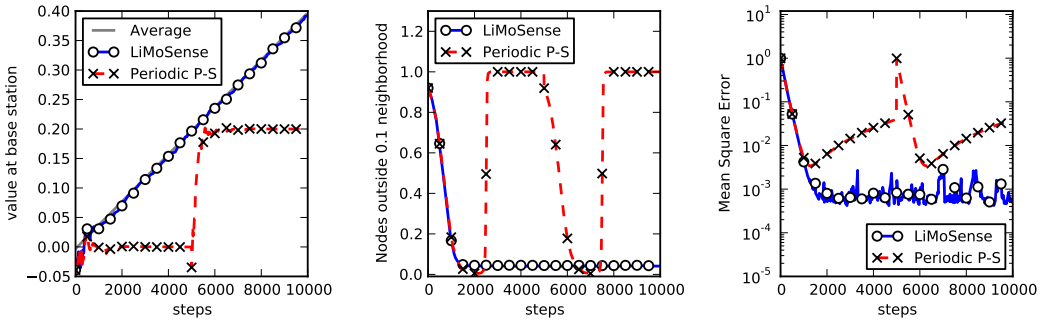
We compare LiMoSense to a periodically-restarting Push-Sum algorithm. We explain our methodology and metrics in Section 3.5.1.

We first study how the algorithm copes with different types of data read changes - a gradual "creeping" change of all values, occurring, e.g., when temperature is gradually rising (Section 3.5.2), an abrupt value change captured by a step function (Section 3.5.3), and a temporary glitch or impulse (Section 3.5.4). We then study the algorithm's robustness to node and link failures (Section 3.5.5).

3.5.1 Methodology

We performed the simulations using a custom made Python event driven simulation that simulated the underlying network and the nodes' operation. Unless specified otherwise, all simulations are of a fully connected network of 100 nodes, with initial values taken from the standard normal distribution. We have seen that in well connected networks the convergence behavior is similar to that of a fully connected network. The simulation proceeds in steps, where in each step, the topology and read values may change according to the simulated scenario, and one node sends a message. Scheduling is uniform synchronous, i.e., the node performing the action is chosen uniformly at random.

Unless specified otherwise, each scenario is simulated 1000 times. In all simulations, we track the algorithm's output and accuracy over time. In all



(a) Base station value read (median)

(b) % nodes off by > 0.1 (average)

(c) MSE (average)

Figure 3.1: **Creeping value change** Every 10 steps, 5 random reads increase by 0.01. We see that LiMoSense promptly tracks the creeping change. It provides accurate estimates to 95% of the nodes, with an MSE of about 10^{-3} throughout the run. The accuracy depends on the rate of change. In contrast, Periodic Push-Sum converges to the correct average only for a short period after each restart, before the average creeps away.

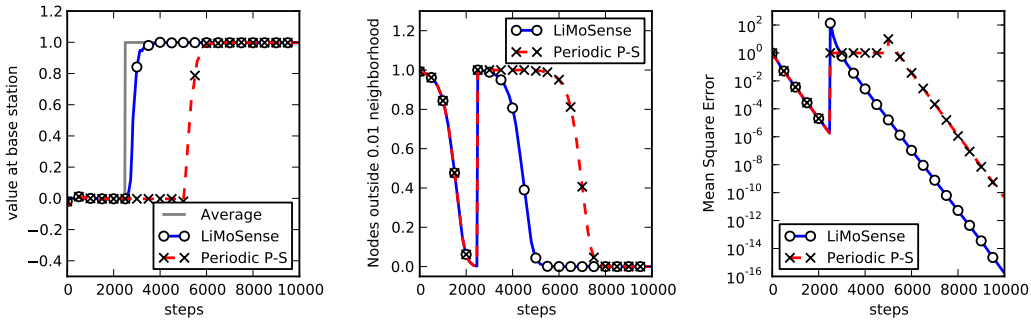
of our graphs, the X axis represents steps in the execution. We depict the following three metrics for each scenario:

(a) **base station**. We assume that a base station collects the estimated read average from some arbitrary node. We show the median of the values obtained in the runs at each step.

(b) **ε-inaccuracy**. For a chosen ε, we depict the percentage of nodes whose estimate is off by more than ε after each step. The average of the runs is depicted.

(c) **MSE**. We depict the average square distance between the estimates at all nodes and the read average at each step. The average of all runs is depicted.

We compare LiMoSense, which does not need restarts, to a Push-Sum algorithm that restarts at a constant frequency — every 5000 steps unless specified otherwise. This number is an arbitrary choice, balancing between convergence accuracy and dynamic response. In base station results, we also show the read average, i.e., the value the algorithms are trying to estimate.



(a) Base station value read (median)

(b) % nodes off by > 0.01 (average)

(c) MSE (average)

Figure 3.2: **Response to a step function** At step 2500, 10 random reads increase by 10. We see that LiMoSense immediately reacts, quickly propagating the new values. In contrast, Periodic Push-Sum starts its new convergence only after its restart.

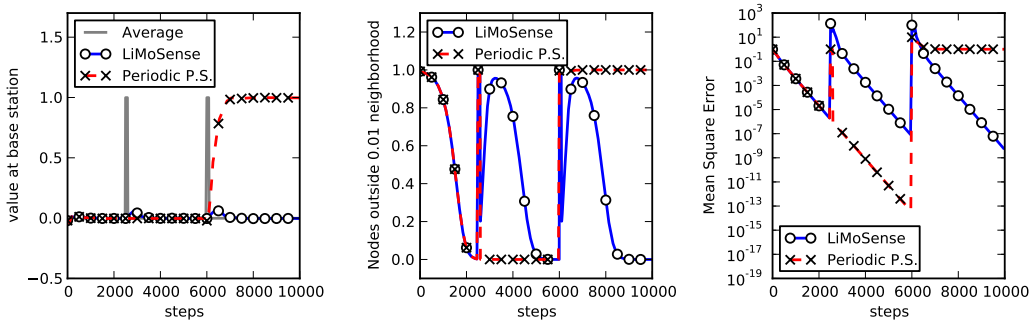
3.5.2 Slow monotonic increase

This simulation investigates the behavior of the algorithm when the values read by the sensors slowly increase. This may happen if the sensors are measuring rainfall that is slowly increasing. Every 10 steps, a random set of 5 of the nodes read values larger by 0.01 than their previous reads. The initial values are taken from the standard normal distribution. The results are shown in Figure 3.1. The accuracy of LiMoSense remains roughly constant, affected by the rate of change.

In Figure 3.1a we see that the average is increasing at a constant rate, and the LiMoSense base station closely follows. The restarting Push-Sum, however, tries to update its value only at constant intervals, unable to follow the read average. The time it takes for convergence is so long that it never gets close the read average line.

In Figure 3.1b we see that after its initial convergence, the LiMoSense algorithm has most of the nodes maintain a good estimate of the read average with less than 10% of the nodes outside the 0.1 neighborhood. The restarting Push-Sum algorithm, on the other hand, has no nodes in this neighborhood most of the time, and most of the nodes in the neighborhood only for short intervals.

Finally, in Figure 3.1c we see that the LiMoSense algorithm maintains a small MSE, with some noise, whereas the restarting Push-Sum algorithm's error quickly converges after restart, until the creeping change takes over and dominates the MSE causing a steady increase until the next restart.



(a) Base station value read (median)

(b) % nodes off by > 0.01 (average)

(c) MSE (average)

Figure 3.3: **Response to impulse** At steps 2500 and 6000, 10 random values increase by 10 for 100 steps. Both impulses cause temporary disturbances in the output of LiMoSense. Periodic Push-Sum is oblivious to the first impulse, since it does not react to changes. The restart of Push-Sum occurs during the second impulse, causing it to converge to the value measured then.

3.5.3 Step function

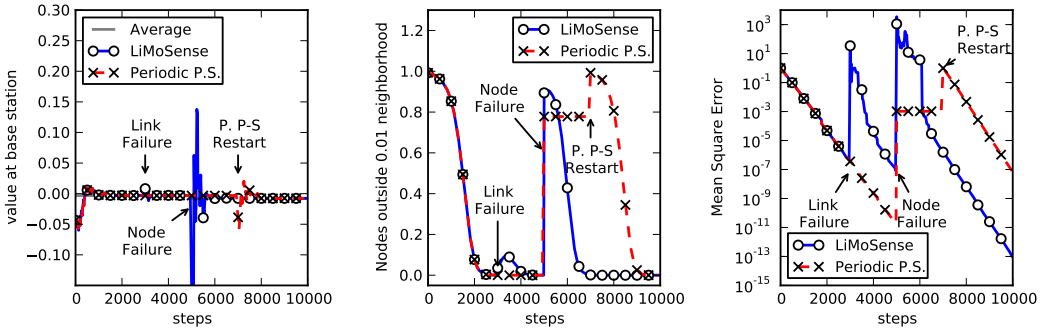
This simulation investigates the behavior of the algorithm when the values read by some sensors are shifted. This may occur due to a fire outbreak in a limited area, as close-by temperature nodes suddenly read high values.

At step 2500, a random set of 10 nodes read values larger by 10 than their previous reads. The initial values are taken from the standard normal distribution. The results are shown in Figure 3.2.

Figure 3.2a shows how the LiMoSense algorithm updates immediately after the shift, whereas the periodic Push-Sum algorithm updates at its first restart only. Figure 3.2b shows the ratio of erroneous sensors with error larger than 0.01 quickly dropping — right after the read average change for LiMoSense, and at restart for the periodic Push-Sum. Figure 3.2c shows the MSE decrease. Both LiMoSense and periodic Push-Sum converge at the same rate, but start a different times. At the step, LiMoSense suffers from a temporary MSE spike as nodes with small weight increase their estimates by a large factor.

3.5.4 Impulse Function

This simulation investigates the behavior of the algorithm when the reads of some sensors are shifted for a limited time, and then return to their previous



(a) Base station value read (median)

(b) % nodes off by > 0.01 (average)

(c) MSE (average)

Figure 3.4: **Failure robustness** In a disc graph topology, the radio range of 10 nodes decays in step 3000, resulting in about 7 lost links in the system. Then, in step 5000, a node crashes. Each failure causes a temporary disturbance in the output of LiMoSense. Periodic Push-Sum is oblivious to the link failure. It recovers from the node failure only after the next restart.

values. This may happen due to sensing errors causing the nodes to read irrelevant data. As an example, one may consider the case of a heavy vehicle driving by seismic sensors used to detect earthquakes. The close-by sensors would read high seismic activity for a short period of time.

At steps 2500 and 6000, a random set of 10 nodes read values larger by 10 than their previous reads, and after 100 steps they return to their values before the shift. The initial values are taken from the standard normal distribution. The results are shown in Figure 3.3.

The LiMoSense algorithm's reaction is independent of the impulse time — a short period of noise raises the estimate at the base station as the impulse value propagates from the sensors that read the impulse. Then, once the impulse is canceled, this value decreases. The estimate with respect to the read average is shown in Figure 3.3a, and the ratio of correct sensors is in Figure 3.3b. The impulse essentially restarts the MSE convergence, as shown in Figure 3.3c — After an impulse ends, the error returns to its starting point and starts convergence anew.

The response of the periodic Push-Sum depends on the time of impulse. If the impulse occurs between restarts (as in step 2500), the algorithm is completely oblivious to it. All three figures 3.3a–3.3c show that apart from the impulse time, convergence continues as if it never happened. If, however, a restart occurs during the impulse (as in step 6000), then the impulse is sampled and the algorithm converges to this new value. This convergence is similar to the reaction to the step function of Section 3.5.3, only in this

case it promptly becomes stale as the impulse ends. Figure 3.3a shows the error quickly propagating to the base station. Since the algorithm has the estimates converge to the read average during impulse, the ratio of inaccurate nodes is 1.0 once the impulse ends, and the MSE stabilizes at a large value as all nodes converge to the wrong estimate.

3.5.5 Robustness

To investigate the effect of link and node failures, we construct the following scenario. The sensors are spread in the unit square, and they have a transmission range of 0.7 distance units. The neighbors of a sensor are the sensors in its range. The system is run for 3000 steps, at which point, due to battery decay, the transmission range of 10 sensors decreases by 0.99. Due to this decay, about 7 links fail, and respective nodes employ their `removeNeighbor` functions. We see the effect of this link removal in Figure 3.4. In Figure 3.4a the effect can hardly be seen, but a temporary decrease of the accurate nodes can be seen in Figure 3.4b, and in Figure 3.4c we see the MSE rising sharply. The failure of links does not effect the periodic Push-Sum algorithm, which continues to converge.

In step 5000, a node fails, removing its read value from the read average. Upon node failure, all of its neighbors call their `removeNeighbor` functions. Figure 3.4a shows the extreme noise at the base station caused by the failure, and in Figure 3.4b we see the ratio of inaccurate nodes rising sharply before converging again. We see in Figure 3.4c that the node removal effectively requires the MSE convergence to restart. However, Periodic Push-Sum has no mechanism for reacting to the change until its next restart. Since the average changes, until that time, the percentage of inaccurate nodes sharply rises to 1.0, and the MSE reaches a static value, as the estimates at the nodes converge to the wrong average. Since in every run a different node crashes, and the median of the removed value is 0, the node crash does not effect the median periodic Push-Sum value at the base station in Figure 3.4a.

Chapter 4

Data Clustering in Sensor Networks

To analyze large data sets, it is common practice to employ *clustering* [40]: In clustering, the data values are *partitioned* into several *clusters*, and each cluster is described concisely using a *summary*. This classical problem in machine learning is solved using various heuristic techniques, which typically base their decisions on a view of the complete data set, stored in some central database.

However, it is sometimes necessary to perform clustering on data sets that are distributed among a large number of nodes. For example, in a grid computing system, load balancing can be implemented by having heavily loaded machines stop serving new requests. But this requires analysis of the load of all machines. If, e.g., half the machines have a load of about 10%, and the other half is 90% utilized, the system's state can be summarized by partitioning the machines into two clusters — lightly loaded and heavily loaded. A machine with 60% load is associated with the heavily loaded cluster, and should stop taking new requests. But, if the cluster averages were instead 50% and 80%, it would have been associated with the former, i.e., lightly loaded, and would keep serving new requests. Another scenario is that of sensor networks with thousands of nodes monitoring conditions like seismic activity or temperature [10, 105].

In both of these examples, there are strict constraints on the resources devoted to the clustering mechanism. Large-scale computation clouds allot only limited resources to monitoring, so as not to interfere with their main operation, and sensor networks use lightweight nodes with minimal hardware. These constraints render the collection of all data at a central location infeasible, and therefore rule out the use of centralized clustering algorithms.

In this chapter, we address the problem of *distributed clustering*. A de-

tailed account of previous work appears in Section 4.1, and a formal definition of the problem appears in Section 4.2.

A solution to distributed clustering ought to summarize data within the network. There exist distributed algorithms that calculate scalar aggregates, such as sum and average, of the entire data set [72, 50]. In contrast, a clustering algorithm must partition the data into clusters, and summarize each cluster separately. In this case, it seems like we are facing a Catch-22 [60]: Had the nodes had the summaries, they would have been able to partition the values by associating each one with the summary it fits best. Alternatively, if each value was labeled a cluster identifier, it would have been possible to distributively calculate the summary of each cluster separately, using the aforementioned aggregation algorithms.

In Section 4.3 we present a generic distributed clustering algorithm to solve this predicament. In our algorithm, all nodes obtain a clustering of the complete data set without actually hearing all the data values. The double bind described above is overcome by implementing adaptive compression: A clustering can be seen as a lossy compression of the data, where a cluster of similar values can be described succinctly, whereas a concise summary of dissimilar values loses a lot of information. Our algorithm tries to distribute the values between the nodes. At the beginning, it uses minimal compression, since each node has only little information to store and send. Once a significant amount of information is obtained, a node may perform efficient compression, joining only similar values.

Our algorithm captures a large family of algorithms that solve various instantiations of the problem — with different approaches, clustering values from any multidimensional domain and with different data distributions, using various summary representations, and running on arbitrary connected topologies. A common approach to clustering is k-means, where each cluster is summarized by its *centroid* (average of the values in the cluster), and partitioning is based on distance. A k-means approach is a possible implementation of our generic algorithm. The result of this implementation, however, would differ from that of the classical centralized k-means algorithm.

Since the summary of clusters as centroids is often insufficient in real life, machine learning solutions typically also take the variance into account, and summarize values as a weighted set of Gaussians (normal distributions), which is called a *Gaussian Mixture (GM)* [97]. In Section 4.4, we present a novel distributed clustering algorithm that employs this approach, also as an instance of our generic algorithm. The GM algorithm makes clustering decisions using a popular machine learning heuristic, *Expectation Maximization (EM)* [39]. We present in Section 4.4.2 simulation results demonstrating the effectiveness of this approach. These results show that the algorithm con-

verges quickly. It can provide a rich description of multidimensional data sets. Additionally, it can detect and remove outlying erroneous values, thereby enabling robust calculation of the average.

The centroids and GM algorithms are but two examples of our generic algorithm; in all instances, nodes independently strive to estimate the clustering of the data. This raises a question that has not been dealt with before: does this process converge? One of the main contributions of this chapter, presented in Section 4.5, is a formal proof that indeed any implementation of our generic algorithm converges, s.t. all nodes in the system learn *the same* clustering of the complete data set. We prove that convergence is ensured under a broad set of circumstances: arbitrary asynchrony, an arbitrary connected topology, and no assumptions on the distribution of the values.

Note that in the abstract settings of the generic algorithm, there is no sense in defining the destination clustering the algorithm converges to precisely, or in arguing about its quality, since these are application-specific and usually heuristic in nature. Additionally, due to asynchrony and lack of constraints on topology, it is also impossible to bound the convergence time.

In summary, this chapter makes the following contributions:

- It provides a generic algorithm that captures a range of algorithms solving this problem in a variety of settings (Section 4.3).
- It provides a novel distributed clustering algorithm based on Gaussian Mixtures, which uses machine learning techniques to make clustering decisions (Section 4.4).
- It proves that the generic algorithm converges in very broad circumstances, over any connected topology, using any clustering criterion, in fully asynchronous settings (Section 4.5).

Preliminary versions of the work presented in this chapter appear in the proceedings of the 5$^{\text{th}}$ Workshop on Hot Topics in System Dependability (HotDep'09) [46], in the proceeding of the 29$^{\text{th}}$ symposium on Principles of distributed computing (PODC'10) [47] and in the Journal of Distributed Computing [45].

4.1 Related Work

Kempe et al. [72] and Nath et al. [88] present approaches for calculating aggregates such as sums and means using gossip. These approaches cannot be directly used to perform clustering, though this work draws ideas from [72], in particular the concept of weight diffusion, and the tracing of value weights.

In the field of machine learning, clustering has been extensively studied for centrally available data sets (see [40] for a comprehensive survey). In this context, parallelization is sometimes used, where multiple processes cluster partial data sets. Parallel clustering differs from distributed clustering in that all the data is available to all processes, or is carefully distributed among them, and communication is cheap.

Centralized clustering solutions typically overcome the Catch-22 issue explained in the introduction by running multiple iterations. They first estimate a solution, and then try to improve it by re-partitioning the values to create a better clustering. K-means [82] and Expectation Maximization [39] are examples of such algorithms. Datta et al. [37] implement the k-means algorithm distributively, whereby nodes simulate the centralized version of the algorithm. Kowalczyk and Vlassis [73] do the same for Gaussian Mixture estimation by having the nodes distributively simulate Expectation Maximization. These algorithms require multiple aggregation iterations, each similar in length to one complete run of our algorithm. The message size in these algorithms is similar to ours, dependent only on the parameters of the dataset, and not on the number of nodes. Finally, they demonstrate convergence through simulation only, but do not provide a convergence proof.

Haridasan and van Renesse [59] and Sacha et al. [96] estimate distributions in sensor networks by estimating histograms. Unlike this work, these solutions are limited to one dimensional data values. Additionally, both use multiple iterations to improve their estimations. While these algorithms are suitable for certain distributions, they are not applicable for clustering, where, for example, small sets of distant values should not be merged with others. They also do not prove convergence.

4.2 Model and Problem Definitions

4.2.1 Network Model

The system consists of a set of n nodes, connected by communication channels, s.t. each node i has a set of neighbors $\texttt{neighbors}_i \subset \{1, \cdots, n\}$, to which it is connected. The channels form a static directed connected network. Communication channels are asynchronous but reliable links: A node may send messages on a link to a neighbor, and eventually every sent message reaches its destination. Messages are not duplicated and no spurious messages are created.

Time is discrete, and an execution is a series of events occurring at times $t = 0, 1, 2, \cdots$.

4.2.2 The Distributed Clustering Problem

At time 0, each node i takes an input val_i — a value from a domain $\mathcal{D}$. In all the examples in this chapter, $\mathcal{D}$ is a d-dimensional Cartesian space $\mathcal{D} = \mathbb{R}^d$ (with $d \in \mathbb{N}$). However, in general, $\mathcal{D}$ may be any domain.

A *weighted value* is a pair $(val, \alpha) \in \mathcal{D} \times (0, 1]$, where α is a *weight* associated with a value *val*. We associate a weight of 1 to a whole value, so, for example, $(val_i, 1/2)$ is half of node i's value. A set of weighted values is called a *cluster*:

Definition 4 (Cluster)**.** *A cluster c is a set of weighted values with unique values. The cluster's weight, $c.weight$, is the sum of the value weights:*

$$c.weight \triangleq \sum_{(val,\alpha)\in c} \alpha \quad .$$

A cluster may be *split* into two new clusters, each consisting of the same values as the original cluster, but associated with half their original weights. Similarly, multiple clusters may be *merged* to form a new one, consisting of the union of their values, where each value is associated with the sum of its weights in the original clusters.

A cluster can be concisely described by a *summary* in a domain $\mathcal{S}$, using a function f that maps clusters to their summaries: $f : (\mathcal{D} \times (0, 1])^* \to \mathcal{S}$. The domain $\mathcal{S}$ is a pseudo-metric space (like metric, except the distance between distinct points may be zero), with a distance function $d_S : \mathcal{S}^2 \to \mathbb{R}$. *For example, in the centroids algorithm, the function f calculates the weighted average of samples in a cluster.*

A cluster c may be partitioned into several clusters, each holding a subset of its values and summarized separately[1]. The set of weighted summaries of these clusters is called a *clustering* of c. Weighted values in c may be split among clusters, so that different clusters contain portions of a given value. The sum of weights associated with a value *val* in all clusters is equal to the sum of weights associated with *val* in c. Formally:

Definition 5 (Clustering)**.** *A clustering C of a cluster c into J clusters $\{c_j\}_{j=1}^J$ is the set of weighted summaries of these clusters: $C = \{(f(c_j), c_j.weight)\}_{j=1}^J$ s.t.*

$$\forall val : \sum_{(val,\alpha)\in c} \alpha \;=\; \sum_{j=1}^{J} \left(\sum_{(val,\alpha)\in c_j} \alpha \right) .$$

A clustering of a value set $\{val_j\}_{j=1}^l$ is a clustering of the cluster $\{(val_j, 1)\}_{j=1}^l$.

[1] Note that partitioning a cluster is different from splitting it, because, when a cluster is split, each part holds the same values.

The number of clusters in a clustering is bounded by a system parameter k.

A clustering algorithm strives to partition the samples into clusters in a way that optimizes some criterion, for example, minimizes some distance metric among values assigned to the same cluster (as in k-means). In this work, we are not concerned with the nature of this criterion, and leave it up to the application to specify the choice thereof.

A clustering algorithm maintains at every time t a clustering $clustering_i(t)$, yielding an infinite series of clusterings. For such a series, we define convergence:

Definition 6 (Clustering convergence)**.** *A series of clusterings*

$$\left\{\left\{(f(c_j(t)), c_j(t).weight)\right\}_{j=1}^{J_t}\right\}_{t=1}^{\infty}$$

converges to a destination clustering, which is a set of l clusters $\{dest_x\}_{x=1}^{l}$, if for every $t \in 0, 1, 2, \cdots$ there exists a mapping ψ_t between the J_t clusters at time t and the l clusters in the destination clustering $\psi_t : \{1, \cdots, J_t\} \to \{1, \cdots, l\}$, such that:

1. *The summaries converge to the clusters to which they are mapped by ψ_t:*

$$\max_j \left\{d_S(f(c_j(t)), f(dest_{\psi_t(j)}))\right\} \xrightarrow{t\to\infty} 0 \ .$$

2. *For each cluster x in the destination clustering, the relative amount of weight in all clusters mapped to x converges to x's relative weight in the clustering:*

$$\forall 1 \leq x \leq l : \frac{\sum_{\{j|\psi_t(j)=x\}} c_j(t).weight}{\sum_{j=1}^{J_t} c_j(t).weight} \xrightarrow{t\to\infty} \frac{dest_x.weight}{\sum_{y=1}^{l} dest_y.weight} \ .$$

We are now ready to define the problem addressed in this chapter, where a set of nodes strive to learn a common clustering of their inputs. As previous works on aggregation in sensor networks [72, 88, 19], we define a converging problem, where nodes continuously produce outputs, and these outputs converge to such a common clustering.

Definition 7 (Distributed clustering)**.** *Each node i takes an input val_i at time 0 and maintains a clustering $\texttt{clustering}_i(t)$ at each time t, s.t. there exists a clustering of the input values $\{val_i\}_{i=1}^{n}$ to which the clustering in all nodes converge.*

4.3 Generic Clustering Algorithm

We now present our generic algorithm that solves the Distributed Clustering Problem. At each node, the algorithm builds a clustering, which converges over time to one that describes all input values of all nodes. In order to avoid excessive bandwidth and storage consumption, the algorithm maintains clusterings as weighted summaries of clusters, and not the actual sets of weighted values. By slight abuse of terminology, we refer by the term cluster to both a set of weighted values c, and its summary–weight pair $(c.\texttt{summary}, c.\mathit{weight})$.

A node starts with a clustering of its own input value. It then periodically splits its clustering into two new ones, which have the same summaries but half the weights of the originals; it sends one clustering to a neighbor, and keeps the other. Upon receiving a clustering from a neighbor, a node merges it with its own, according to an application-specific merge rule. The algorithm thus progresses as a series of merge and split operations.

We begin with an illustrative example in Section 4.3.1 which summarizes clusters as their *centroids* — the averages of their weighted values.

Then, in Section 4.3.2, we present the generic distributed clustering algorithm. It is instantiated with a domain $\mathcal{S}$ of summaries used to describe clusters, and with application-specific functions that manipulate summaries and make clustering decisions. We use the centroid algorithm as an example instantiation.

In Section 4.3.3, we enumerate a set of requirements on the functions the algorithm is instantiated with. We then show that in any instantiation of the generic algorithm with functions that meet these requirements, the weighted summaries of clusters are the same as those we would have obtained, had we applied the algorithm's operations on the original clusters, and then summarized the results.

4.3.1 Example — Centroids

We begin by considering the example case of centroid summaries, where a cluster is described by its centroid and weight $(c.\mu, c.w)$.

Initially, the centroid is the sensor's read value, and the weight is 1, so at node i the cluster is $(val_i, 1)$. A node occasionally sends half of its clusters to a neighbor. A node with clusters $(c_1.\mu, c_1.w), (c_2.\mu, c_2.w)$ would keep $(c_1.\mu, \frac{1}{2}c_1.w), (c_2.\mu, \frac{1}{2}c_2.w)$ and send to a neighbor a message with the pair $(c_1.\mu, \frac{1}{2}c_1.w), (c_2.\mu, \frac{1}{2}c_2.w)$. The neighbor receiving the message will consider the received clusters with its own, and merge clusters with close centroids. Merge is performed by calculating the weighted sum. For example, the merge

Algorithm 5: Generic Distributed Data Clustering Algorithm. Dashed frames show auxiliary code.

1 **state**
2 $clustering_i$, initially $\{(\texttt{valToSummary}(val_i), 1, e_i)\}$
3 Periodically do **atomically**
4 Choose $j \in neighbors_i$ (Selection has to ensure fairness)
5 $old \leftarrow clustering_i$
6 $clustering_i \leftarrow \bigcup_{c \in old}\{(c.\texttt{summary}, half(c.weight), \frac{half(c.weight)}{c.weight} \cdot c.aux)\}$
7 send $(j, \bigcup_{c \in old}\{(c.\texttt{summary}, c.weight - half(c.weight), \left(1 - \frac{half(c.weight)}{c.weight}\right) \cdot c.aux)\})$
8 Upon receipt of *incoming* do **atomically**
9 $bigSet \leftarrow clustering_i \cup incoming$
10 $M \leftarrow \texttt{partition}(bigSet)$ (The function `partition` returns a set of cluster sets)
11 $clustering_i \leftarrow$

$$\bigcup_{x=1}^{|M|}\left\{\left(\texttt{mergeSet}\left(\bigcup_{c \in M_x}\{(c.\texttt{summary}, c.weight)\}\right), \sum_{c \in M_x} c.weight, \sum_{c \in M_x} c.aux\right)\right\}$$

12 **function** $half(\alpha)$
13 return the multiple of q which is closest to $\alpha/2$.

of two clusters $(c.\mu, c.w)$ and $(d.\mu, d.w)$ is

$$\left(\frac{\frac{1}{2}c.w \cdot c.\mu + d.w \cdot d.\mu}{\frac{1}{2}c.w + d.w}, \frac{1}{2}c.w + d.w\right) .$$

We now proceed to describe the generic algorithm.

4.3.2 Algorithm

The algorithm for node i is shown in Algorithm 5 (at this stage, we ignore the parts in dashed frames). The algorithm is generic, and it is instantiated with the summary domain $\mathcal{S}$ and the functions `valToSummary`, `partition` and `mergeSet`. *The functions of the centroids example are given in Algorithm 6. The summary domain $\mathcal{S}$ in this case is the same as the value domain, i.e., $\mathbb{R}^d$.*

Initially, each node produces a clustering with a single cluster, based on the single value it has taken as input (Line 2). The weight of this cluster is 1, and its summary is produced by the function $\texttt{valToSummary} : \mathcal{D} \to \mathcal{S}$. *In the centroids example, the initial summary is the input value (Algorithm 6,* `valToSummary` *function).*

A node occasionally sends data to a neighbor (Algorithm 5, Lines 3–7): It first splits its clustering into two new ones. For each cluster in the original clustering, there is a matching cluster in each of the new ones, with the

same summary, but with approximately half the weight. Weight is quantized, limited to multiples of a system parameter q ($q, 2q, 3q, \dots$). This is done in order to avoid a scenario where it takes infinitely many transfers of infinitesimal weight to transfer a finite weight from one cluster to another (Zeno effect). We assume that q is small enough to avoid quantization errors: $q \ll \frac{1}{n}$. In order to respect the quantization requirement, the weight is not multiplied by exactly 0.5, but by the closest factor for which the resulting weight is a multiple of q (function *half* in Algorithm 5). One of the clusters is attributed the result of *half* and the other is attributed the complement, so that the sum of weights is equal to the original, and system-wide conservation of weight is maintained. Note that despite the weight quantization, values and summaries may still be continuous, therefore convergence may still be continuous.

If the communication topology is dense, it is possible to perform scalable random peer sampling [85], even under message loss [58], in order to achieve data propagation guarantees.

The node then keeps one of the new clusterings, replacing its original one (Line 6), and sends the other to some neighbor j (Line 7). The selection of neighbors has to ensure fairness in the sense that in an infinite run, each neighbor is chosen infinitely often; this can be achieved, e.g., using round robin. Alternatively, the node may implement gossip communication patterns: It may choose a random neighbor and send data to it (push), or ask it for data (pull), or perform a bilateral exchange (push-pull).

When a message with a neighbor's clustering reaches the node, an event handler (Lines 8–11) is called. It first combines the two clusterings of the nodes into a set *bigSet* (Line 9). Then, an application-specific function `partition` divides the clusters in *bigSet* into sets $M = \{M_x\}_{x=1}^{|M|}$ (Line 10). The clusters in each of the sets in M are merged into a single cluster, together forming the new clustering of the node (Line 11). The summary of each merged cluster is calculated by another application-specific function, `mergeSet`, and its weight is the sum of weights of the merged clusters.

To conform with the restrictions of k and q, the partition function must guarantee that (1) $|M| \leq k$; and (2) no M_x includes a single cluster of weight q (that is, every cluster of weight q is merged with at least one other cluster).

Note that the parameter k forces lossy compression of the data, since merged values cannot later be separated. At the beginning, only a small number of data values is known to the node, so it performs only a few (easy) clustering decisions. As the algorithm progresses, the number of values described by the node's clustering increases. By then, it has enough knowledge of the data set, so as to perform correct clustering decisions, and achieve a

Algorithm 6: Centroid Functions

function `valToSummary`(*val*)
  return *val*

function `mergeSet`(`clusters`)

  return $\left(\sum_{\substack{(avg,\, m) \in \\ \texttt{clusters}}} m \right)^{-1} \times \sum_{\substack{(avg,\, m) \in \\ \texttt{clusters}}} m \cdot avg$

function `partition`(*bigSet*)
  $M \leftarrow \{\{c\}\}_{c \in bigSet}$
  If there are sets in M whose clusters' weights are q, then unify them arbitrarily with others
  while $|M| > k$ **do**
    let M_x and M_y be the (different) cluster sets in M whose centroids are closest
    $M \leftarrow M \setminus \{M_x, M_y\} \cup (M_x \cup M_y)$
  return M

high compression ratio without losing valuable data.

In the centroids algorithm, the summary of the merged set is the weighted average of the summaries of the merged clusters, calculated by the implementation of `mergeSet` *shown in Algorithm 6. Merging decisions are based on the distance between cluster centroids. Intuitively, it is best to merge close centroids, and keep distant ones separated. This is done greedily by* `partition` *(shown in Algorithm 6) which repeatedly merges the closest sets, until the k bound is reached. For $k = 1$, the algorithm is reduced to push-sum.*

4.3.3 Auxiliaries and Instantiation Requirements

For the algorithm to perform a meaningful and correct clustering of the data, its functions must respect a set of requirements. In Section 4.3.3.a we specify these requirements and in Section 4.3.3.b we show that the centroids algorithm described above meets these requirements. In Section 4.3.3.c we prove that these requirements ensure that the summaries described by the algorithm indeed represent clusters.

4.3.3.a Instantiation Requirements

To phrase the requirements, we describe a cluster in $(\mathcal{D}, (0, 1])^*$ as a vector in the *Mixture Space* — the space $\mathbb{R}^n$ (n being the number of input values), where each coordinate represents one input value. A cluster is described in this space as a vector whose j'th component is the weight associated with val_j in that cluster. For a given input set, a vector in the mixture space precisely describes a cluster. We can therefore redefine f as a mapping from

mixture space vectors of clusters to cluster summaries, according to the input set $I \in \mathcal{D}^n$. We denote this mapping $f_I : \mathbb{R}^n \to \mathcal{S}$.

We define the distance function $d_M : (\mathbb{R}^n)^2 \to \mathbb{R}$ between two vectors in the mixture space to be the angle between them. Clusters consisting of similar weighted values are close in the mixture space (according to d_M). Their summaries should be close in the summary space (according to d_S), with some scaling factor ρ. Simply put — clusters consisting of similar values (i.e., close in d_M) should have similar summaries (i.e., close in d_S). Formally:

R1 For any input value set I,

$$\exists \rho \ : \ \forall v_1, v_2 \ \in \ (0,1]^n \ : \ d_S(f_I(v_1), f_I(v_2)) \ \leq \ \rho \cdot d_M(v_1, v_2).$$

In addition, operations on summaries must preserve the relation to the clusters they describe. Intuitively, this means that operating on summaries is similar to performing the various operations on the value set, and then summarizing the results.

R2 Initial values are mapped by f_I to their summaries:

$$\forall i, 1 \leq i \leq n : \texttt{valToSummary}(val_i) = f_I(e_i).$$

R3 Summaries are oblivious to weight scaling:

$$\forall \alpha > 0, v \in (0,1]^n : f_I(v) = f_I(\alpha v).$$

R4 Merging a summarized description of clusters is equivalent to merging these clusters and then summarizing the result[2]:

$$\texttt{mergeSet}\left(\bigcup_{v \in V}(\{f_I(v), \|v\|_1)\}\right) = f_I\left(\sum_{v \in V} v\right).$$

4.3.3.b The Centroids Case

We show now that the centroids algorithm respects the requirements. Recall that f_I in this case is the weighted average of the samples, and let d_S be the L^2 distance between centroids. We show that the requirements are respected.

Claim 1. *For the centroids algorithm, as described in Algorithm 6, the requirements R1–R4 are respected.*

[2]Denote by $\|v\|_p$ the L^p norm of v.

Proof. Denote by $\delta_{\max}$ the maximal L^2 distance between values, and let $\rho \triangleq 2\delta_{\max}\sqrt{n}$. Let $\tilde{v}$ be the L^2 normalized vector v. We show that R1 holds with this ρ.

$$
\begin{aligned}
& d_S(f_I(v_1), f_I(v_2)) \overset{(1)}{=} \\
& d_S(f_I(\tilde{v}_1), f_I(\tilde{v}_2)) \overset{(2)}{\leq} \\
& \delta_{\max}\|\tilde{v}_1 - \tilde{v}_2\|_1 \overset{(3)}{\leq} \\
& \delta_{\max}\sqrt{n}\|\tilde{v}_1 - \tilde{v}_2\|_2 \overset{(4)}{\leq} \\
& 2\delta_{\max}\sqrt{n} \cdot d_M(\tilde{v}_1, \tilde{v}_2) = \rho \cdot d_M(v_1, v_2)
\end{aligned}
$$

(1) By the definition of f_I and d_S.

(2) Each value may contribute at most $\delta_{\max}$ to the coordinate difference.

(3) The L^1 norm is smaller than $\sqrt{n}$ times the L^2 norm, so a factor of $\sqrt{n}$ is added.

(4) Recall that d_M is the angle between the two vectors. The L^2 difference of normalized vectors in $[0, 1)^n$ is smaller than twice the angle between them (Law of Cosines).

It is readily seen that requirements R2–R4 also hold. □

4.3.3.c Correctness of Auxiliary Algorithm

Returning to the generic case, we show that the weighted summaries maintained by the algorithm to describe clusters that are merged and split, indeed do so. To do that, we define an auxiliary algorithm. This is an extension of Algorithm 5 with the auxiliary code in the dashed frames. Clusters are now triplets, containing, in addition to the summary and weight, the cluster's mixture space vector *c.aux*.

At initialization (Line 2), the auxiliary vector at node i is e_i (a unit vector whose i'th component is 1). When splitting a cluster (Lines 6–7), the vector is factored by about 1/2 (the same ratio as the weight). When merging a set of clusters, the mixture vector of the result is the sum of the original clusters' vectors (Line 11).

The following lemma shows that, at all times, the summary maintained by the algorithm is indeed that of the cluster described by its mixture vector:

Lemma 9 (Correctness of auxiliary algorithm)**.** *The generic algorithm, instantiated by functions satisfying R2–R4, maintains the following invariant: For any cluster c either in a node's clustering ($c \in \texttt{clustering}_i$) or in transit in a communication channel, the following two equations hold:*

$$f_I(c.aux) = c.\texttt{summary} \tag{4.1}$$
$$\|c.aux\|_1 = c.weight \tag{4.2}$$

Proof. By induction on the global states of the system.

Basis Initialization puts at time 0, at every node i the auxiliary vector e_i, a weight of 1, and the summary `valToSummary` of value i. Requirement R2 thus ensures that Equation 4.1 holds in the initial state, and Equation 4.2 holds since $\|e_i\| = 1$. Communication channels are empty.

Assumption At time $j-1$ the invariant holds.

Step Transition j may be either send or receive. Each of them removes clusters from the set, and produces a cluster or two. To prove that at time j the invariant holds, we need only show that in both cases the new cluster(s) maintain the invariant.

Send We show that the mapping holds for the kept cluster c_{keep}. A similar proof holds for the sent one c_{send}. Proof of Equation 4.1:

$$\begin{aligned}
&c_{\text{keep}}.\texttt{summary} \overset{\text{line 6}}{=} \\
&= c.\texttt{summary} \overset{\substack{\text{induction}\\ \text{assumption}}}{=} \\
&= f_I(c.aux) \overset{\text{R3}}{=} \\
&= f_I\left(\frac{half(c.weight)}{c.weight} \cdot c.aux\right) \overset{\substack{\text{auxiliary}\\ \text{line 6}}}{=} \\
&= f_I(c_{\text{keep}}.aux)
\end{aligned}$$

Proof of Equation 4.2:

$$
\begin{aligned}
& c_{\text{keep}}.\mathit{weight} \overset{\text{line 6}}{=} \\
& = \mathit{half}(c.\mathit{weight}) = \\
& = \frac{\mathit{half}(c.\mathit{weight})}{c.\mathit{weight}} \cdot c.\mathit{weight} \overset{\substack{\text{induction}\\ \text{assumption}}}{=} \\
& = \frac{\mathit{half}(c.\mathit{weight})}{c.\mathit{weight}} \cdot \|c.\mathit{aux}\|_1 = \\
& = \left\| \frac{\mathit{half}(c.\mathit{weight})}{c.\mathit{weight}} \cdot c.\mathit{aux} \right\|_1 \overset{\substack{\text{auxiliary}\\ \text{line 6}}}{=} \\
& = \|c_{\text{keep}}.\mathit{aux}\|_1
\end{aligned}
$$

Receive We prove that the mapping holds for each of the m produced clusters. Each cluster c_x is derived from a set M_x. Proof of Equation 4.1:

$$
\begin{aligned}
& c_x.\texttt{summary} \overset{\text{line 11}}{=} \\
& = \texttt{mergeSet}\left(\bigcup_{c \in M_x} \{(c.\texttt{summary}, c.\mathit{weight})\} \right) \overset{\substack{\text{induction}\\ \text{assumption}}}{=} \\
& = \texttt{mergeSet}\left(\bigcup_{c \in M_x} \{(f_I(c.\mathit{aux}), \|c.\mathit{aux}\|_1)\} \right) \overset{\text{R4}}{=} \\
& = f_I\left(\sum_{c \in M_x} c.\mathit{aux} \right) \overset{\substack{\text{auxiliary}\\ \text{line 11}}}{=} \\
& = f_I\left(c_x.\mathit{aux}\right)
\end{aligned}
$$

Proof of Equation 4.2:

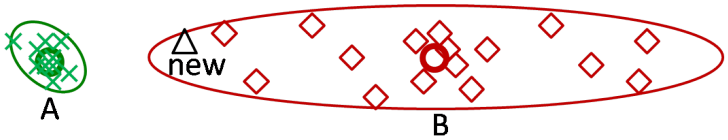
$$
\begin{aligned}
& c_x.\mathit{weight} \overset{\text{line 11}}{=} \\
& = \sum_{c \in M_x} c.\mathit{weight} \overset{\substack{\text{induction}\\ \text{assumption}}}{=} \\
& = \sum_{c \in M_x} \|c.\mathit{aux}\|_1 = \\
& = \left\| \sum_{c \in M_x} c.\mathit{aux} \right\|_1 \overset{\substack{\text{auxiliary}\\ \text{line 11}}}{=} \\
& = \|c_x.\mathit{aux}\|_1
\end{aligned}
$$

□

4.4 Gaussian Clustering

When clustering value sets from a metric space, the centroids solution is seldom sufficient. Consider the example shown in Figure 4.1, where we need to associate a new value with one of two existing clusters. Figure 4.1a shows the information that the centroids algorithm has for clusters A and B, and a new value. The algorithm would associate the new value to cluster A, on account of it being closer to its centroid. However, Figure 4.1b shows the set of values that produced the two clusters. We see that it is more likely that the new value in fact belongs to cluster B, since it has a much larger variance.

(a) Centroids and a new value



(b) Gaussians and a new value

Figure 4.1: Associating a new value when clusters are summarized (a) as centroids and (b) as Gaussians.

The field of machine learning suggests the heuristic of clustering data using a Gaussian Mixture (a weighted set of normal distributions), allowing for a rich and accurate description of multivariate data. Figure 4.1b illustrates the summary employed by GM: An ellipse depicts an equidensity line of the Gaussian summary of each cluster. Given these Gaussians, one can easily classify the new value correctly.

We present in Section 4.4.1 the GM algorithm — a new distributed clustering algorithm, implementing the generic one by representing clusters as Gaussians, and clusters as Gaussian Mixtures. Contrary to the classical ma-

chine learning algorithms, ours performs the clustering without collecting the data in a central location. Nodes use the popular machine learning tool of Expectation Maximization to make clustering decisions (Section 4.4.1). A taste of the results achieved by our GM algorithm is given in Section 4.4.2 via simulation. It demonstrates the clustering of multidimensional data and more. Note that due to the heuristic nature of EM, the only possible evaluation of our algorithm's quality is empirical.

4.4.1 Generic Algorithm Instantiation

The summary of a cluster is a tuple (μ, σ), comprised of the average of the weighted values in the cluster $\mu \in \mathbb{R}^d$ (where $\mathcal{D} = \mathbb{R}^d$ is the value space), and their covariance matrix $\sigma \in \mathbb{R}^{d \times d}$. Together with the weight, a cluster is described by a weighted Gaussian, and a clustering consists of a weighted set of Gaussians, or a *Gaussian Mixture*.

Let $v = (v_1, \cdots, v_n)$ be an auxiliary vector; we denote by $\tilde{v}$ a normalized version thereof:

$$\tilde{v} = \frac{v}{\sum_{j=1}^{s} v_j} \ .$$

Recall that v_j represents the weight of val_j in the cluster. The centroid $\mu(v)$ and covariance matrix $\sigma(v)$ of the weighted values in the cluster are calculated as follows:

$$\mu(v) = \sum_{j=1}^{n} \tilde{v}_j \cdot val_j \quad \text{, and}$$

$$\sigma(v) = \frac{1}{1 - \sum_{k=1}^{n} \tilde{v}_k^2} \sum_{j=1}^{n} \tilde{v}_j (val_j - \mu)(val_j - \mu)^T \ .$$

We use them to define the mapping f_I from the mixture space to the summary space:

$$f_I(v) = (\mu(v), \sigma(v)) \ .$$

Note that the use of the normalized vector $\tilde{v}$ makes both $\mu(v)$ and $\sigma(v)$ invariant under weight scaling, thus fulfilling Requirement R3.

We define d_S as in the centroids algorithm. Namely, it is the L^2 distance between the centroids of clusters. This fulfills requirement R1 (see Section 4.3.3.b).

The function `valToSummary` returns a cluster with an average equal to *val*, a zero covariance matrix, and a weight of 1. Requirement R2 is trivially satisfied.

To describe the function `mergeSet` we use the following definitions: Denote the weight, average and covariance matrix of cluster x by w_x, μ_x and σ_x, respectively. Given the summaries and weights of two clusters a and b, one can calculate the summary of a cluster c created by merging the two:

$$\mu_c = \frac{w_a}{w_a + w_b}\mu_a + \frac{w_b}{w_a + w_b}\mu_b$$
$$\sigma_c = \frac{w_a}{w_a + w_b}\sigma_a + \frac{w_b}{w_a + w_b}\sigma_b + \frac{w_a \cdot w_b}{(w_a + w_b)^2} \cdot (\mu_a - \mu_b) \cdot (\mu_a - \mu_b)^T$$

This merging function maintains the average and covariance of the original values [97], therefore it can be iterated to merge a set of summaries and implement `mergeSet` in a way that conforms to R4.

Expectation Maximization Partitioning

To complete the description of the GM algorithm, we now explain the `partition` function. When a node has accumulated more than k clusters, it needs to merge some of them. In principle, it would be best to choose clusters to merge according to Maximum Likelihood, which is defined in this case as follows: We denote a Gaussian Mixture of x Gaussians *x-GM*. Given a too large set of $l \geq k$ clusters, an *l-GM*, the algorithm tries to find the *k-GM* probability distribution for which the *l-GM* has the maximal likelihood. However, computing Maximum Likelihood is NP-hard. We therefore instead follow common practice and approximate it with the *Expectation Maximization* algorithm [82].

Our goal is to re-classify GM_{old}, an *l-GM* with $l > k$, to GM_{new}, a *k-GM*. Denote by V the d dimensional space in which the distributions are defined. Denote by $f_X(v)$ the probability density at point v of distribution X. If X is a weight distribution such as a Gaussian mixture, it is normalized s.t. it constitutes a probability density.

The *likelihood* that the samples concisely described by GM_{old} are the result of the probability distribution described by (the normalized) GM_{new} is:

$$L = \sum_{c \in GM_{\text{new}}} \sum_{g \in GM_{\text{old}}} \left(\int_{v \in V} w_c f_c(v) \cdot w_g f_g(v) dv \right) .$$

The merge employs the Expectation Maximization algorithm to approximate Maximum Likelihood. It arbitrarily groups the clusters in GM_{old} into k sets, and merges each set into a single Gaussian, forming a *k-GM* GM_{new}. It then alternately regroups GM_{old}'s clusters to maximize their likelihood w.r.t. GM_{new}, and recalculates GM_{new} according to this grouping. This process is repeated until convergence.

4.4.2 Simulation Results

Due to the heuristic nature of the Gaussian Mixture clustering and of EM, the quality of their results is typically evaluated experimentally. In this section, we briefly demonstrate the effectiveness of our GM algorithm through simulation. First, we demonstrate the algorithm's ability to cluster multi-dimensional data, which could be produced by a sensor network. Then, we demonstrate a possible application using the algorithm to calculate the average while removing erroneous data reads and coping with node failures. This result also demonstrates the convergence speed of the algorithm.

In both cases, we simulate a fully connected network of 1,000 nodes. Like previous works [44, 59], we measure progress in rounds, where in each round each node sends a clustering to one neighbor. Nodes that receive clusterings from multiple neighbors accumulate all the received clusters and run EM once for the entire set.

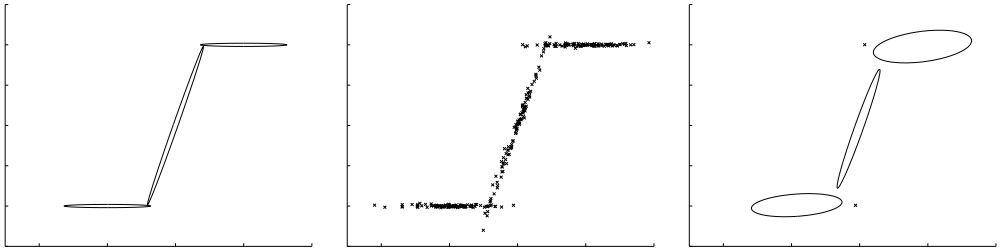
4.4.2.a Multidimensional Data Clustering

As an example input, we use data generated from a set of three Gaussians in $\mathbb{R}^2$. Values are generated according to the distribution shown in Figure 4.2a, where the ellipses are equidensity contours of normal distributions. This input might describe temperature readings taken by a set of sensors positioned on a fence located by the woods, and whose right side is close to a fire outbreak. Each value is comprised of the sensor's location x and the recorded temperature y. The generated input values are shown in Figure 4.2b. We run the GM algorithm with this input until its convergence; $k = 7$ and q is set by floating point accuracy.

The result is shown in Figure 4.2c. The ellipses are equidensity contours, and the x's are singleton clusters (with zero variance). This result is visibly a usable estimation of the input data.

4.4.2.b Robustness

Erroneous samples removal As an example application, we use the algorithm to calculate a statistically robust average. We consider a sensor network of 1,000 sensors reading values in $\mathbb{R}^2$. Most of these values are sampled from a given Gaussian distribution and we wish to calculate their average. Some values, however, are erroneous, and are unlikely to belong to this distribution. They may be the result of a malfunctioning sensor, or of a sensing error, e.g., an animal sitting on an ambient temperature sensor. These values should be removed from the statistics.



(a) Distribution (b) Values (c) Result

Figure 4.2: Gaussian Mixture clustering example. The three Gaussians in Figure 4.2a were used to generate the data set in Figure 4.2b. The GM algorithm produced the estimation in Figure 4.2c.

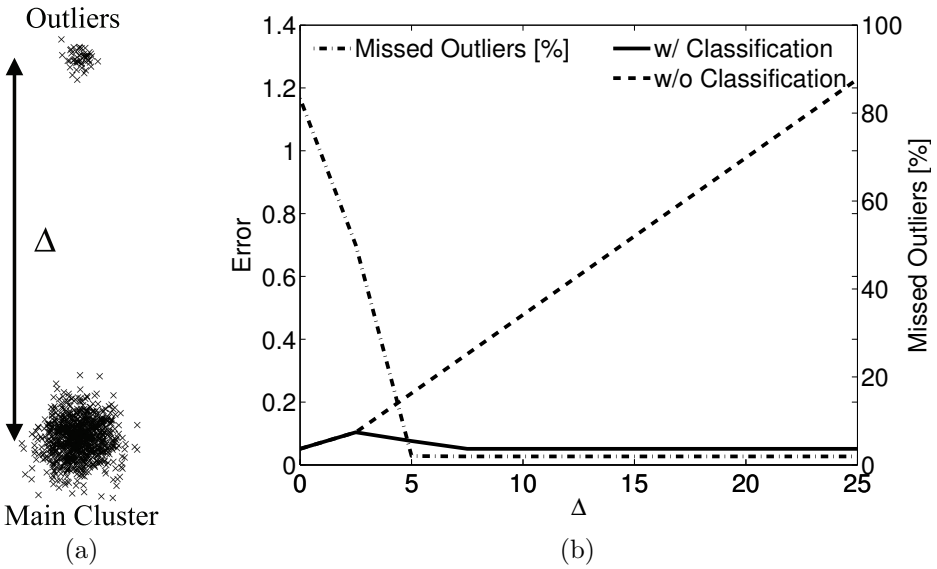
We use 950 values from the standard normal distribution, i.e., with a mean $(0, 0)$ and a unit covariance matrix I. Fifty additional values are distributed normally with covariance matrix $0.1 \cdot I$ and mean $(0, \Delta)$, with Δ ranging between 0 and 25. The distribution of all values is illustrated in Figure 4.3a.

For each value of Δ, the protocol is run until convergence. We use $k = 2$, so that each node has at most 2 clusters at any given time — hopefully one for good values and one for the erroneous values.

The results are shown in Figure 4.3b. The dotted line shows the average weight ratio belonging to erroneous samples yet incorrectly assigned to the good cluster. Erroneous samples are defined to be values with probability density lower than $f_{min} = 5 \times 10^{-5}$ (for the standard normal distribution). The other two lines show the error in calculating the mean, where error is the average over all nodes of the distance between the estimated mean and the true mean $(0, 0)$. The solid line shows the result of our algorithm, which removes erroneous samples, while the dashed line shows the result of regular average aggregation, which does not.

We see that when the erroneous samples are close to the good values, the number of misses is large — the proximity of the clusters makes their separation difficult. However, due to the small distance, this mistake hardly influences the estimated average. As the erroneous samples' mean moves further from the true mean, their identification becomes accurate and their influence is nullified.

Note that even for large Δ's, a certain number of erroneous samples is still missed. These are values from the good distribution, relatively close to the main cluster, yet with probability density lower than f_{min}. The protocol mistakenly considers these to be good values. Additionally, around $\Delta = 5$ the miss rate is dropped to its minimum, yet the robust error does not. This



(a) (b)

Figure 4.3: Effect of the separation of erroneous samples on the calculation of the average: A 1,000 values are sampled from two Gaussian distributions (a). As the erroneous samples' distribution moves away from the good one, the regular aggregation error grows linearly (b). However, once the distance is large enough, our protocol can remove the erroneous samples, which results in an accurate estimation of the mean.

is due to the fact that bad values are located close enough to the good mean so that their probability density is higher than f_{min}. The protocol mistakes those to belong to f_G and allows them to influence the mean. That being said, for all Δ's, the error remains small, confirming the conventional wisdom that "clustering is either easy or not interesting".

Crash robustness and convergence speed We next examine how crash failures impact the results obtained by our protocol. Figure 4.4 shows that the algorithm is indifferent to crashes of nodes. The source data is similar to the one above, with $\Delta = 10$. After each round, each node crashes with probability 0.05. We show the average node estimation error of the mean in each round. As we have seen above, our protocol achieves a lower error then the regular one.

Figure 4.4 also demonstrates the convergence speed of our algorithm. With and without crashes, the convergence speed of our algorithm is equivalent to that of the regular average aggregation algorithm.

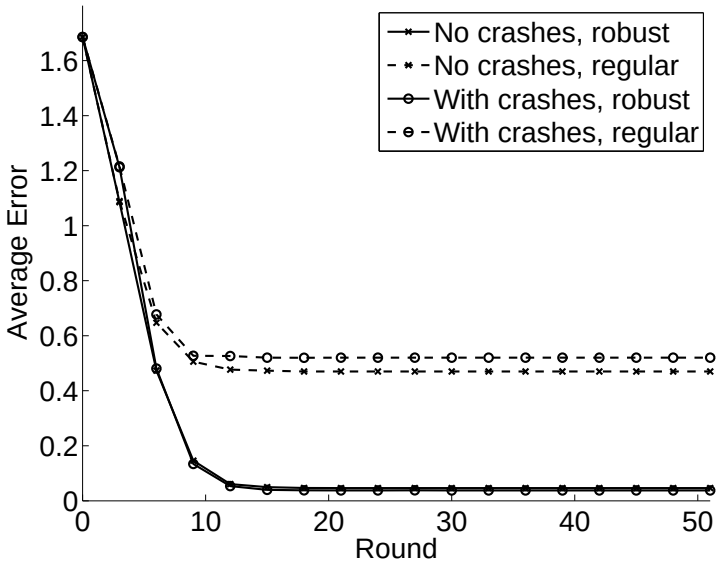


Figure 4.4: The effect of crashes on convergence speed and on the accuracy of the mean.

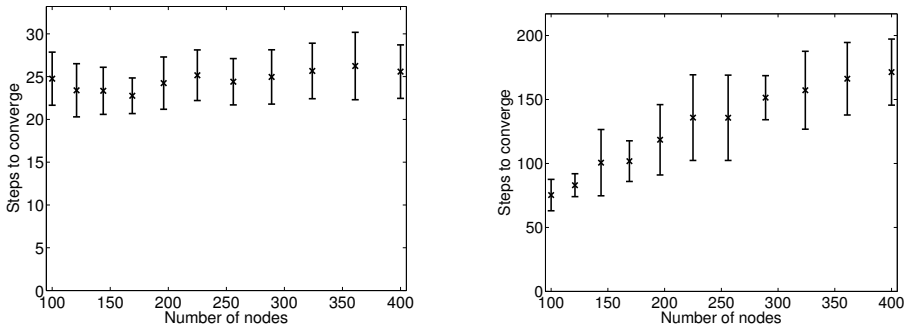
4.4.2.c Scalability

To evaluate convergence time we measure the number of rounds until the estimations at the different nodes are the same. The samples are taken from a Gaussian mixture of two Gaussians of the same weight with variance 1 at distance 5 from each other. Since there is no scalar convergence value, the $\varepsilon - \delta$ measure which is used, e.g., for analysing Push-Sum, is not applicable to this scenario. Instead, we use the Kolmogorov-Smirnov (KS) statistic as the measure of difference between two distributions[3]. For a range of network sizes, we let the algorithm run until the maximal KS-statistic between the estimations of any pair of nodes[4] falls below an arbitrary threshold of 0.01. The results for a complete topology and a grid topology (with samples taken independently of the grid coordinates) are shown in figures 4.5a and 4.5b, respectively. For each network size we show the average convergence time with the 95% confidence interval.

As expected, the scalability in a grid topology is worse than in a complete topology. The trends shown in these figures match those calculated by Boyd et al. [21] for the Push-Sum algorithm.

[3]The Kolmogorov-Smirnov statistic for two distributions is the maximal difference between their cumulative distribution functions

[4]To shorten simulation time, we calculate the statistics for $4n$ random pairs of nodes.



(a) (b)

Figure 4.5: Convergence time of the distributed clustering algorithm as a function of the number of nodes (a) in a fully connected topology and (b) in a grid topology.

4.5 Convergence Proof

We now prove that the generic algorithm presented in Section 4.3 solves the distributed clustering problem. To prove convergence, we consider the pool of all the clusters in the system, at all nodes and communication channels. This pool is in fact, at all times, a clustering of the set of all input values. In Section 4.5.1 we prove that the pool of all clusters converges, i.e., roughly speaking, it stops changing. Then, in Section 4.5.2, we prove that the clusterings in all nodes converge to the same destination.

4.5.1 Collective Convergence

In this section, we ignore the distributive nature of the algorithm, and consider all the clusters in the system (at both processes and communication channels) at time t as if they belonged to a single multiset $pool(t)$. A run of the algorithm can therefore be seen as a series of splits and merges of clusters.

To argue about convergence, we first define the concept of cluster descendants. Intuitively, for $t_1 \leq t_2$, a cluster $c_2 \in pool(t_2)$ is a descendant of a cluster $c_1 \in pool(t_1)$ if c_2 is equal to c_1, or is the result of operations on c_1. Formally:

Definition 8 (Cluster genealogy)**.** *We recursively define the descendants of a cluster $c \in pool(t)$. First, at t, the descendant set is simply $\{c\}$. Next, consider $t_1 > t$.*

Assume the t_1'th operation in the execution is splitting (and sending) (Lines 3–7) a set of clusters $\{c_x\}_{x=1}^{l} \subset pool(t_1 - 1)$. This results in two

new sets of clusters, $\{c_x^1\}_{x=1}^l$ *and* $\{c_x^2\}_{x=1}^l$*, being put in* $pool(t_1)$ *instead of the original set. If a cluster* c_x *is a descendant of* c *at* $t_1 - 1$*, then the clusters* c_x^1 *and* c_x^2 *are descendants of* c *at* t_1*.*

Assume the t_1*'th operation is a (receipt and) merge (Lines 8–11), then some* m *(*$1 \leq m \leq k$*) sets of clusters* $\{M_x\}_{x=1}^m \subset pool(t_1 - 1)$ *are merged and are put in* $pool(t_1)$ *instead of the merged ones. For every* M_x*, if any of its clusters is a descendant of* c *at* $t_1 - 1$*, then its merge result is a descendant of* c *at* t_1*.*

By slight abuse of notation, we write $v \in pool(t)$ when v is the mixture vector of a cluster c, and $c \in pool(t)$; vector genealogy is similar to cluster genealogy.

We now state some definitions and the lemmas used in the convergence proof. We prove that, eventually, the descendants of each vector in the pool converge (normalized) to one destination. To do that, we investigate the angles between a vector v and the axes unit vectors. Note that all angles are between zero and $\pi/2$. For $i \in \{1, \ldots, d\}$, we call the angle between v and the i'th axis v's i'th *reference angle* and denote it by φ_i^v. We denote by $\varphi_{i,\max}(t)$ the maximal i'th reference angle over all vectors in the pool at time t:

$$\varphi_{i,\max}(t) \stackrel{\Delta}{=} \max_{v \in \texttt{pool}(t)} \varphi_i^v \ .$$

We now show that the i'th reference angle is monotonically decreasing for any $1 \leq i \leq n$. To achieve this, we use Lemma 10 which states that the sum of two vectors has an i'th reference angle not larger than the larger i'th reference angle of the two. Its proof is deferred to Appendix 4.A.

Lemma 10 (Decreasing reference angle)**.** *The sum of two vectors in the mixture space is a vector with a smaller* i*'th reference angle than the larger* i*'th reference angle of the two, for any* $1 \leq i \leq n$*.*

We are now ready to prove that the maximal reference angle is monotonically decreasing:

Lemma 11. *For* $1 \leq i \leq n$*,* $\varphi_{i,max}(t)$ *is monotonically decreasing.*

Proof. The pool changes in split and merge operations. In case of a split, the new vectors have the same angles as the split one, so $\varphi_{i,\max}$ is unchanged. In case of a merge, a number of vectors are replaced by their sum. This can be seen as the result of a series of steps, each of which replaces two vectors by their sum. The sum of two vectors is a vector with a no larger i'th reference angle than the larger of the i'th reference angles of the two (Lemma 10). Therefore, whenever a number of vectors are replaced by their sum, the maximal reference angle may either remain the same or decrease. □

Since the maximal reference angles are bounded from below by zero, Lemma 11 shows that they converge, and we can define

$$\hat{\varphi}_{i,\max} \triangleq \lim_{t\to\infty} \varphi_{i,\max}(t) \ .$$

By slight abuse of terminology, we say that the i'th reference angle of a vector $v \in \mathit{pool}(t)$ *converges* to φ, if for every ε there exists a time t', after which the i'th reference angles of all of v's descendants are in the ε neighborhood of φ.

We proceed to show that there exists a time after which the pool is partitioned into clusters, and the vectors from each cluster merge only with one another. Moreover, the descendants of all vectors in a cluster converge to the same reference angle. More specifically, we show that the vectors in the pool are partitioned into clusters by the algorithm according to the i'th reference angle their descendants converge to (for any $1 \leq i \leq n$). We further show that, due to the quantization of weight, a gap is formed between descendants that converge to the maximal reference angle, and those that do not, as those that do not remain within some minimum distance ε from $\hat{\varphi}_{i,\max}$.

Since the i'th maximal reference angle converges (Lemma 10), for every ε there exists a time after which there are always vectors in the ε neighborhood of $\hat{\varphi}_{i,\max}$. The weight (sum of L^1 norms of vectors) in this neighborhood changes over time, and due to the quantization of weight there exists a minimal weight q^i_ε such that for every time t there exists a time $t' > t$ when the weight in the neighborhood is q^i_ε.

The following observations immediately follow:

Observation 1. *For every $\varepsilon' < \varepsilon$, the relation $q^i_{\varepsilon'} \leq q^i_\varepsilon$ holds. Moreover, $q^i_\varepsilon - q^i_{\varepsilon'} = l \cdot q$ with $l \in \{0, 1, \cdots\}$.*

Observation 2. *There exists an ε such that for every $\varepsilon' < \varepsilon$, the minimal weight in the ε' neighborhood of $\hat{\varphi}_{i,max}$ is the same as for ε. That is, $q^i_\varepsilon = q^i_{\varepsilon'}$.*

The next lemma shows that vectors from different sides of the gap are never merged. Its proof is deferred to Appendix 4.B.

Lemma 12. *For any ε, there exists an $\varepsilon' < \varepsilon$ such that if a vector v_{out} lies outside the ε-neighborhood of $\hat{\varphi}_{i,max}$ (i.e., has a reference angle smaller than $\hat{\varphi}_{i,max} - \varepsilon$), and a vector v_{in} lies inside the ε'-neighborhood (i.e., has a reference angle larger than $\hat{\varphi}_{i,max} - \varepsilon'$), then their sum v_{sum} lies outside the ε' neighborhood.*

We are now ready to prove the that eventually the vectors are partitioned.

Lemma 13 (Cluster formation)**.** *For every $1 \leq i \leq n$, there exists a time t_i and a set of vectors*

$$V_{i,max} \subset pool(t_i)$$

s.t. the i'th reference angles of the vectors converge to $\hat{\varphi}_{i,max}$, and their descendants are merged only with one another.

Proof. For a given i, choose an ε such that for every $\varepsilon' < \varepsilon$ the minimal weights are the same: $q^i_\varepsilon = q^i_{\varepsilon'}$. Such an ε exists according to Observation 2.

According to Lemma 12, there exists an $\tilde{\varepsilon}$ s.t. the sum of a vector inside the $\tilde{\varepsilon}$ neighborhood and a vector outside the ε neighborhood is outside the $\tilde{\varepsilon}$ neighborhood. Choose such an $\tilde{\varepsilon}$.

Denote

$$v_{\text{in},\tilde{\varepsilon}} \triangleq \sum_{v' \in V_{\text{in},\tilde{\varepsilon}}} v' \ , \ v_{\text{out},\varepsilon} \triangleq \sum_{v' \in V_{\text{out},\varepsilon}} v' \ .$$

Since the $V_{\text{out},\varepsilon}$ vectors have reference angles outside the ε neighborhood, $v_{\text{out},\varepsilon}$ is also outside the ε neighborhood (Lemma 10). $v_{\text{in},\tilde{\varepsilon}}$ may either be inside the $\tilde{\varepsilon}$ neighborhood or outside it. If $v_{\text{in},\tilde{\varepsilon}}$ is inside the $\tilde{\varepsilon}$ neighborhood, then the sum v is outside the ε neighborhood, due to the choice of $\tilde{\varepsilon}$. If it is outside, then v is outside the $\tilde{\varepsilon}$ neighborhood (Lemma 10 again).

Choose a t_i s.t. $t_i > t_{\tilde{\varepsilon}}$ and at t_i the ε neighborhood contains a weight q_ε. Since $q_\varepsilon = q_{\tilde{\varepsilon}}$, the weight in the $\tilde{\varepsilon}$ neighborhood cannot be smaller than $q_{\tilde{\varepsilon}}$, therefore the weight is actually in the $\tilde{\varepsilon}$ neighborhood.

We now show that all operations after t_i keep the descendants of the vectors that were in the $\tilde{\varepsilon}$ neighborhood at t_i inside that neighborhood, and never mix them with the other vector descendants, all of which remain outside the ε neighborhood.

We prove by induction that the descendants of the vectors that were inside the $\tilde{\varepsilon}$ at t_i are always in this neighborhood, and the descendants of the vectors outside the ε neighborhood at t_i are always outside this neighborhood. The assumption holds at t_i. Assume it holds at t_j. If the step is a send operation (Lines 3–7), it does not change the angle of the descendants, therefore the claim holds at t_{j+1}. If the step is a receive operation (Lines 8–11), then vectors are merged. There is never a merger of vectors from both inside the $\tilde{\varepsilon}$ neighborhood and outside the ε neighborhood, since the result is outside the $\tilde{\varepsilon}$ neighborhood, leaving inside it a weight smaller than $q_{\tilde{\varepsilon}}$. Due to the same reason, the sum of vectors inside the $\tilde{\varepsilon}$ neighborhood is always inside this neighborhood. Finally, the sum of two vectors outside the ε neighborhood is outside the ε neighborhood (Lemma 10).

Due to the choice of ε, for every $\varepsilon' < \tilde{\varepsilon}$ there exists a time after which there are vectors of weight q_ε in the ε' neighborhood of $\hat{\varphi}_{i,\max}$. According to

what was shown above, these are descendants of the set of vectors $V_{\text{in},\tilde{\varepsilon}}$ that are never mixed with vectors that are not descendants thereof. This set is therefore the required $V_{i,\max}$. □

We next prove that the pool of auxiliary vectors converges:

Lemma 14 (Auxiliary collective convergence)**.** *There exists a time t, such that the normalized descendants of each vector in pool(t) converge to a specific destination vector, and merge only with descendants of vectors that converge to the same destination.*

Proof. By Lemmas 11 and 13, for every $1 \leq i \leq n$, there exists a maximal i'th reference angle, $\hat{\varphi}_{i,\max}$, a time, t_i, and a set of vectors, $V_{i,\max} \subset \mathit{pool}(t_i)$, s.t. the i'th reference angles of the vectors $V_{i,\max}$ converge to $\hat{\varphi}_{i,\max}$, and the descendants of $V_{i,\max}$ are merged only with one another.

The proof continues by induction. At t_i we consider the vectors that are not descendants of $V_{i,\max} \in \mathit{pool}(t_i)$. The descendants of these vectors are never merged with the descendants of the $V_{i,\max}$ vectors. Therefore, the proof applies to them with a new maximal i'th reference angle. This can be applied repeatedly, and since the weight of the vectors is bounded from below by q, we conclude that there exists a time t after which, for every vector v in the pool at time $t' > t$, the i'th reference of v converges. Denote that time $t_{\mathit{conv},i}$.

Next, let $t_{\mathit{conv}} = \max\{t_{\mathit{conv},i} | 1 \leq i \leq n\}$. After t_{conv}, for any vector in the pool, all of its reference angles converge. Moreover, two vectors are merged only if all of their reference angles converge to the same destination. Therefore, at t_{conv}, the vectors in $\mathit{pool}(t_{\mathit{conv}})$ can be partitioned into disjoint sets s.t. the descendants of each set are merged only with one another and their reference angles converge to the same values. For a cluster x of vectors whose reference angles converge to $(\varphi_i^x)_{i=1}^n$, its destination in the mixture space is the normalized vector $(\cos\varphi_i^x)_{i=1}^n$. □

We are now ready to derive the main result of this section.

Corollary 1. *The clustering series pool(t) converges.*

Proof. Lemma 14 shows that the pool of vectors is eventually partitioned into clusters. This applies to the weighted summaries pool as well, due to the correspondence between summaries and auxiliaries (Lemma 9).

For a cluster of clusters, define its destination cluster as follows: Its weight is the sum of weights of clusters in the cluster at t_{conv}, and its summary is that of the mixture space destination of the cluster's mixture vectors. Using requirement R1, it is easy to see that after t_{conv}, the clustering series $\mathit{pool}(*)$ converges to the set of destination clusters formed this way. □

4.5.2 Distributed Convergence

We show that the clusterings in each node converge to the same clustering of the input values.

Lemma 15. *There exists a time t_{dist} after which each node holds at least one cluster from each cluster of clusters.*

Proof. First note that after t_{conv}, once a node has obtained a cluster that converges to a destination x, it will always have a cluster that converges to this destination, since it will always have a descendant of that cluster — no operation can remove it.

Consider a node i that obtains a cluster that converges to a destination x. It eventually sends a descendant thereof to each of its neighbors due to the fair choice of neighbors. This can be applied repeatedly and show that, due to the connectivity of the graph, eventually all nodes hold clusters converging to x. □

Boyd et al. [21] analyzed the convergence of weight based average aggregation. The following lemma can be directly derived from their results:

Lemma 16. *In an infinite run of Algorithm 5, after t_{dist}, at every node, the relative weight of clusters converging to a destination x converges to the relative weight of x (in the destination clustering).*

We are now ready to prove the main result of this section.

Theorem 3. *Algorithm 5, with any implementation of the functions* `valToSummary`*,* `partition` *and* `mergeSet` *that conforms to Requirements R1–R4, solves the Distributed clustering Problem (Definition 7).*

Proof. Corollary 1 shows that the pool of all clusters in the system converges to some clustering *dest*, i.e., there exist mappings ψ_t from clusters in the pool to the elements in *dest*, as in Definition 6. Lemma 15 shows that there exists a time t_{dist}, after which each node obtains at least one cluster that converges to each destination.

After this time, for each node, the mappings ψ_t from the clusters of the node at t to the *dest* clusters show convergence of the node's clustering to the clustering *dest* (of all input values). Corollary 1 shows that the summaries converge to the destinations, and Lemma 16 shows that the relative weight of all clusters that are mapped to a certain cluster x in *dest* converges to the relative weight of x. □

Appendix

4.A Decreasing Reference Angle

We prove Lemma 10, showing that the sum of two vectors results in a vector with a reference angle not larger than those of the original vectors. The proof considers the 3-dimensional space spanned by the two summed vectors and the i'th axis. We show in Lemma 17 that it is sufficient to consider the angles of the two vectors in a 2-dimensional space they span.

Recall we denote by $\|v\|_p$ the L^p norm of v. For simplicity, we denote the Euclidean (L^2) norm by $\|v\|$. Denote by $v_1 \cdot v_2$ the scalar product of the vectors v_1 and v_2. Then the angle between two vectors in the mixture space is:

$$\arccos\left(\frac{v_a \cdot v_b}{\|v_a\| \cdot \|v_b\|}\right) .$$

We now show that we may prove for 2-dimensions rather than 3:

Lemma 17 (Reduction to 2 dimensions)**.** *In a 3 dimensional space, let v_a and v_b be two vectors lying on the XY plane with angles not larger than $\pi/2$ with the X axis, and v_a's angle with the X axis is larger than that of v_b. Let v_e be a vector in the XZ plane whose angle with the X axis is smaller than $\pi/2$ and with the Z axis not larger than $\pi/2$. Then v_b's angle with v_e is smaller than that of v_a.*

Proof. Let us express the angle of the vector v_a on the XY plane with v_e using the angle of the vector with the X axis, i.e., with the projection of v_e on the XY plane, as shown in Figure 4.6. Denote the end point of the vector by A, and the origin by O. Construct a perpendicular line to the X axis passing through A. Denote the point of intersection $\tilde{E}$. From $\tilde{E}$ take a perpendicular line to the XY plane, until intersecting v_e. Denote that intersection point E. OE is the vector e_i and $O\tilde{E}$ is its projection on the XY axis. Denote the angle AOE by φ_a and $AO\tilde{E}$ by $\tilde{\varphi}_a$. Denote the angle $EO\tilde{E}$ by $\tilde{\varphi}_e$.

$$\begin{aligned}
O\tilde{E} &= |v| \cos\tilde{\varphi}_a \\
OE &= \frac{O\tilde{E}}{\cos\tilde{\varphi}_e} = \frac{|v|\cos\tilde{\varphi}_a}{\cos\tilde{\varphi}_e} \\
E\tilde{E} &= O\tilde{E}\tan\tilde{\varphi}_e = |v|\cos\tilde{\varphi}_a\tan\tilde{\varphi}_e \\
A\tilde{E} &= |v|\sin\tilde{\varphi}_a \\
AE &= \sqrt{E\tilde{E}^2 + A\tilde{E}^2} = \sqrt{(|v|\cos\tilde{\varphi}_a\tan\tilde{\varphi}_e)^2 + (|v|\sin\tilde{\varphi}_a)^2}
\end{aligned}$$

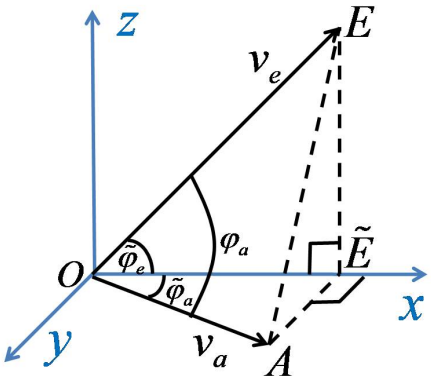


Figure 4.6: The angles of the vectors v_a and v_e.

Now we can use the law of cosines to obtain:

$$\varphi_a = \arccos \frac{OA^2 + OE^2 - AE^2}{2 \cdot OA \cdot OE} = \arccos(\cos\tilde{\varphi}_a \cos\tilde{\varphi}_e) \ . \tag{4.3}$$

Since $0 \leq \tilde{\varphi}_a \leq \pi/2$ and $0 \leq \tilde{\varphi}_e \leq \pi/2$, we see that φ_a is monotonically increasing with $\tilde{\varphi}_a$. We use similar notation for the vector b, and since $\tilde{\varphi}_b < \tilde{\varphi}_a$, and both are smaller than $\pi/2$, then:

$$\begin{aligned} \cos\tilde{\varphi}_a &\leq \cos\tilde{\varphi}_b \\ \cos\tilde{\varphi}_a \cos\tilde{\varphi}_e &\leq \cos\tilde{\varphi}_b \cos\tilde{\varphi}_e \\ \arccos(\cos\tilde{\varphi}_a \cos\tilde{\varphi}_e) &\geq \arccos(\cos\tilde{\varphi}_b \cos\tilde{\varphi}_e) \\ \varphi_a &\geq \varphi_b \ . \end{aligned} \tag{4.4}$$

□

Now we return to the n dimensional mixture space.

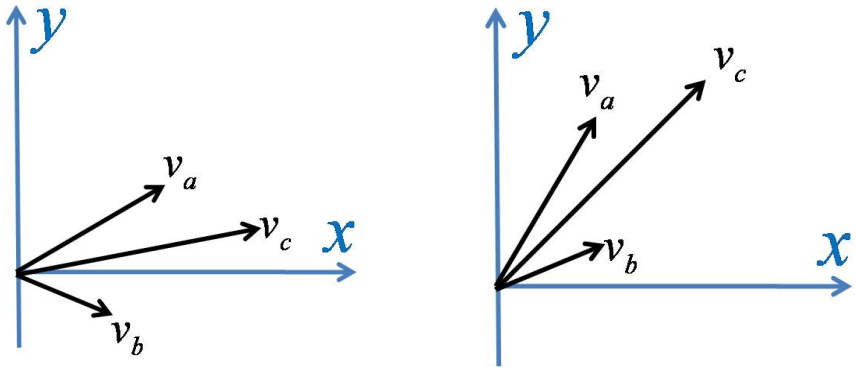
Lemma 10 (restated) *The sum of two vectors in the mixture space is a vector with a smaller i'th reference angle than the larger i'th reference angle of the two, for any $1 \leq i \leq n$.*

Proof. Denote the two vectors v_a and v_b, and their i'th reference angles φ_i^a and φ_i^b, respectively. Assume without loss of generality that $\varphi_i^a \geq \varphi_i^b$. Denote the sum vector by v_c.

It is sufficient to prove the above in the 3 dimensional space spanned by v_a, v_b and e_i. Align the XYZ axes such that v_a and v_b lie on the XY plane and the projection of e_i on that plane is on the X axis. The vector v_c lies on the XY plane, as it is a linear combination of two vectors on the plane.

By Lemma 17, it is sufficient to show that the angle of v_c with the projection of the reference vector is smaller than the angle of v_a with the projection.

The angle between v_c and the X axis is smaller than v_a's angle with it. The only two possible constructions are shown in Figure 4.7.



(a) (b)

Figure 4.7: The possible constructions of two vectors v_a and v_b and their sum v_c, s.t. their angles with the X axis are smaller than $\pi/2$ and v_a's angle is larger than v_b's angle.

□

4.B ε' Exists

Lemma 12 (restated) *For any ε, there exists an $\varepsilon' < \varepsilon$ such that if a vector v_{out} lies outside the ε-neighborhood of $\hat{\varphi}_{i,max}$ (i.e., has a reference angle smaller than $\hat{\varphi}_{i,max} - \varepsilon$), and a vector v_{in} lies inside the ε'-neighborhood (i.e., has a reference angle larger than $\hat{\varphi}_{i,max} - \varepsilon'$), then their sum v_{sum} lies outside the ε' neighborhood.*

Proof. Consider the 3 dimensional space spanned by v_{in}, v_{out} and e_i. Align the XYZ axes such that v_{in} and v_{out} lie on the XY plane and the projection of e_i on that plane is aligned with the X axis. Denote this projection by $\tilde{e}_i$. v_{sum} lies on the XY plane, as it is a linear combination of two vectors on the plane. Denote by $\tilde{\varphi}^i_{\text{in}}$, $\tilde{\varphi}^i_{\text{out}}$ and $\tilde{\varphi}^i_{\text{sum}}$ the angles between $\tilde{e}_i$ and the vectors v_{in}, v_{out} and v_{sum}, respectively. Denote by $\tilde{\varphi}^i_{e_i}$ the angle between e_i and its projection $\tilde{e}_i$.

Consider some $\varepsilon' \leq \varepsilon/2$, so that the angle between v_{in} and v_{out} is at least $\varepsilon/2$. Notice that the L^2 norms of v_{in} and v_{out} are bounded between q from

below and $\sqrt{s}$ from above. Observing Figure 4.7 again, we deduce that there is a lower bound on the difference between the angles:

$$\tilde{\varphi}^i_{\text{sum}} < \tilde{\varphi}^i_{\text{in}} - x_1 \ . \tag{4.5}$$

Due to the previous bound, and noting that all angles are not larger than $\pi/2$, a constant x_2 exists such that

$$\cos \tilde{\varphi}^i_{\text{in}} < \cos \tilde{\varphi}^i_{\text{sum}} - x_2 \ . \tag{4.6}$$

Since the reference angles of v_{in} and v_{out} are different, at least one of them is smaller than $\pi/2$, therefore $\tilde{\varphi}^i_{e_i} < \pi/2$ for any such couple. Therefore, $\cos \tilde{\varphi}^i_{e_i}$ is a bounded size, and factoring Inequality 4.6 we deduce that there exists a constant x_3 such that

$$\cos \tilde{\varphi}^i_{\text{in}} \cos \tilde{\varphi}^i_{e_i} < \cos \tilde{\varphi}^i_{\text{sum}} \cos \tilde{\varphi}^i_{e_i} - x_3 \ . \tag{4.7}$$

We use the inverse cosine function with Inequality 4.7 to finally deduce that there exists a constant x_4 such that

$$\arccos(\cos \tilde{\varphi}^i_{\text{in}} \cos \tilde{\varphi}^i_{e_i}) > \arccos(\cos \tilde{\varphi}^i_{\text{sum}} \cos \tilde{\varphi}^i_{e_i}) + x_4 \tag{4.8}$$

$$\varphi^i_{\text{in}} > \varphi^i_{\text{sum}} + x_4 \tag{4.9}$$

Therefore, for a given ε, we choose

$$\varepsilon' < \min\left\{\frac{1}{2}x_4, \frac{1}{2}\varepsilon\right\}.$$

With this ε', we obtain $\varphi^i_{\text{sum}} < \hat{\varphi}_{i,\max} - \varepsilon'$, as needed. □

Part II

Consistency in Cloud Storage

Chapter 5

Background

Advances in data center technologies have led to extensive use of data centers to store large volumes of data in a managed distributed system. Internet services such as social networks, search engines and shopping sites store huge volumes of data. Large companies often use their own datacenters, but both large and small companies often use external storage services. In both cases, the users of such storage systems require reliability, as they store their user's data, low latency, and high throughput, to swiftly interact with a multitude of users.

However, with datacenters consisting of thousands of servers, the failure of individual machines and network components is the norm, rather than the exception. Thus, to achieve reliability, replication must be used. Second, due to the large number of concurrent processes and complex network structure, it is undesirable for replication algorithms to make timing assumptions, as these may decrease efficiency, requiring unnecessary waits. To avoid such assumptions, asynchronous algorithms must be used.

In this part we address two challenges regarding reliable large data storage in datacenters. Recent years have shown that even the largest cloud storage providers occasionally fail, and users have to replicate data among multiple providers to obtain reliability. However, classical replication techniques (e.g., ABD [11]) are not applicable here, since storage services typically export only a key-value store (KVS) interface, with functions for storing and retrieving values associated with unique keys.

In Chapter 6 we present the first efficient, wait-free algorithm that emulates multi-reader, multi-writer reliable storage from a set of potentially faulty KVS replicas in an asynchronous environment. We implemented the algorithm and tested it using real-world providers, and we demonstrate through simulation its low overhead in terms of space and speed in various scenarios.

Next, in Chapter 7 we present an architecture for a datacenter storage

service called ACID-RAIN: ACID transactions in a Resilient Archive with Independent Nodes. ACID-RAIN is a novel architecture for efficiently implementing transactions in a distributed data store — a sought-after but difficult to achieve capability.

ACID-RAIN uses logs in a novel way, limiting reliability to a single tier of the system. A set of independent nodes form an outer layer that caches the data, backed by a set of independent reliable log services. ACID-RAIN avoids collisions between transactions by using prediction to order the transactions before they take actions that would lead to an abort. ACID-RAIN is efficient in overcoming failures, and its throughput increases linearly with the number of nodes.

Chapter 6

Robust Data Sharing with KVS's

Motivation

In the recent years, the *key-value store* (*KVS*) abstraction has become the most popular way to access Internet-scale "cloud" storage systems. Such systems provide storage and coordination services for online platforms [38, 9, 75, 104], ranging from web search to social networks, but they are also available to consumers directly [8, 26, 94, 86].

A KVS offers a range of simple functions for manipulation of unstructured data objects, called *values*, each one identified by a unique *key*. While different services and systems offer various extensions to the KVS interface, the common denominator of existing KVS services implements an associative array: A client may *store* a value by associating the value with a key, *retrieve* a value associated with a key, *list* the keys that are currently associated, and *remove* a value associated with a key.

Although existing KVS services provide high availability and reliability using replication internally, a KVS service is managed by one provider; many common components (and thus failure modes) affect its operation. A problem with any such component may lead to service outage or even to data being lost, as witnessed, for example in an Amazon S3 incident [6], an Amazon service disruption [7], Google's temporary loss of email data [56] and Microsoft's Azure's cluster failure [13]. As a remedy, a client may increase data reliability by replicating it among several storage providers (all offering a KVS interface), using the guarantees offered by robust distributed storage algorithms [54, 11]. Data replication across different clouds is a topic of active research [2, 24, 95, 17]; we discuss related work in Section 6.1.

Problem

Our data replication scheme relies on multiple providers of raw storage, called *base objects* here, and emulates a single, more reliable shared storage abstraction, which we model as a *read/write register*. A register represents the most basic form of storage, from which a KVS service or more elaborate abstractions may be constructed. The emulated register tolerates asynchrony, concurrency, and faults among the clients and the base objects. For increased parallelism, the clients do not communicate with each other for coordination, and they may not even be aware of each other. We detail the system model in Section 6.2.

Many well-known robust distributed storage algorithms exist (for an overview see [23]). They all use versioning [103], whereby each stored value is associated with a logical timestamp. For instance, with the multi-writer variant of the register emulation by Attiya et al. [11], the base objects perform custom *computation* depending on the timestamp, in order to identify and to retain only the newest written value. Without this an *old-new overwrite* problem might occur when a slow write request with an old value and a small timestamp reaches a base object after the latter has already updated its state to a newer value with a higher timestamp. On the other hand, one might let each client use its own range of timestamps and retain all versions of a written value at the KVSs [51, 1], but this approach is overly expensive in the sense that it requires as many base objects as there are clients. If periodic garbage collection (GC) is introduced to reduce the consumed storage space, one may face a *GC racing* problem, whereby a client attempts to retrieve a value associated with a key that has become obsolete and was removed.

Contribution

We provide a robust, asynchronous, and space-efficient emulation of a register over a set of KVSs, which may fail by crashing. Our formalization of a key-value store (KVS) object represents the common denominator among existing commercial KVSs, which renders our approach feasible in practice. Inspired by Internet-scale systems, the emulation is designed for an unbounded number of clients and supports multiple readers and writers (MRMW). The algorithm is *wait-free* [62] in the sense that all operations invoked by a correct client eventually complete. It is also *optimally resilient*, i.e., tolerates the failure of any minority of the KVSs and of any number of clients.

We give two variations of the emulation in Section 6.3. Our basic algorithm emulates a register with *regular* semantics in the multi-writer model [98]. It does not require read operations to write to the KVSs. Precluding readers

from writing is practically appealing, since the clients may belong to different domains and not all readers may have write privileges for the shared memory. But it also poses a challenge because of the GC racing problem. Our solution stores the same value *twice* in every KVS: (1) under an *eternal* key, which is never removed by a garbage collector, and therefore is vulnerable to an old-new overwrite and (2) under a *temporary* key, named according to the version; obsolete temporary keys are garbage-collected by write operations, which makes these keys vulnerable to the GC racing problem. The algorithm for reading accesses the values in the KVSs according to a specific order, which guarantees that every read terminates eventually despite concurrent write operations. In a sense, the eternal and temporary copies complement each other and, together, guarantee the desirable properties of our emulation outlined above.

We then present an extension that emulates an *atomic* register [78]. It uses the standard approach of having the readers write back the returned value [11]. This algorithm requires read operations to write, but this is necessary [78, 12]. Section 6.4 analyzes the correctness of both algorithms.

Our emulations maintain only two copies of the stored value per KVS in the common case (i.e., failure-free executions without concurrent operations). We show that this is also necessary. In the worst case, a stored value exists in every KVS once for every concurrent write operation, in addition to the one stored under the eternal key. Hence, our emulations have optimal space complexity.

Even though it is well-known how to implement a shared, robust multi-writer register from simpler storage primitives such as unreliable single-writer registers [12], our algorithm is the first to achieve an emulation from KVSs with feasible space overhead. Section 6.5 establishes bounds on the algorithms' space complexity.

Note that some of the available KVSs export proprietary versioning information [8, 104]. However, one cannot exploit this for a data replication algorithm before the format and semantics of those versions has been harmonized. Another KVS prototype allows to execute client operations [52], but this technique is far from commercial deployment. We believe that some KVSs may also support atomic "read-modify-write" operations at some future time, thereby eliminating the problem addressed here. But until these extensions are deployed widely and have been standardized, our algorithm represents the best possible solution for minimizing space overhead of data replication on KVSs.

Last but not least, we simulate the algorithm with practical network parameters for exploring its properties. The results, given in Section 6.6, demonstrate that in realistic cases, our algorithm seldom increases the du-

ration of read operations beyond the optimal duration. Furthermore, the algorithm scales to many concurrent writers without incurring any slowdown. We have also implemented our approach and report in Section 6.7 on benchmarks obtained with cloud-storage providers; they confirm the practicality of the algorithm.

A preliminary version of the work presented in this chapter appears in the proceedings of the 42nd International Conference on Dependable Systems and Networks (DSN'12) [15].

6.1 Related Work

There is a rich body of literature on robust register emulations that provide guarantees similar to ours. However, virtually all of them assume read-modify-write functionalities, or computation at the base objects. These include the single-writer multi-reader (SWMR) atomic wait-free register implementation of Attiya et al. [11], its dynamic multi-writer counterparts by Lynch and Shvartsman [81, 55] and Englert and Shvartsman [43], wait-free simulations of Jayanti et al. [65], low-latency atomic wait-free implementations of Dutta et al. [41] and Georgiou et al. [53], and the consensus-free versions of Aguilera et al. [3]. These solutions are not directly applicable to our model where KVSs are used as base objects, due to the old-new overwrite problem.

Notable exceptions that are applicable in our KVS context are SWMR regular register emulation by Gafni and Lamport [51] and its Byzantine variant by Abraham et al. [1] that use registers as base objects. However, transforming these SWMR emulations to support a large number of writers is inefficient: standard register transformations [12, 23] that can be used to this end require at least as many SWMR regular registers as there are clients, even if there are no faults. This is prohibitively expensive in terms of space complexity and effectively limits the number of supported clients. Chockler and Malkhi [29] acknowledge this issue and propose an algorithm that supports an unbounded number of clients (like our algorithm). However, their method uses base objects (called "active disks") that may carry out computations. In contrast, our emulation leverages the operations in the KVS interface, which is more general than a register due to its list and remove operations, and supports an unbounded number of clients. Ye et al. [106] overcome the GC racing problem by having the readers "reserve" the versions they intend to read, by storing extra values that signal to the garbage collector not to remove the version being read. This approach requires readers to have write access, which is not desirable.

Two recent works share our goal of providing robust storage from KVS base objects. Abu-Libdeh et al. [2] propose RACS, an approach that casts RAID techniques to the KVS context. RACS uses a model different from ours and basically relies on a proxy between the clients and the KVSs, which may become a bottleneck and single point-of-failure. In a variant that supports multiple proxies, the proxies communicate directly with each other for synchronizing their operations. Bessani et al. [17] propose a distributed storage system, called DepSky, which employs erasure coding and cryptographic tools to store data on KVS objects prone to Byzantine faults. However, the basic version of DepSky allows only a single writer and thereby circumvents the problems addressed here. An extension supports multiple writers through a locking mechanism that determines a unique writer using communication among the clients. In comparison, the multi-writer versions of RACS and DepSky both serialize write operations, whereas our algorithm allows concurrent write operations from multiple clients in a wait-free manner. Therefore, our solution scales easily to a large number of clients.

6.2 Model

6.2.1 Executions

The system is comprised of multiple *clients* and *(base) objects*. We model them as I/O automata [80], which contain state and potential transitions that are triggered by *actions*. The interface of an I/O automaton is determined by external (input and output) actions. A client may *invoke* an *operation*[1] on an object (with an output action of the client automaton that is also an input action of the object automaton). The object reacts to this invocation, possibly involving state transitions and internal actions, and returns a *response* (an output action of the object that is also an input action of the client). This *completes* the operation. We consider an asynchronous system, i.e., there are no timing assumptions that relate invocations and responses. (Consult [80, 12] for details.)

Clients and objects may *fail* by stopping, i.e., *crashing*, which we model by a special action *stop*. When *stop* occurs at automaton A, all actions of A become disabled indefinitely and A no longer modifies its state. A client or base object that does not fail is called *correct*.

An *execution* σ of the system is a sequence of invocations and responses. We define a partial order among the operations. An operation o_1 *precedes*

[1] For simplicity, we refer to an *operation* when we should be referring to *operation execution*.

another operation o_2 (and o_2 *follows* o_1) if the response of o_1 precedes the invocation of o_2 in σ. We denote this by $o_1 \prec_\sigma o_2$. The two operations are *concurrent* if neither of them preceded the other. An operation o is *pending* in an execution σ if σ contains the invocation of o but not its response; otherwise the operation is *complete*. An execution σ is *well-formed* if every subsequence thereof that contains only the invocations and responses of one client on one object consists of alternating invocations and responses, starting with an invocation. A well-formed execution σ is *sequential* if every prefix of σ contains at most one pending operation; in other words, in a sequential execution, the response of every operation immediately follows its invocation.

A *real-time sequential permutation* π of an execution σ is a sequential execution that contains all operations that are invoked in σ and only those operations and in which for any two operations o_1 and o_2 such that $o_1 \prec_\sigma o_2$, it holds $o_1 \prec_\pi o_2$.

A *sequential specification* of some object O is a prefix-closed set of sequential executions containing operations on O. It defines the desired behavior of O. A sequential execution π is *legal* with respect to the sequential definition of O if the subsequence of σ containing only operations on O lies in the sequential specification of O.

Finally, an object implementation is *wait-free* if it eventually responds to an invocation by a correct client [61].

6.2.2 Register Specifications

6.2.2..1 Sequential Register A *register* [78] is an object that supports two operations: one for writing a value $v \in \mathcal{V}$, denoted by **write**(v), which returns ACK, and one for reading a value, denoted by **read**(), which returns a value in $\mathcal{V}$. The sequential specification of a register requires that every **read** operation returns the value written by the last preceding **write** operation in the execution, or the special value $\perp$ if no such operation exists. For simplicity, our description assumes that every distinct value is written only once.

Registers may exhibit different semantics under concurrent access, as described next.

6.2.2..2 Multi-Reader Multi-Writer Regular Register The following semantics describe a *multi-reader multi-writer regular register* (*MRMW-regular*), adapted from [98]. A MRMW-regular register only guarantees that different **read** operations agree on the order of preceding **write** operations.

Definition 9 (MRMW-regular register)**.** *A well-formed execution σ of a register is MRMW-regular if there exists a sequential permutation π of the operations in σ as follows: for each* ***read*** *operation r in σ, let π_r by a subsequence of π containing r and those* ***write*** *operations that do not follow r in σ; furthermore, let σ_r be the subsequence of σ containing r and those* ***write*** *operations that do not follow it in σ; then π_r is a legal real-time sequential permutation of σ_r. A register is MRMW-regular if all well-formed executions on that register are MRMW-regular.*

Atomic Register A stronger consistency notion for a concurrent register object than regular semantics is *atomicity* [78], also called linearizability [62]. In short, atomicity stipulates that it should be possible to place each operation at a singular point (linearization point) between its invocation and response.

Definition 10 (Atomicity)**.** *A well-formed execution σ of a concurrent object is* atomic *(or* linearizable*), if σ can be extended (by appending zero or more responses) to some execution σ', such that there is a legal real-time sequential permutation π of σ'. An object is atomic if all well-formed executions on that object are atomic.*

6.2.3 Key-Value Store

A *key-value store* (*KVS*) object is an associative array that allows storage and retrieval of *values* in a set $\mathcal{X}$ associated with *keys* in a set $\mathcal{K}$. The size of the stored values is typically much larger than the length of a key, so the values in $\mathcal{X}$ cannot be translated to elements of $\mathcal{K}$ and be stored as keys.

A KVS supports four operations: (1) *Storing* a value x associated with a key *key* (denoted **put**(*key*, x)), (2) *retrieving* a value x associated with a key ($x \leftarrow$ **get**(*key*)), which may also return FAIL if *key* does not exist, (3) *listing* the keys that are currently associated (*list* $\leftarrow$ **list**()), and (4) *removing* a value associated with a key (**remove**(*key*)).

Our formal sequential specification of the KVS object is given in Algorithm 7. This implementation maintains in a variable *live* the set of associated keys and values. The *space complexity* of a KVS at some time during an execution is given by the number of associated keys, that is, by the value $|live|$.

6.2.4 Register Emulation

The system is comprised of a finite set of clients and a set of n atomic wait-free KVSs as base objects. Each client is named with a unique identifier

Algorithm 7: Key-value store object i

```
1  state
2      live ⊆ 𝒦 × 𝒳, initially ∅
3  On invocation put_i(key, value)
4      live ← (live \ {⟨key, x⟩ | x ∈ 𝒳}) ∪ ⟨key, value⟩
5      return ACK
6  On invocation get_i(key)
7      if ∃x : ⟨key, x⟩ ∈ live then
8          return x
9      else
10         return FAIL
11 On invocation remove_i(key)
12     live ← live \ {⟨key, x⟩ | x ∈ 𝒳}
13     return ACK
14 On invocation list_i()
15     return {key | ∃x : ⟨key, x⟩ ∈ live}
```

from an infinite ordered set $\mathcal{ID}$. The KVS objects are numbered $1, \ldots, n$. Initially, the clients do not know the identities of other clients or the total number of clients.

Our goal is to have the clients *emulate* a MRMW-regular register and an atomic register using the KVS base objects [80]. The emulations should be wait-free and tolerate that any number of clients and any minority of the KVSs may crash. Furthermore, an emulation algorithm should associate only few keys to values in every KVS (i.e., have low space complexity).

6.3 Algorithm

6.3.1 Pseudo Code Notation

Our algorithm is formulated using functions that execute the register operations. They perform computation steps, invoke operations on the base objects, and may *wait for* such operations to complete. To simplify the pseudo code, we imagine there are concurrent execution "threads" as follows. When a function **concurrently** executes a block, it perform the same steps and invokes the same operations once for each KVS base object in parallel. An algorithm proceeds past a **concurrently** statement as indicated by a termination property; in all our algorithms, this condition requires that the block completes for a majority of base objects.

In order to maintain a well-formed execution, the system implicitly keeps track of pending operations at the base objects. Relying on this state, every instruction to **concurrently** execute a code block explicitly waits for a base object to complete a pending operation, before its "thread" may invoke

another operation. This convention avoids cluttering the pseudo code with state variables and complicated predicates that have the same effect.

6.3.2 MRMW-Regular Register

We present an algorithm for implementing a MRMW-regular register, where **read** operations do not store data at the KVSs.

Inspired by previous work on fault-tolerant register emulations, our algorithm makes use of versioning. Clients associate versions with the values they store in the KVSs. In each KVS there may be several values stored at any time, with different versions. Roughly speaking, when writing a value, a client associates it with a version that is larger than the existing versions, and when reading a value, a client tries to retrieve the one associated with the largest version [11]. Since a KVS cannot perform computations and atomically store one version and remove another one, values associated with obsolete versions may be left around. Therefore our algorithm explicitly removes unused values, in order to reduce the space occupied at a KVS.

A version is a pair[2] $(seq, id) \in \mathbb{N}_0 \times \mathcal{ID}$, where the first number is a sequence number and the second is the identity of the client that created the version and used it to store a value. When comparing versions with the $<$ operator and using the max function, we respect the lexicographic order on pairs. We assume that the key space of a KVS is the version space, i.e., $\mathcal{K} = \mathbb{N}_0 \times \mathcal{ID}$, and that the value space of a KVS allows clients to store either a register value from $\mathcal{V}$ or a version and a value in $(\mathbb{N}_0 \times \mathcal{ID}) \times \mathcal{V}$.[3]

At the heart of our algorithm lies the idea of using *temporary keys*, which are created and later removed at the KVSs, and an *eternal key*, denoted ETERNAL, which is never removed. Both represent a register value and its associated version. When a client writes a value to the emulated register, it determines the new version to be associated with the value, accesses a majority of the KVSs, and stores the value and version *twice* at every KVS — once under a new temporary key, named according to the version, and once under the eternal key, overwriting its current value. The data stored under a temporary key directly represents the written value; data stored under the eternal key contains the register value and its version. The writer also performs garbage collection of values stored under obsolete temporary keys, which ensures the bound on space complexity.

[2]We denote by $\mathbb{N}_0$ the set $\{0, 1, 2, \dots\}$.

[3]In other words, $\mathcal{X} = \mathcal{V} \cup (\mathbb{N}_0 \times \mathcal{ID}) \times \mathcal{V}$. Alternatively one may assume that there exists a one-to-one transformation from the version space to the KVS key space, and from the set of values written by the clients to the KVS value space. In practical systems, where $\mathcal{K}$ and $\mathcal{X}$ are strings, this assumptions holds.

6.3.2.a Read

When a client reads from the emulated register through algorithm **regularRead** (Algorithm 9), it obtains a version and a value from a majority of the KVSs and returns the value associated with the largest obtained version.

To obtain such a pair from a KVS i, the reader invokes a function **getFromKVS**(i) (shown in Algorithm 8). It first determines the currently largest stored version, denoted by ver_0, through a snapshot of temporary keys with a **list** operation.

Then the reader enters a loop, from which it only exits after finding a value associated with a version that is at least ver_0. It first attempts to retrieve the value under the key representing the largest version. If the key exists, the reader has found a suitable value. However, this step may fail due to the GC racing problem, that is, because a concurrent writer has removed the particular key between the times when the client issues the **list** and the **get** operations.

In this case, the reader retrieves the version/value pair stored under the eternal key. As the eternal key is stored first by a writer and never removed, it always exists after the first write to the register. If the retrieved version is greater than or equal to ver_0, the reader returns this value. However, if this version is smaller than ver_0, an old-new overwrite has occurred, and the reader starts another iteration of the loop.

This loop terminates after a bounded number iterations: Note that an iteration is not successful only if a GC race and an old-new overwrite have both occurred. But a concurrent writer that may cause an old-new overwrite must have invoked its write operation *before* the reader issued the first **list** operation on some KVS. Thus, the number of loop iterations is bounded by the number of clients that concurrently execute a **write** operation in parallel to the **read** operation (i.e., the point contention of **write** operations).

Algorithm 8: Retrieve a legal version-value pair

```
1  function getFromKVS(i)
2      list ← list_i() \ ETERNAL
3      if list = ∅ then
4          return ((0, ⊥), ⊥)
5      ver_0 ← max(list)
6      while True do
7          val ← get_i(max(list))
8          if val ≠ FAIL then
9              return (max(list), val)
10         (ver, val) ← get_i(ETERNAL)
11         if ver ≥ ver_0 then
12             return (ver, val)
13         list ← list_i() \ ETERNAL
```

Algorithm 9: Client c MRMW-regular **read** operation

```
1 function regularRead_c()
2     results ← ∅
3     concurrently for each 1 ≤ i ≤ n, until a majority completes
4         if an op. is pending at KVS i then wait for a response
5         result ← getFromKVS(i)
6         results ← results ∪ {result}
7     return val such that (ver, val) ∈ results and ver' ≤ ver for any (ver', val') ∈ results
```

6.3.2.b Write

A client writes a value to the register using algorithm **regularWrite** (Algorithm 11). First, the client lists the temporary keys in each base object and determines the largest version found in a majority of them. It increments this version and obtains a new version to be associated with the written value.

Then the client stores the value and the new version in all KVSs using a function **putInKVS**, shown in Algorithm 10, which also performs garbage collection. It first lists the existing keys and removes obsolete temporary keys, i.e., all temporary keys excluding the one corresponding to the maximal version. Subsequently the function stores the value and the version under the eternal key. To store the value under a temporary key, the algorithm checks whether the new version is larger than the maximal version of an existing key. If yes, it also stores the new value under the temporary key corresponding to the new version and removes the key holding the previous maximal version.

Once the function **putInKVS** finishes for a majority of the KVSs, the algorithm for writing to the register completes. It is important for ensuring termination of concurrent **read** operations that the writer first stores the value under the eternal key and later under the temporary key.

Algorithm 10: Store a value and a given version

```
1 function putInKVS(i, ver_w, val_w)
2     list ← list_i()
3     obsolete ← {v | v ∈ list ∧ v ≠ ETERNAL ∧ v < max(list)}
4     foreach ver ∈ obsolete do
5         remove_i(ver)
6     put_i(ETERNAL, (ver_w, val_w))
7     if ver_w > max(list) then
8         put_i(ver_w, val_w)
9         remove_i(max(list))
```

Algorithm 11: Client c MRMW-regular **write** operation

```
1  function regularWrite_c(val_w)
2      results ← {(0, ⊥)}
3      concurrently for each 1 ≤ i ≤ n, until a majority completes
4          if an op. is pending at KVS i then wait for a response
5          list ← list_i()
6          results ← results ∪ list
7      (seq_max, id_max) ← max(results)
8      ver_w ← (seq_max + 1, c)
9      concurrently for each 1 ≤ i ≤ n, until a majority completes
10         if an op. is pending at KVS i then wait for a response
11         putInKVS(i, ver_w, val_w)
12     return ACK
```

6.3.3 Atomic Register

The atomic register emulation results from extending the algorithm for emulating the regular register. Atomicity is achieved by having a client write back its read value before returning it, similar to the write-back procedure of Attiya et al. [11].

The **write** operation is the same as before, implemented by function **regularWrite** (Algorithm 11). The **read** operation is implemented by function **atomicRead** (Algorithm 12). Its first phase is unchanged from before and obtains the value associated with the maximal version found among a majority of the KVSs. Its second phase duplicates the second phase of the **regularWrite** function, which stores the versioned value to a majority of the KVSs.

Algorithm 12: Client c atomic **read** operation

```
1  function atomicRead_c()
2      results ← ∅
3      concurrently for each 1 ≤ i ≤ n, until a majority completes
4          if an op. is pending at KVS i then wait for a response
5          result ← getFromKVS(i)
6          results ← results ∪ {result}
7      choose (ver, val) ∈ results such that ver' ≤ ver for any (ver', val') ∈ results
8      concurrently for each 1 ≤ i ≤ n, until a majority completes
9          if an op. is pending at KVS i then wait for a response
10         putInKVS(i, ver, val)
11     return val
```

6.4 Correctness

In this section we sketch the arguments for correctness of the MRMW-regular register. The correctness of the atomic register follows analogously.

We say a **read** operation *reads a version ver* when the returned value has been associated with *ver* (Algorithm 9 line 7), and a **write** operation *writes a version ver* when an induced **put** operation stores a value under a temporary key corresponding to *ver* (Algorithm 11 line 11).

6.4.1 Safety

Consider any execution $\bar{\sigma}$ of the algorithm, the induced execution σ of the KVSs (in terms of KVS operations), and a real-time sequential permutation π of σ. Denote by π_i the sequence of actions from π that occur at some KVS replica i.

We first establish that for every KVS, the maximums of the versions returned by consecutive **list** operations cannot decrease, despite the fact that **write** operations also remove versions.

Lemma 18 (KVS version monotonicity)**.** *Consider a KVS i, a write operation w that writes version ver, and some operation $\textbf{\textit{put}}_i$ in π_i induced by w with a temporary key. Then the response of any operation $\textbf{\textit{list}}_i$ in π_i that follows $\textbf{\textit{put}}_i$ contains at least one temporary key that corresponds to a version equal to or larger than ver.*

The next step ensures that the versions of the emulated **read** and **write** operations respect the partial order of the operations in the execution. It holds because **read** and **write** operations always access a majority of the KVSs, and hence every two operations access at least one common KVS.

Lemma 19 (Partial order)**.** *In an execution $\bar{\sigma}$ of the algorithm, the versions of the read and write operations in $\bar{\sigma}$ respect the partial order of the operations in $\bar{\sigma}$:*

*a) When a **write** operation w writes a version v_w and a subsequent (in $\bar{\sigma}$) **read** operation r reads a version v_r, then $v_w \leq v_r$.*

*b) When a **write** operation w_1 writes a version v_1 and a subsequent **write** operation w_2 writes a version v_2, then $v_1 < v_2$.*

We may now construct a sequential permutation $\bar{\pi}$ of an execution $\bar{\sigma}$ by ordering all **write** operations of $\bar{\sigma}$ according to their versions and then adding all **read** operations after their matching **write** operations; concurrent **read** operations are added after their respective **write**s in the same order as in $\bar{\sigma}$. The safety of the MRMW-regular register follows.

Theorem 4 (MRMW-regular safety)**.** *Every well-formed execution $\bar{\sigma}$ of the MRMW-regular register emulation in Algorithms 9 and 11 is MRMW-regular.*

6.4.2 Liveness

The **write** routine obviously completes in finite time. The critical element is the **read** operation, for which we include a detailed proof.

Lemma 20 (Wait-free read)**.** *Every **read** operation completes in finite time.*

Proof. We argue that when a client c invokes **getFromKVS** for a correct KVS i, it returns in finite time. Algorithm 8 first obtains a list *list* of all temporary keys from KVS i and returns if no such key exists. If some temporary key is found, it determines the corresponding largest version ver_0 and enters a loop.

Towards a contradiction, assume that client c never exits the loop in some execution $\bar{\sigma}$ and consider the induced execution σ of the KVSs.

We examine one iteration of the loop. Note that its operations are wait-free and the iteration terminates. Prior to starting the iteration, the client determines *list* from an operation $\mathbf{list}_i$. In line 8 the algorithm attempts to retrieve the value associated with key $v_c = \max(list)$ through an operation $\mathbf{get}_c(v_c)$. This returns FAIL and the client retrieves the eternal key with an operation $\mathbf{get}_c(\text{ETERNAL})$. We observe that $\mathbf{list}_c \prec_\sigma \mathbf{get}_c(v_c) \prec_\sigma \mathbf{get}_c(\text{ETERNAL})$.

Since $\mathbf{get}_c(v_c)$ fails, some client must have removed it from the KVS with a $\mathbf{remove}(v_c)$ operation. Applying Lemma 18 to version v_c now implies that prior to the invocation of $\mathbf{get}_c(v_c)$, there exists a temporary key in KVS i corresponding to a version $v_d > v_c$ that was stored by a client d. Denote the operation that stored v_d by $\mathbf{put}_d(v_d)$. Combined with the previous observation, we conclude that $\mathbf{list}_c \prec_\sigma \mathbf{put}_d(v_d) \prec_\sigma \mathbf{get}_c(v_c) \prec_\sigma \mathbf{get}_c(\text{ETERNAL})$.

Furthermore, according to Algorithm 10, client d has stored a tuple containing $v_d > v_c$ under the eternal key prior to $\mathbf{put}_d(v_d)$ with an operation $\mathbf{put}_d(\text{ETERNAL})$. But the subsequent $\mathbf{get}_c(\text{ETERNAL})$ by client c returns a value containing a version *smaller* than v_c. Hence, there must be an *extra* client e writing concurrently, and its version-value pair has overwritten v_d and the associated value under the eternal key. This means that operation $\mathbf{put}_e(\text{ETERNAL})$ precedes $\mathbf{get}_c(\text{ETERNAL})$ in σ and stores a version $v_e < v_c$. Note that $\mathbf{put}_e(\text{ETERNAL})$ occurs exactly once for KVS i during the write by e.

As client e also uses Algorithm 11 for writing, its *results* variable must contain the responses of **list** operations from a majority of the KVSs. Denote by $\mathbf{list}_e$ its **list** operation whose response contains the largest version, as determined by e. Let $\mathbf{list}_c^0$ denote the initial list operation by c that determined ver_0 in Algorithm 8 (line 5). We conclude that $\mathbf{list}_e$ precedes $\mathbf{list}_c^0$ in σ. Summarizing the partial-order constraints on e, we have

$\mathbf{list}_e \prec_\sigma \mathbf{list}_c^0 \prec_\sigma \mathbf{put}_e(\text{ETERNAL}) \prec_\sigma \mathbf{get}_c(\text{ETERNAL})$.

Thus, in one iteration of the loop by reader c, some client d concurrently writes to the register. An extra client e has invoked a **write** operation before $\mathbf{list}_c^0$ and irrevocably makes progress after d invokes a **write** operation. Therefore, client e may cause *at most one* extra iteration of the loop by the reader. Since there are only a finite number of such clients, client c eventually exits the loop, and the lemma follows. □

6.5 Efficiency

We discuss the space complexity of the algorithms in this section. Note that the algorithm for writing performs garbage collection on a KVS before storing a temporary key in the KVS. This is necessary for bounding the space at the KVS, since the **putInKVS** function is called concurrently for all KVSs and may be aborted for some of them. If the algorithm would remove the obsolete temporary keys after storing the value, the function may be aborted just before garbage collection. This way, many obsolete keys might be left around and permanently occupy space at the KVS.

We provide an upper bound on the space usage. The time complexity of our emulations follows from analogous arguments.

It is obvious from Algorithm 11 that when a **write** operation runs in isolation (i.e., without any concurrent operations) and completes the **putInKVS** function on a set $\mathcal{C}$ of more than $n/2$ correct KVSs, then every KVS in $\mathcal{C}$ stores only the eternal key and one temporary key. Every such KVS has space complexity two. When there are concurrent operations, the space complexity may increase by one for every concurrent write operation, i.e., by the point contention of writes.

Theorem 5. *The space complexity of the MRMW-regular register emulation at any KVS is at most two plus the point contention of concurrent write operations.*

Proof. Consider an execution $\bar{\sigma}$ of the MRMW-regular register emulation. We prove the theorem by considering the operations $o_1, o_2, \ldots$ of some legal real-time sequential permutation π of σ, the KVS execution induced by $\bar{\sigma}$.

If at some operation o_t the number of keys that is written to KVS i but not removed is x, then at some operation prior to o_t, at least x register operations were concurrently run. We prove by induction on t. Initially the claim holds since there are no keys put and no clients run. Assume it holds until o_{t-1} and prove for o_t. If operation o_t is not a **put**, then the number of put keys is the same as at o_{t-1} and the claim holds by the induction assumption.

If operation o_t is $\mathbf{put}_i$, invoked by some client c, then it is performed by this client's $\mathbf{write}_c$ that first removed all but one temporary keys in its GC routine (Algorithm 10 lines 4–9). These **remove** operations precede the **put** in $\bar{\sigma}$, and therefore also its real-time sequential permutation π. All (except maybe one) versions that were written by **write**s that completed before $\mathbf{write}_c$ are therefore removed before operation o_t. The temporary keys in the system at o_{t-1} are ones that were written by operations concurrent with $\mathbf{write}_c$. The $\mathbf{put}_c$ operation therefore increases their number by one, so the number of keys is at most the number of concurrent **write** operations, as required. □

The same bound can be shown for the atomic register emulation, except here **read** operations may also increase the space complexity.

6.6 Simulation

To assess the properties of the algorithm, we analyze it through simulations under realistic conditions in this section. In particular, we demonstrate the scalability properties of our approach and compare it with a single-writer replication approach. In Section 6.7, we also assert the accuracy of the simulator by comparing its output with that of experiments run with an implementation of the algorithm, which accessed actual KVS cloud-storage providers over the Internet.

We have built a dedicated event-driven simulation framework in Python for this task. The simulator models our algorithm for clients (Algorithms 8, 9, 10, and 11) and for KVS replicas (Algorithm 7). In each simulation run, one or more clients perform **read** and **write** operations using our register emulation.

6.6.1 Simulation Setup

The simulated system contains a varying number of clients and three KVS replicas. The time for a client to execute a KVS operation consists of three parts: (1) the time for the invocation message to reach a KVS replica; (2) the time for a KVS to execute the operation, always assumed to be 0; and (3) the time for the response message to reach the client. Message delays (1) and (3) are influenced by two factors: first, the *network latency* of the client, which we model as a random variable with exponential distribution with a given mean; and, second, by the *size* of the transferred value and the available *network bandwidth*. We assume that metadata is always of negligible size and consider only the size of the stored values.

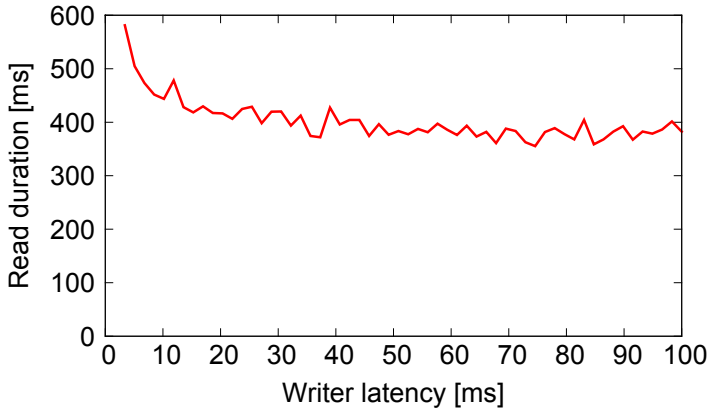


Figure 6.1: Simulation of the average duration of **read** operations shown with one concurrent writer accessing the KVS replicas at varying network latencies. The mean network latency of the reader is 100 ms; only when the writer has a much smaller latency does the **read** operations take longer than the expected minimum of 400 ms.

As the base case for our explorations, we use a network latency with a mean of 100 ms. Unless stated differently, the network available to every client has 1 MBps bandwidth and the data size is small, namely 500 bytes.

The simulator drives the algorithm through **read** and **write** operations of the clients. Clients issue operations in a closed-loop manner: each client issues a new request only after it has received a response for the previous request. For measuring a statistic like the average duration of **read** and **write** operations, a run is simulated for some time, the number of completed operations is counted, and the average of the statistic per operation is output. The runs are sufficiently long to produce a reliable average.

6.6.2 Read Duration

6.6.2..1 Latency A **read** operation takes at least two operations on the KVSs: an initial **list**, followed by at least one iteration of the loop in Algorithm 8. More iterations are needed only in the presence of concurrent **write** operations.

To observe this behavior, we run the simulation with a single writer and one reader. The two network latencies for the reader have a mean of 100 ms each. We vary the two network latencies of the writer from 2 ms to 100 ms in increments of 2 ms, to investigate a higher rate of **write** operations than **read** operations. Every average is computed from a simulation running for 40 s.

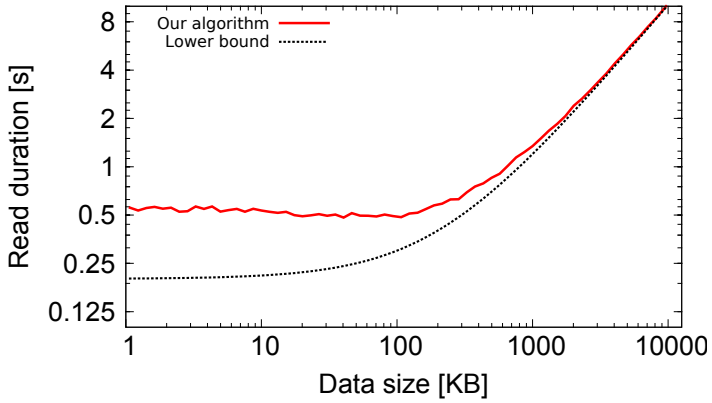


Figure 6.2: Simulation of the average duration of **read** operations as a function of the data size. For small values, the network latency dominates; for large value, the duration converges to the time for transferring the data.

The average duration of the **read** operations is shown in Figure 6.1. As two network roundtrips are needed by every **read**, the minimum expected duration is 400 ms. We note that only when the writer's network latency is about 20 ms or less, will **read** operations take noticeably longer than their minimal duration. This corresponds to a writer that operates at least five times faster than the reader. However, an average **read** operation never exceeds 600 ms.

6.6.2..2 Data size The second parameter that affects the **read** duration behavior is the data transfer time. We have already seen that for small values, **read** operations take longer than their minimal duration only in the presence of very fast **write** operations.

For this simulation, we let a fast writer with 1 ms mean network latency run concurrently to the reader. We vary the data size from 1 KB to 10 MB by multiplicative increments and simulate 16 data points for every 10-fold increase in size. We compare the average **read** duration of our algorithm to the theoretical lower bound, which is achieved by a non-robust algorithm that retrieves the value from one KVS.

The result is depicted in Figure 6.2. It shows that for small sizes, the network latency dominates the time for reading. Here, the read duration corresponds to the time needed for about three network roundtrips and matches the simulation of the reader's latency with much faster concurrent writes described previously. With larger sizes, the data transfer time becomes dominant, the **write** operations take longer, and the probability that the reader

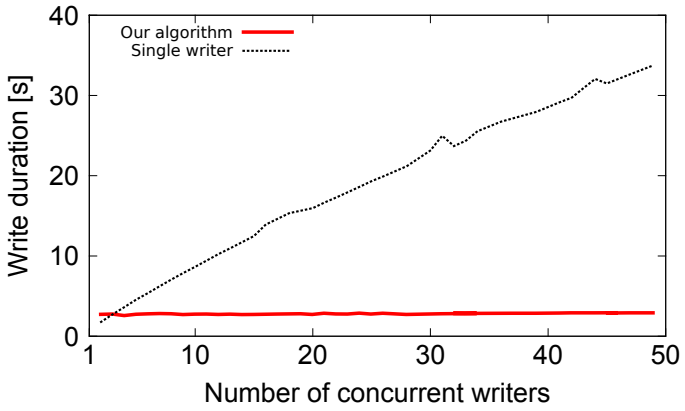


Figure 6.3: Simulation of the average duration of **write** operations as a function of the number of concurrent writers. The single-writer approach with serialized operations is shown for comparison.

runs extra iterations of its loop decreases. For a data size of about 400 KB or more, our algorithm converges to the lower bound. This is because the value is transferred from the KVS only once, and the data transfer time dominates the operation duration.

6.6.3 Write Duration

This simulation addresses the scalability of **write** operations in the presence of multiple concurrent writers. We use a medium data size of 1 MB to illustrate the critical issue of **write** contention. With shorter values, the **put** operations finish quickly and we have not experienced much contention in preliminary simulations. For comparison we also simulate the performance of single-writer replication approaches, which have been considered in the related literature about data replication for cloud storage [2, 17]. These approaches provide the multi-writer capability by agreeing on a schedule with a single writer at any given time. In effect, this causes serial writes.

The network latencies for all writers are 100 ms; data size of 1 MB incurs a delay of 1 s because of the bandwidth constraint, which is imposed on the connection from every writer to the KVS replicas. Figure 6.3 shows the average duration of **write** operations invoked concurrently by a pool of clients, which grows from 1 to 50 clients. The averages are obtained by running the simulations for 30 s. The single-writer algorithm models **write** serialization through agreement, where we ignore the cost of reaching agreement.

For this simulation we use a batched garbage collection scheme, where a

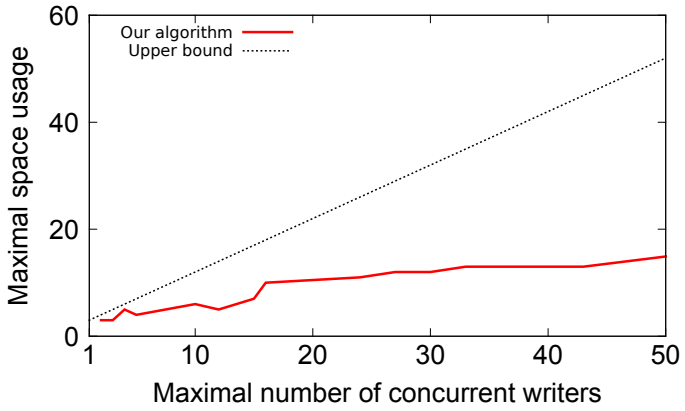


Figure 6.4: Simulation of the maximal space usage depending on the number of concurrent writers. The upper bound is the number of writers plus two according to Theorem 5.

writing client invokes all **remove** operations concurrently. Although such a parallelization is impossible in our formal model, it is a practical optimization feasible with all KVS services we encountered.

The figure shows how the average duration of a **write** in our algorithm remains constant, even with many writers. In contrast, the time for writing in the single-writer approach obviously grows linearly with the number of concurrent writers.

6.6.4 Space Usage

To gain insight in the storage overhead, we measure the maximal space used at any KVS depending on the number of concurrently writing clients. The data size is 500 bytes, and the simulations are run for 50 s.

Figure 6.4 shows the *maximal* space usage at a KVS, where the number of concurrent writers increases from 1 to 50. Space usage is normalized to multiples of the data size. The upper bound from Theorem 5, given by the number of concurrent writers plus two, is included for comparison. The simulation shows that this bound is pessimistic and that the space used in practice is much smaller.

Further investigations show that the *average* space usage lies in the range of 2–5 in this simulation. This behavior can be explained by referring to the **write** algorithm. Concurrent writers indeed leave a large number of temporary keys behind, but the next writer removes all of them during garbage collection. As the time until removal is relatively short, the average space usage is small.

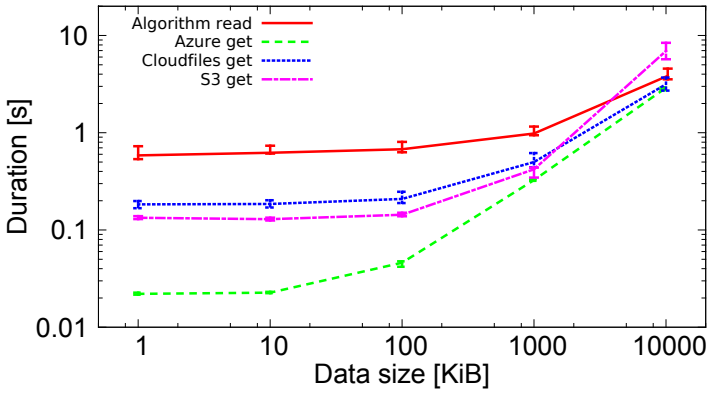


Figure 6.5: The median duration of **read** operations and **get** operations as the data size grows. The box plots also show the 30$^{\text{th}}$ and the 70$^{\text{th}}$ percentile.

6.7 Implementation

6.7.1 Benchmarks

To evaluate the performances of **read** and **write** operations on cloud-storage KVSs in practice, we have implemented the algorithm in Java. The implementation uses the *jclouds* library [66], which supports more than a dozen practical KVS services.

Every client is initialized with a list of n accounts of KVS cloud-storage providers. The client library buffers operations on the KVSs as required by our model. Specifically, when a **read** or a **write** operation triggers a series of operations on the KVSs, these are appended to a dedicated FIFO queue for each one of the n KVSs; for each KVS, the implementation fetches the first operation from its queue and executes it as soon as the preceding one terminates.

The benchmark uses $n = 3$ KVS providers: Amazon S3, Microsoft Azure Storage, and Rackspace Cloudfiles [8, 26, 94]. The client performs two **write** operations with the same key (so as to trigger the deletion of the first version) for 1000 different keys in closed-loop mode, followed by as many **read** operations with the keys written previously. We have instrumented the code to measure the completion time of the individual **list**, **put**, **get**, and **remove** operations as well as the duration of the **read** and **write** operations. The benchmark explores a data size ranging from 1 KiB to 10000 KiB in ten-fold increments.

Figures 6.5 and 6.6 show the results of the benchmark. Closer investigation of these times reveals that the duration of **read** operations is equal to the

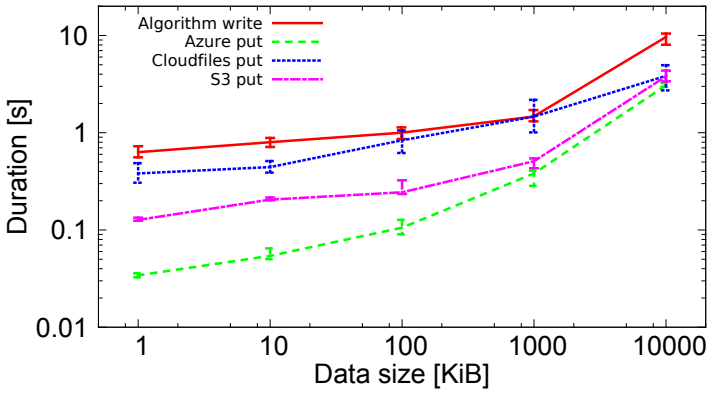


Figure 6.6: The median duration of **write** operations and **put** operations as the data size grows. The box plots also show the 30$^{\text{th}}$ and the 70$^{\text{th}}$ percentile.

duration of the second-slowest **get** plus the duration of the second-slowest **list**. The reason is that the reader only waits for responses from a majority of the providers, and hence ignores the slowest response here. As for **write** operations, we observe that their duration equals twice the duration of the second-slowest **put** operation plus the duration of the second-slowest **list**. We also notice that **read** and **write** operations are faster than the slowest **get** and **put** operations: this can be seen in Figure 6.5, where Amazon S3 **get** operations are much slower than **read** operations for 10000 KiB data size, and in Figure 6.6, where Cloudfiles **put** operations are slightly slower than **write** operations for 1000 KiB input files.

6.7.2 Comparison of Simulation and Benchmarks

To compare the simulations with the behavior of the implemented system, we run an experiment with three KVS replicas and one client that performs 1000 **write** operations followed 1000 **read** operations. The data size is 2 MB. The same scenario is simulated with parameters set to values that were obtained from the experiment.

In particular, the simulation uses the same model as described before, with exponentially distributed network latencies for KVS operations. We measured the network latency of KVS operations excluding the time for data transfer. We assume that the invocation and response latencies of the simulated operations are symmetric and set their mean to half of the measured network latency. Furthermore, we determined the bandwidth of every KVS provider from the measurements of **put** and **get** operations.

For **get** and **put**, the mean network latency for the KVSs is set to 39.4 ms,

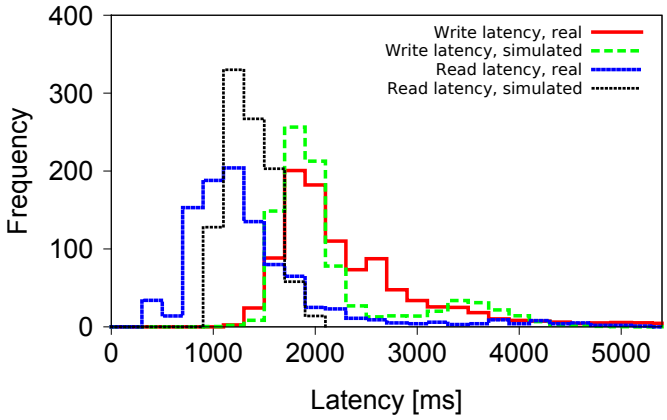


Figure 6.7: Comparison of the duration of **read** and **write** operations for the real system (solid lines) and the simulated system (dotted lines). The graph shows a histogram of the operation durations for 1000 **read** operations (centered at about 1200 ms) and 1000 **write** operations (centered at about 1800 ms).

90.4 ms, and 81.2 ms, respectively. For **list**, the mean network latency is 36.5 ms, 181.1 ms, and 130.9 ms; and for **remove**, network latency is 18.5 ms, 100 ms, and 59.5 ms. The bandwidth limitations for the providers are 6.67 MBps, 2.33 MBps, and 1.5 MBps, respectively.

Figure 6.7 compares the durations of **read** and **write** operations in the experiment and the simulation. The graphs show a good match between the experimental system and the simulation. This reinforces the confidence in the simulation results.

Chapter 7

ACID-RAIN

Large scale computing systems often employ massive data sets that must be spread over large numbers of storage nodes. Clients are provided with read and update access primitives, and to the extent that client transactions access shared data items, the issue of consistency arises. Ideally, we would use a system with ACID transactions [14, 71, 4], because this model facilitates reasoning about system properties and makes possible a variety of high-assurance guarantees. Nonetheless, the ACID model is widely avoided due to efficiency concerns [57].

Today's popular cloud-scale data management systems [28, 25, 101, 34, 89] divide the objects into subsets so as to group objects likely to be accessed by a single transaction into a single subset. Each subset can then be hosted by a single (reliable) entity. Such a structure makes it possible to allow atomic operations for objects in the same subset; more difficult are cases where a transaction spans multiple subsets, and hence multiple nodes.

Were one to support ACID semantics in this case, the most common approach is to obtain ordering and atomicity by making a central certification entity responsible for validating the actions taken when the transaction is ready to commit. This entity can be a reliable replicated service, such as Zookeeper [63] or Corfu [84]. With this structure in place, transactions can perform quite a bit of work optimistically. At commit time, the desired actions and their read/write dependencies are routed through the certification authority, which can then abort actions that violated ACID consistency. However, this kind of certification entity is limited by the throughput of the service, and therefore the approach cannot scale beyond a certain point.

Another option is to use a combination of locking or timestamped version management, together with two phase commit (2PC). This permits atomic operations on objects that reside in different machines. However, 2PC is generally avoided in high availability systems due to performance and fault-

tolerance concerns. When servers fail, a common event in large scale systems, locks may persist, blocking other operations. Specifically, if a lock holder crashes mid-transaction, its peers have to wait until its crash is verified (through the use of server leases), and the lock holder is resurrected and releases the locks, before allowing the transaction to complete (either committing or aborting). Resurrecting a lock holder often involves reloading its state from reliable storage, and until that time, no new transaction with potential collisions can be processed. Optimistic concurrency control has at best a limited impact on these costs, as we'll see in our review of related work in Section 7.1.

In this chapter, we present ACID-RAIN — an architecture for ACID transactions in a Resilient Archive with Independent Nodes. The exact model and goal are described in Section 7.2.

7.0.2..3 Architecture Our approach uses logs in a novel manner. Classically, logs are used to store the state of individual system components, and to restore them upon failure; these are either rollback or roll-forward logs, in which a single server stores its decisions. Our logs, on the other hand, collaboratively describe the state of the entire system. They do this in a distributed

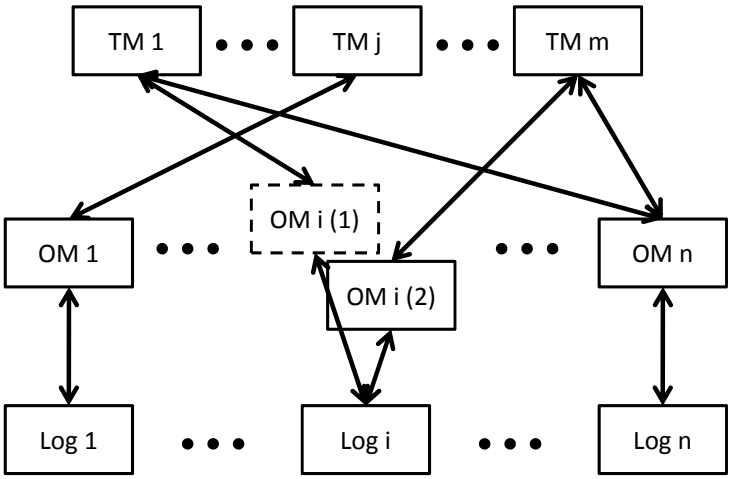


Figure 7.1: Schematic structure of ACID-RAIN. Transaction Managers (TMs) $1, \ldots, m$ access multiple objects per transaction. Objects are managed (cached) by Object Managers (OMs) $1, \ldots, n$. $OM_{i(1)}$ is falsely suspected to have failed, and therefore replaced by $OM_{i(2)}$, causing them to concurrently serve the same objects. The OMs are backed by reliable logs $1, \ldots, n$, to which they store tentative transaction operations for serialization, as well as (later) certification results.

fashion, i.e., one would need to combine all logs in order to materialize the global state. The structure of the system is illustrated in Figure 7.1. Each log is accessed through an *Object Manager* (*OM*) that caches the data and provides the data structure abstraction — exporting read and write operations in the KVS case. *Transaction Managers* (*TM*s) provide the atomic transaction abstraction, sending each individual operation to the object's log (via the OM), and certifying the transaction by checking for conflicts in each log (again, via its OM). We thus separate the consistency of individual objects, maintained with logs, from that of full transactions, achieved by our algorithm. The reliability for each log is achieved with replication chains or state machine replication, abstracted away as reliable logs.

The benefit of our approach is that other than the logs, no system entities are required to be reliable. If an OM crashes, it is replaced by another OM, whose state is restored from the log. Since OMs merely cache the state in the log, safety is not endangered by having multiple OMs acting concurrently. This allows for quick restoration, since we needn't worry about verifying the crash of a potentially slow agent. Should a TM fail, work can be shifted to other TMs, and transactions currently underway can be aborted. If a transaction has been written to its logs and certification is underway, some other TM can take over the certification process: there is no risk of confusion, because certification is a deterministic algorithm that depends purely on the log states. To abort a transaction, a TM *poisons* it by appending poison entries to the relevant logs, thus preserving the invariant that one can deterministically compute the state of any object purely from the logs. The system and algorithm are described in detail in Section 7.3, and its correctness is discussed in Section 7.4.

The ACID-RAIN architecture model scales linearly, allowing system resizing by adding shards (reliable logs and OMs) and enough TMs to avoid bottlenecks. A TM always performs read and write operations purely on the outer (OM) layer, allowing for short latency. Update transactions write back to the logs only at commit time using sequential access, allowing for short latency and high throughput.

7.0.2..4 Prediction By its nature, our system uses optimistic concurrency control (OCC), since OMs respond to concurrent TM instructions with no locks. To improve latency, the OMs serve requests with speculative local data structures, referring to their validated local data structure only for certification. However, the use of OCC exposes us to the risk of aborts in scenarios with contention. We significantly decrease the number of aborts using *predictors* that can foresee the likely access pattern of the transac-

tions. Such predictors can be implemented with machine learning tools that monitor access patterns over time [92]. The predictor can be conservative, predicting a "covering set" of objects bigger than the actual one. False negatives (failure to predict that transaction T_i would access object o) may lead to aborts, but when the predictive component is accurate, all transactions are always committed when there are no failures, and can therefore be performed optimistically, in a lock-free manner.

To leverage prediction, we use object *leases*, where a transaction leases a version of an object for use on an OM it is predicted to access. Note that unlike true locks, employed by servers in other systems, failure to respect an object lease does not violate safety, and therefore does not delay OM restoration on failures.

7.0.2..5 Evaluation We evaluate ACID-RAIN through simulation with the transactional YCSB benchmark [36, 42]. We demonstrate the algorithm's linear (i.e., optimal) scalability, and the effectiveness of using predictions in scenarios with contention. Finally, we demonstrate the importance of fast server replacement upon failure, and why it is important to avoid server leases. The results are detailed in Section 7.5.

7.0.2..6 Contributions To summarize, our contributions are:

- An architecture that employs a novel use of logs to limit reliability to a single tier, allowing fast restoration in case of failure.
- A linearly scalable system that allows high throughput (with serial logs) with low latency (most of the time cache access).
- Use of prediction to obtain good throughput by reducing abort rate in high contention scenarios.

A preliminary version of the work presented in this chapter was submitted for publication.

7.1 Related Work

The holy grail of low-latency high-throughput ACID transactions has long fascinated the data management community. We detail below the most relevant work with respect to this one.

One approach is to realize weaker consistency models that enable better performance, e.g., parallel snapshot isolation in Walter [99], causal+ consistency in COPS [79], and snapshot isolation in Percolator [93]. It is also

possible to avoid transactions altogether, as in PNUTS [30], HBase [101], and other NoSQL systems. However, our target is providing full fledged ACID transactions.

RAM-Cloud [89], Elastras [34], Bigtable[28], Windows Azure Storage [25], HBase [101] and others offer atomic transactions in single shards, i.e., for objects located in a single machine. However, transactions are limited to single machines only in very particular scenarios, so this approach doesn't generally hold. G-Store [35] adds a collocation primitive that allows the user to explicitly move objects together for collocated transactions, and Schism [33] cleverly and automatically collocates objects in order to prevent multi-server transactions. However, real world scenarios, such as social network data, do not always have clear separation and would require constant object migration.

Another approach, used by Megastore and its variants [14, 91], H-store [71], and Spanner [32] is to use two-phase commit for cross-server transactions. Sinfonia [4] uses an architecture similar in many ways to ours, but employs locking to provide atomic transactions, and does not take advantage of prediction as ACID-RAIN does. The downside of these approaches compared to ACID-RAIN is that they require a coordinator that performs transactions on multiple objects to be highly available. This requires consensus for each operation, resulting in high latency. High latency reduces throughput, since conflicting transactions block one another, as we demonstrate in Section 7.5.

Sprint [27] and Hyder [16] order transactions by a global service (a multicast service, and a log, resp.). The result of each transaction, commit or abort, is determined by the order of previous transactions. A transaction commits if and only if it has no conflicts with previous committed transactions. In both cases, the global service used is highly efficient, and sufficient for the target application. However, at a high enough scale, a global service becomes a bottleneck. In contrast, our system has no such bottleneck and achieves unbounded linear scale-out.

The approach of MDCC [74] is close to ACID-RAIN. However, unlike ACID-RAIN, MDCC requires storage nodes to keep the metadata of all transactions ever executed. If the failure detector suspects a transaction to be partially written due to a failure, it initiates a re-execution. To prevent transaction double execution due to a false suspicion, storage servers need to check this history on every vote.

7.2 Model and Goal

7.2.1 Model

Our system is designed to run in a single data center. We assume unreliable servers that may crash or hang, in an asynchronous, loss-prone network. To accommodate reliable storage, we employ reliable, available, sequentially consistent logs, as explained in 7.3.2.

7.2.2 Service

The system exposes a transactional data store supporting serializable transactions. A client invokes a begin-transaction command, followed by the transaction's operations. Each operation is either a read (e.g., a field from a table), an update (e.g., setting the value of a field in a table or adding a key to a key-value store) or a read-modify-write (e.g., pushing or popping a queue element). Finally the client invokes the end-transaction command, and the system responds with either a commit or an abort. Servers are equipped with predictors that predict which objects a transaction is likely to touch during is run.

In order to achieve progress, the system should be *obstruction-free*. That is, there should exist a time T after which, if only a single client remains active, all its transactions eventually either commit or abort and any new transactions always commit. This is true regardless of the previous system's state and previous behavior of other clients, which might, in particular, be in the midst of transactions. If the logging service is operational, and the predictive layer is accurate, and there are no failures, transactions complete without obstruction, and never abort.

7.3 ACID-RAIN

We now describe the operation of ACID-RAIN. The overall structure of the system is illustrated in Figure 7.1, and we enumerate the system's elements in Section 7.3.1. We describe the specification of a reliable log in Section 7.3.2, and proceed in Section 7.3.3 to overview a simplified version of the algorithm, skipping resilience issues and prediction. Then we detail the resilient TM and OM algorithms in sections 7.3.4 and 7.3.5, respectively, and the prediction mechanism in Section 7.3.6.

We use the following notation to describe concurrency. When a $\|$ symbol precedes a function call, the function is called asynchronously (e.g., by a

different thread), and the current thread's control proceeds immediately to the next line. Additionally, in a loop of type **parForeach**, all the loop's iterations are run concurrently, and control proceeds after the loop when all of them complete.

Remote function invocations, including the ones running in **parForeach** loops, can time out (due to crashes or message loss) and return $\perp$.

7.3.1 System Structure

We detail the components of the system below.

1. Clients, typically front-end machines — invoke the transaction operations and begin/end it.

2. Logs — reliable and available logs. Append entries, and update registered servers when entries are appended.

3. Transaction managers (TMs) — implement the transaction abstraction. TMs receive instructions from the client to start and end a transaction, and operations to perform on individual objects. They return object values and the commit/abort result. A TM accesses the OMs to read objects, write objects and coordinate transaction certification.

4. Object managers (OMs) — implement the object abstraction (e.g., table or queue). They maintain a cache of the objects in the logs. They return the values requested by the TM on reads and acknowledgements on updates.

5. Membership monitors — in charge of deciding and publishing which machines perform which roles, namely which machines run the log and OM for each shard, and which TMs are available. Any client can access any TM for any given transaction. Other than the logs, server role assignment may be inconsistent. Each object (transaction) is supposed to be managed by a single OM (TM, resp.) at a given time, but this may change due to an unjustified crash suspicion whereupon an object (transaction, resp.) may temporarily be managed by two OMs (TMs, resp.) that do not know of one another.

In Figure 7.1 there are n reliable logs and object managers associated with the logs. One, OM i, is falsely suspected as malfunctioning, so it has two copies, $OM_{i(1)}$ and $OM_{i(2)}$. There are also m transaction managers, each running a transaction that touches a different set of OMs.

Note that in an implementation of the system one may use multiple OMs per log, dividing the log's object set, or the other way around, have multiple logs report to a single OM. The choice depends on the throughput of the specific implementations chosen for each service. Here we use a 1:1 mapping for simplicity of presentation.

7.3.2 Log Specification

ACID-RAIN uses log servers for reliable storage of data. Each log server provides a sequentially consistent log object, i.e., update operations are linearizable, but reads may return outdated results. A client of the log object works as follows. To append an entry, it invokes append(e), and the log appends the entry e according to the arrival order, associating it with a sequence number, starting from 0 for each log.

In order to track the log state, a client registers to the log. The log sends to each registered client all entries as (i, e) (sequence number and entry content), from the first one in the log, to its end, and then new entries as they arrive. The client is then able to construct a local copy of the log to work with. Note that due to processing time and network delays, the client's copy of the log may be out of date.

A client may also perform garbage collection (GC) on the log, by invoking truncate(i) that causes the log to discard all entries earlier than sequence number i. This sequence number is now the first entry of the log (for new registering clients, for example). Sequence numbers are not changed due to a GC event (nor for any other reason).

Such logs may be implemented with chain replication [102], with the Isis^2 replication library [20], or using Paxos or a similar SMR algorithm [77, 70], however we abstract this away, and assume logs are always available.

7.3.3 Simplified Algorithm

We now describe a simplified version of the algorithm. A flow diagram illustrating the algorithm's progress is given in Figure 7.2.

When receiving a begin-transaction from a client, the TM assigns the transaction a unique identifier *txnID* and awaits the transaction's operations. It then services the operations by routing them to the appropriate OMs. Each operation is sent to the OM in charge of the object, along with the transaction ID. The response is delivered back to the client.

Each committed transaction is assigned a timestamp. When reading an object, the timestamp of the latest transaction that wrote this object is

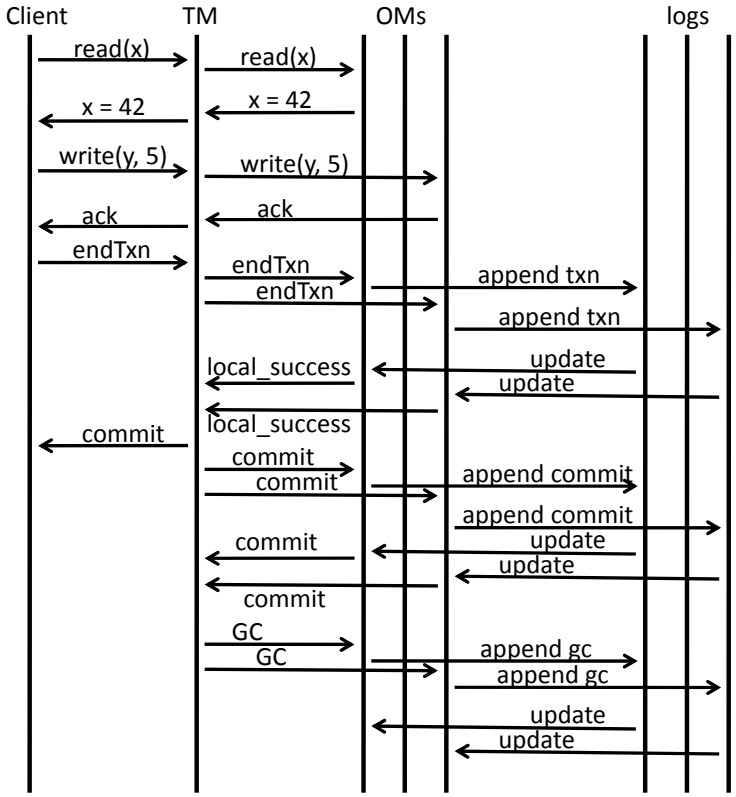


Figure 7.2: An example flow of the simplified algorithm. A front end runs a transaction that reads object x and writes object y. The client ends the transaction, and the TM certifies it with both relevant OMs, both going through the appropriate logs before returning their local results. The TM then sends the commit result to the client. Later, it marks the transaction in the logs (via the OMs) as committed, and then as ready to GC.

returned to the TM. The TM calculates the transaction's timestamp by incrementing the largest timestamp returned to it in any of the transaction's operations. Once a transaction is done, the TM also forms its *log set*, the set of logs in charge of the shards it touched.

Certification

Once a TM receives an end-transaction instruction from a client, it notifies relevant OMs, detailing the transaction's timestamp and log set. When it receives an end-transaction instruction, an OM appends to the log of its shard an entry consisting of the *txnID*, its timestamp, its read- and write-sets (read-set with timestamps read, write-set with written values), and its log set.

If the transaction was written to all logs, and it does not collide with previous transactions on any of them, it is construed as having committed. Collisions are violations of read-write, write-read or write-write order, including circular dependencies, and can be checked by comparing timestamps. In our architecture, each OM can only detect local collisions, so the result of the transaction can only be certified by combining the information from multiple logs.

To certify a transaction, the TM goes over all of *logSet* and confirms the transaction has no collisions in any of the logs. A transaction that reads an object updated by a concurrent transaction, cannot be certified until the latter was certified.

A TM (usually the committing TM), asks all the relevant OMs for their local collision result. If none of them collided, then the transaction has committed, otherwise it has aborted. The TM then instructs the OMs to place the transaction result in the logs. In case of a crash of a TM or an OM or a missing result entry (due to message loss), resulting in a partially certified transaction, another TM may read the uncertified entry in one of the logs, find *logSet*, and restart/continue the certification process. It is okay to have multiple result entries for a transaction — they will be the same. Note that each transaction entry is written once, with no retries on failure, to avoid duplicates. A lost transaction entry will therefore trigger an abort.

Garbage Collection

Running our algorithm for a long time leads to lengthy logs. This has two drawbacks. First, an agent that registers itself with the log has to download and run this long log. Second, storage bounds prohibit infinite-length logs. Therefore we wish to occasionally compact the log by summarizing its prefix, placing this summary in the log and GCing the prefix. However, this compaction must not break transaction certification. Each transaction should be either committed or aborted in all its logs, and therefore cannot be removed from any of them before the result is published.

To allow GC, an agent (usually the committing TM) goes over *logSet* to check that the certification result was appended. If it was not, the TM certifies the transaction by itself. If it was, it proceeds to clear the transaction for GC by appending to each log in the log set the tuple $(txnID, \textsc{gcOk})$. Again, in case of a crash of the TM, or a lost GCOK-tuple, another TM may read the entry in one of the logs, learn the log set, and restart/continue the clear-to-GC process. Multiple TMs may attempt to move the system from one state to the other until the transaction is done (committed or aborted) and GCed.

7.3.4 Transaction Manager

The resilient TM's operation is described in Algorithms 13–14. To limit the complexity of the code, we consider a KVS data structure, so each operation either reads a key (tmRead), or writes a value to a key (tmWrite). The extension to arbitrary data types is immediate. Additionally, the TM handles tmBeginTxn and tmEndTxn.

When beginning a transaction, the TM chooses a unique identifier for it, and returns this ID to the client. All the transaction's operations are tagged with this ID.

It then uses a predictor to estimate its read and write sets, and chooses a predicted timestamp by conferring with the relevant OMs and leasing the objects. The leasing mechanism is described in Section 7.3.6, but omitted from the pseudo-code for readability.

When asked to read an object, the TM contacts the OM in charge to retrieve the object's value and latest written version, and returns the value to the invoking client. When asked to write an object, the TM calls the OM in charge to store the write. It maintains locally the highest timestamp each transaction saw (for certification), and the set of logs that are in charge of the objects touched by the transaction (for failure recovery purposes).

Finally, when instructed to end a transaction, the TM proceeds as follows. If the transaction belongs to the TM (not a recovery), the timestamp chosen is one higher than the max TS seen, and an end-transaction instruction is concurrently sent to all relevant OMs with this timestamp and with the transaction's log set.

Any invocation returning a local failure triggers an immediate abort, since other results will not change this outcome. This is not shown in the pseudo-code for brevity.

Based on the local results obtained by the omEndTxn invocations, the TM learns the result of the transaction. It returns this result to the client, and the proceeds to complete the end-transaction. First it appends the result (either commit or abort) to all logs (via the OMs). Then it notifies each OM the transaction is ready for GC. Note that when committing, the transaction may have disappeared from all logs, due to concurrent garbage collection. While this situation is inherently possible for a system with bounded memory, it is unlikely, and requires the return of an UNKNOWN result to the caller.

A slightly different version of tmEndTxn, is called when the client wants a TM to end a transaction started by another TM. In this case, the log set is also included in the parameters, so the TM knows where the transaction's objects reside, and the function does not return a result, since it is called by an internal system component that wants to clear the transaction, and not

by a client.

With message loss and OM failure, it is possible that an end-transaction routine ends with only a subset of its transactions written to the logs, so OMs cannot calculate their local results, thus preventing progress. We solve this by poisoning a transaction in the relevant shards. Poisoning causes the shard to return a local failure (as if the transaction has collided), and allows the TM to abort the transaction. If the transaction was already written to the log, the poison instruction is ignored, and the transaction is processed as usual.

the TM concurrently and repeatedly sends poison instructions to each of the OMs, until a local result is returned to the end-transaction invocation.

Algorithm 13: Transaction Manager — Client interface (1/2)

```
1  initially
2      ∀x : logSets(x) = ∅
3      ∀x : TSs(x) = ⊥
4      ∀x : results_x = [ ]

5  function tmBeginTxn()
6      txnID ← UUID()                                   (Universally unique identifier)
7      TSs(txnID) ← (−1, −1)                            (Latest timestamp seen by transaction txnID)
8      return txnID

9  function tmRead(txnID, objID)
10     logSets(txnID) ← logSets(txnID) ∪ {logInCharge(objID)}
11     (val, writeVer) ← omRead_{omInCharge(objID)}(txnID, objID))
12     TSs(txnID) ← max(TSs(txnID), writeVer)
13     return val

14 function tmWrite(txnID, objID, newValue)
15     logSets(txnID) ← logSets(txnID) ∪ {logInCharge(objID)}
16     omWrite_{omInCharge(objID)}(txnID, objID, newValue))
17     return ACK
```

7.3.5 Object Manager

The OM operation is comprised of two interacting components, an interface that interacts with the TM, and a monitor that monitors the log.

7.3.5..7 Interface The interface component, shown in Algorithm 15, responds to TM invocations of omRead, omWrite, omEndTxn, and omPoison. The OM is not explicitly informed on the beginning of a new transactions, but deduces it when receiving transaction operations. For each operation, it returns the required response according to the data structure's state, and registers the operation as part of the transaction. The object state is not affected (only for OCC, as explained below).

Algorithm 14: Transaction Manager — Client interface (2/2)

```
18 function tmEndTxn(txnID)
19     results_txnID ← (⊥)^|logSets(txnID)|
20     parForeach i ∈ logSets(txnID) do
21         ||tmSendPoison(txnID)
22         if TSs(txnID) ≠ ⊥ then                                   (my transaction)
23             results_txnID[i] ← omEndTxn_omInCharge(i)(txnID, logSets(txnID), inc(TSs(txnID)))

24     if COMMIT ∈ results or results ∈ {LOCALSUCCESS, MISSING}^|logSet| then
25         result ← COMMIT
26     else if ABORT ∈ results or results ∈ {LOCALFAILURE, MISSING}^|logSet| then
27         result ← ABORT
28     else
29         result ← UNKNOWN
30     || completeEndTxn(txnID)
31     return result

32 function tmEndTxn(txnID, logSet)
33     logSets(txnID) ← logSet
34     results_txnID ← (⊥)^|logSets(txnID)|
35     parForeach i ∈ logSets(txnID) do
36         ||tmSendPoison(txnID)
37     || completeEndTxn(txnID)

38 function inc(ver)
39     return (ver[0] + 1, clientID)

40 function tmSendPoison(txnID)
41     while results_txnID[i] = ⊥ do
42         wait for timeout, allowing entry to take effect
43         omPoison_omInCharge(logSets(txnID)[i])(txnID)

44 function completeEndTxn(txnID)
45     while results_txnID ∉ {COMMIT, MISSING}^|logSet| ∪ {ABORT, MISSING}^|logSet| do
46         if COMMIT ∈ results_txnID or results_txnID = (LOCALSUCCESS)^|logSet| then
47             parForeach i s.t. results_txnID[i] = LOCALSUCCESS do
48                 results_txnID[i] ← omCommit_omInCharge(logSets(txnID)[i])(txnID)
49         else if ABORT ∈ results_txnID or results_txnID = (LOCALFAILURE)^|logSet| then
50             parForeach i s.t. results_txnID[i] ∈ {LOCALFAILURE, LOCALSUCCESS} do
51                 results_txnID[i] ← omAbort_omInCharge(logSets(txnID)[i])(txnID)

52     foreach i ∈ logSets(txnID) do
53         || clearToCompact_omInCharge(logSets(txnID)[i])(txnID)
```

When it receives an end-transaction invocation, the OM appends the transaction entry to the log, and waits for its result to appear in a local variable. This local variable is updated with the result by the monitor component once it can be calculated, as detailed later. When it receives a transaction result (COMMIT or ABORT) from the TM, the OM executes the transaction if necessary, thus updating its data structure, and appends the result to the log. It also removes the entry from its local entry log and marks it committed, for reasons explained later. When receiving a clear-to-compact invocation, the OM locally marks the transaction as ready to GC, a fact used by the monitor to clear the log.

A poison request causes the OM to append a poison entry to the log, and return immediately. This is done to release a concurrently pending end-transaction routine.

Algorithm 15: Object Manager — Interface

```
1  initially
2      ∀x : RS(x) = ∅
3      ∀x : WS(x) = ∅
4      ∀x : objs(x) = ⊥
5      ∀x : txns(x) = MISSING
6      myLog = ID of shard's log

7  function omRead(txnID, objID)
8      if txns(txnID) = ⊥ then txns(txnID) ← RUNNING
9      if objs(objID) ≠ ⊥ then
10         (val, readVer, writeVer) ← objs(objID)          (We read the latest written version)
11     else
12         (val, readVer) ← (⊥, (0, 0))
13     RS(txnID) ← RS(txnID) ∪ {(objID, writeVer)}
14     return (val, writeVer)

15 function omWrite(txnID, objID, val)
16     WS(txnID) ← WS(txnID) ∪ {(objID, val)}
17     return WRITEACK

18 function omEndTxn(txnID, TS)
19     if txns(txnID) = ⊥ then
20         return MISSING
21     else if txns(txnID) ∉ (RUNNING, PENDING) then
22         return (txnID, txns(txnID))
23     txns(txnID) ← PENDING
24     append_myLog((TXNENTRY, txnID, RS(txnID), WS(txnID), TS))
25     wait until txns(txnID) ∈ {LOCALSUCCESS, LOCALFAILURE}
26     return txns(txnID)

27 function omPoison(txnID)
28     if txns(txnID) = ⊥ then
29         return MISSING
30     else if txns(txnID) ∈ (RUNNING, PENDING) then
31         txns(txnID) ← PENDING
32         ‖ append_myLog((POISON, txnID))

33 function omCommit(txnID)
34     if txns(txnID) ≠ ⊥ then
35         txns(txnID) ← COMMIT
36         execute transaction
37         remove txnID from localEntryLog
38         ‖ append_myLog((txnID, COMMIT))

39 function omAbort(txnID)
40     if txns(txnID) ≠ ⊥ then
41         txns(txnID) ← ABORT
42         remove txnID from localEntryLog
43         ‖ append_myLog((txnID, ABORT))

44 function clearToCompact(txnID)
45     if txns(txnID) ≠ ⊥ then
46         readyToGCTxns ← readyToGCTxns ∪ {txnID}
```

7.3.5..8 monitor The OM monitors the log with the code given in Algorithm 16. On initialization, it looks for the first summary in the log, and loads it if it finds one. Then, it concurrently runs two functions. The first, parseLog(), goes through the log, and parses the entries, interpreting the log and updating local variables accordingly. It puts transaction entries in a local queue *localEntryLog*. On POISON and ABORT entries it removes the transactions from the local queue, and on COMMIT it performs the transaction (if it was not committed already by the interface component, i.e., if it was committed through another OM). Additionally, parseLog() occasionally adds summaries, and invokes log truncation when possible. To do that, it maintains two pointers, *iStart* pointing to the latest summarized prefix (prefix whose summary is found in a later entry), and *iClear* pointing to the first entry that is not ready to be GCed.

The second function, handleLocalLog(), is in charge of checking whether a transaction is locally successful or not. It monitors *localEntryLog* for transactions that were appended to the log, and whose dependencies (prior entries in *localEntryLog* that touch colliding objects) are resolved. Once it finds such entries, it decides on the transaction's local result and locally stores it. This result is read by the interface component, which responds to the TM.

Speculation

In order to facilitate rapid progress when there are no failures (the common case), the OM maintains two copies of the data structure — the validated copy, which is the one in the code above, and a speculative copy that applies all updates once they happen. The latter is omitted from the pseudo-code for brevity. An OM responds to the transactions' operations (reads and writes) from its speculative copy, acting as a non-consistent cache, and applies committed changes to the validated copy. Note that when using this mechanism, clients may observe inconsistent data during the course of a transaction, however only transactions with consistent views may commit. Once a transaction aborts, the OM discards any associated speculative data.

7.3.6 Prediction

ACID-RAIN leverages predictable transactions by employing leases at the OM layer. Note that these leases are advisory: failure to respect them harms liveness (aborts can be triggered), but not safety. This means they can be ignored, and therefore they do not create a risk of deadlocks.

When a transaction starts, a black-box machine learning mechanism predicts its read and write sets, and leases them to the transaction. Given these

Algorithm 16: Object Manager — Log monitor

```
1  function omInit()
2      iStart ← first index in log
3      iSummary ← iStart
4      while log[iSummary] isn't a summary pointing to iStart or later and it's not the log's end
       do
5          iSummary ← iSummary + 1
6      if log[iSummary] is a summary pointing to iStart or later then
7          load summary in log[iSummary]
8          iStart ← where log[iSummary] points to
9      ||parseLog()
10     ||handleLocalLog()
11 function parseLog()
12     iLatest ← iStart − 1
13     iClear ← iStart − 1
14     while TRUE do
15         iLatest ← iLatest + 1
16         entry ← log[i_lastest], or wait until it's ready
17         if entry = (TXNENTRY, txnID, RS, WS, TS) then
18             localEntryLog.append(txnID, RS, WS, TS)
19         else if entry = (txnID, ABORT) then
20             txns(txnID) ← ABORT
21             remove txnID from localEntryLog
22         else if entry = (txnID, COMMIT) then
23             if txns(txnID) ≠ COMMIT then
24                 execute txnID
25                 remove txnID from localEntryLog
26         else if entry = (POISON, txnID) then
27             if txns(txnID) ∈ {MISSING, RUNNING} then
28                 txns(txnID) ← LOCALFAILURE
29         else if entry is a summary then
30             iStart ← iLatest
31         while log[iClear] ≠ (TXNENTRY, ...) or log[iClear] ∈ readyToGCTxns do
32             iClear ← iClear + 1
33             if log[iClear] is a transaction with ID txnID then
34                 readyToGCTxns ← readyToGCTxns \ {txnID}
35         if iStart − iLatest > summaryThreshold then
36             append_myLog(summary of data structure)
37         if min(iClear, iStart) − lastClear > threshold then
38             truncate_myLog(min(iClear, iStart))
39             lastClear ← iClear
40 function noCollisions(RS, WS, TS)
41     foreach (obj, ver) ∈ RS do
42         if writeVer(obj) ≠ ver then
43             return FALSE
44     foreach (obj, ver) ∈ WS do
45         if max(writeVer(obj), readVer(obj)) > TS then
46             return FALSE
47     return TRUE
48 function handleLocalLog()
49     while TRUE do
50         wait for entry (txnID, RS, WS, TS) in localEntryLog with no previous dependencies
51         if noCollisions(RS, WS, TS) then
52             txns(txnID) ← LOCALSUCCESS
53         else
54             txns(txnID) ← LOCALFAILURE
```

access predictions, the TM runs a simple two-phase protocol with the OMs to lease (reserve) a set of object versions valid at some instant in logical time, using an ordering mechanism introduced by Lamport [76]: When starting a transaction, the TM interrogates the OMs about all objects it is predicted to access. Each OM proposes a logical time at which the transaction could be executed (a logical timestamp larger than the latest version of the requested objects). The TM computes the maximum over these times (this is the same method used in Algorithms 13–14), and requests the OMs to reserve the objects with this logical time, its *predicted commit timestamp*. Each OM leases the objects iff they are not reserved already with a larger timestamp (due to concurrent transactions), and returns accept/deny. Should this fail (i.e., upon detection of a potential deadlock), the process can be repeated until the leases are successfully acquired. The TM then proceeds to run the transaction, sending its operations to the OMs. If two or more transactions concurrently attempt to access the same object, they are ordered by their predicted commit time, and each waits until the outcome of its predecessor is known by the OM. This transaction ordering will be consistent across the set of OMs at which the transactions conflict.

Access-set misprediction can result in situations in which an OM receives an operation that accesses an object that was not reserved. If the object is free, or reserved with a smaller timestamp, the OM can reserve it on the spot. If it is already reserved with a higher timestamp, however, the unpredicted access is denied, and the transaction must either abort. or try to shift to a larger timestamp.

In the interest of shortening latency, a TM can speculatively start performing transaction operations before the lease phase is complete. It would have to abort, however, if a version discrepancy is detected which would, in retrospect, require it to have returned a different version for a completed operation.

7.4 Correctness

We provide proof roadmaps of our system's correctness.

Serializability

Committed transactions are serializable, i.e., an execution of the system is equivalent to a serial execution of the transactions.

An execution of our system is a series of read, update, begin-transaction and end-transaction invocations and responses, as seen by clients, together

with internal protocol messages and operations. To prove serializability, we need to show that the execution is equivalent to one where transactions are executed serially (with no overlap). We consider only committed transactions.

To prove our statement, we reduce the execution to match the requirements of Theorem 2 in [90], adapted as Theorem 6:

Theorem 6. *Given an execution e with transactions $t_1, \ldots, t_n$, form a dependency graph D with vertices $v_1, \ldots, v_n$, and directed edges E. An edge (v_i, v_j) is in E if and only if transactions t_i and t_j both access some object o, and at least one of them updates o. If the dependency graph D is acyclic, then the execution e is serializable.*

The history of committed transactions can be modeled as a directed graph D s.t. each transaction is a vertex, and a directed edge connects transactions i and j if both update an object o, or if one reads and the other updates an object o. It remains to show that the graph D is cycle free. Each transaction in our algorithm has a timestamp, and would not commit if there exists an edge to it in the graph from a transaction with a smaller timestamp. Therefore, any path in the graph passes through strictly increasing timestamps, and there cannot be a cycle. Therefore, according to Theorem 6, any execution of our system is serializable.

We note that the prediction mechanism does not violate safety. On OM or TM restoration due to a failure, lost leases can be safely ignored, preserving safety — collisions are always checked at commit time according to the logs.

Obstruction Freedom

Finally, we show that our algorithm is obstruction free in the absence of server failures. That is, when run uninterrupted from any moment (possibly after previous server failures), a single client can always commit transactions. We show that if there exists a time t_{stable} after which no failures occur, a client c running alone will successfully commit a transaction. With no failures, all servers respond in a timely fashion, and so after sufficiently long, all timeouts expire. Then all failed agents are replaced, and hanging transactions are resent to certification. Specifically, for every transaction i that was submitted (end-transaction) to an OM but never certified, the OM invokes an end-transaction at some TM. The TM then poisons the transaction in each relevant OM, and checks the result, and either commits or aborts the transaction. This goes on for every pending transaction at every OM, eventually leaving the OM queues empty. When client c next reads an object, it receives an up-to-date value, and when it commits, each OM appends the transaction

to its log, resulting in a a local success, followed by a commit result in the TM.

Abort Freedom

With perfect prediction and no failures, transactions never abort. Before starting, transactions contend on predicted commit timestamps, and eventually each will successfully lease its objects. If the predictive mechanism is accurate, and guesses in advance all the objects needed by the transaction, the transaction thus obtains leases for all its objects before starting. Thereafter, as transactions run, they only wait on transactions with lower timestamps. This prohibits deadlocks: if transaction Y waits for transaction X at some OM, X must have a smaller timestamp, hence wait-cycles cannot arise. Finally, since no failures occur, waiting time for an operation is bounded, and if a TM failed earlier, its leases soon expire.

7.5 Evaluation

We use a custom-built event-driven simulation to evaluate the architecture of ACID-RAIN. We simulate each of the agents in the system — clients, transaction managers, object managers and reliable logs. For every run, we set an average transaction per unit-time rate (TPUT), and transactions arrivals are governed by a Poisson process with the required TPUT. In order to avoid second-order effects (e.g., increased latency due to retries triggered due to latency), aborted transactions are not retried, so the average number of incoming transactions remains constant, independent of the commit ratio. Unless noted otherwise, each transaction touches 10 objects out of millions.

Our experiments are an adaptation of the transactional YCSB specification used in [36] and [42], based on the original (non-transactional) YCSB workloads [31]. Each transaction has a set of read/update operations spread along its execution. Object accesses follow one of three different random distributions — (1) uniform, where each object is chosen uniformly at random among the available objects, (2) Zipfian, and (3) hot-zone, where some of the objects belong to a so called hot-zone, and each access is either to the hot-zone, or outside of it (chosen uniformly at random within the picked zone).

7.5.1 Latency and Throughput

As the TPUT increases, queues form at the logs, resulting in an increased commit time. Figure 7.3 demonstrates how this latency increases for a given number of shards, until reaching a point where the system finally can't keep up, at which point queue lengths and delays increase without limit. The figure depicts how the latency closely follows the length of the logs' append queues. Increasing the number of shards decreases the latency for a given workload and postpones the saturation point, allowing the system to accommodate a higher workload.

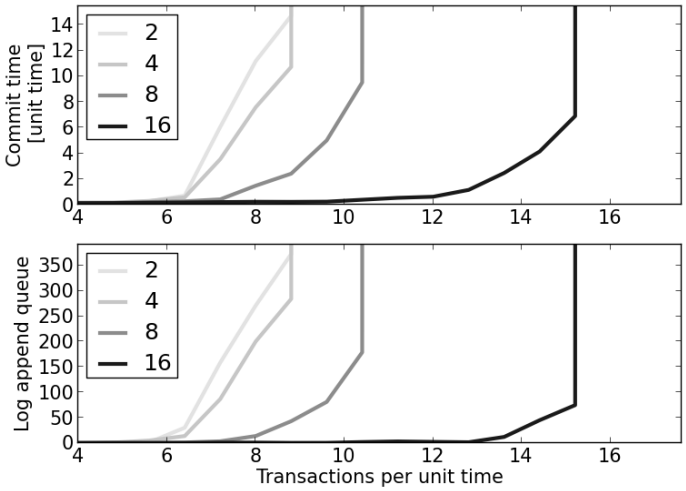


Figure 7.3: Running with 500,000 objects, we increase the rate of incoming transactions, each touching a random set of 10 objects. Increasing the number of shards (2, 4, 8, and 16) improves latency as it decreases the average queue length at the logs.

7.5.2 Scalability

To evaluate the scalability of ACID-RAIN, we measure the maximal TPUT it can accommodate with an increasing number of shards (with 3 reads and 3 writes per transaction of 10^5 objects with uniform access). The result, depicted in Figure 7.4 demonstrates a linear scaling. This is expected, as the conflict rate in is negligible, and our system is scaled without forming any bottlenecks.

We compare ACID-RAIN with the approaches of (1) using 2PC and highly available independent TMs, implemented as replicated state machines

and (2) a global log. In both cases, we allow the systems to skip garbage collection (while ACID-RAIN does perform it). We simulate highly available TMs by increasing the TM's latency to that of a single point to point message (reality would require at least RTT for Paxos or an equivalent, i.e., even worse). To simulate a global log, we bound the TMs' total throughput to about 400 operations per unit time. All other parameters are identical.

While the parameters we choose are arbitrary, the trends are apparent; choosing other parameters would provide similar results, though perhaps at different scales. Improving the efficiency of the highly available TM or the global log would allow them to handle more load than in this example, but they would both reach a bottleneck, at some point.

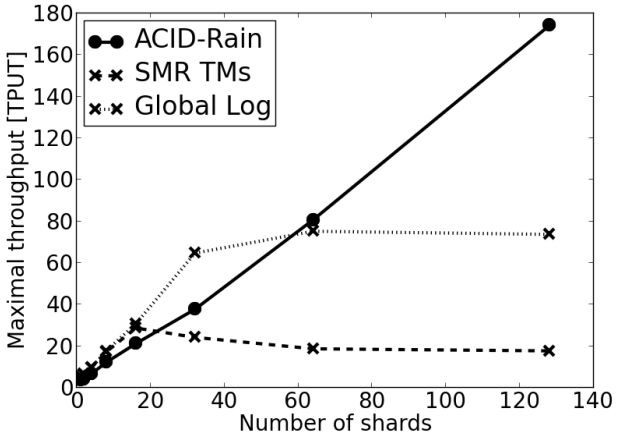


Figure 7.4: For an increasing number of shards, we run multiple simulations to find the maximal TPUT the system can handle. We observe linear scaling for ACID-RAIN, whereas 2PC and global log reach a bound.

7.5.3 Collision Effects

We demonstrate the system behavior under different workloads. In all runs we use an incoming workload well below the system's capacity with 16 shards.

The simulation is faithful to the algorithm, with the exception of a small shortcut — the OM grants leases by arrival time rather than by timestamp. This change results in deadlocks in high contention scenarios, and these are resolved with timeouts. Granting leases by timestamp can expedite deadlock detection.

We compare different predictor qualities to show the importance of prediction. We vary the accuracy, i.e., the ratio of objects the predictor leases

in advance, to evaluate the importance of the prediction mechanism. Separately, we demonstrate that if the predictor requests leases on more objects than necessary (slack), efficiency decreases only in extreme circumstances. This means that real world predictors can decrease their miss-rate by enlarging the predicted object set.

Accuracy

First, we consider uniform random load (Figure 7.5). For a varying number of objects, we measure the commit rate. We see how commit rate drops as the number of objects becomes small. Note that a birthday-paradox effect causes significant collision rate even with a large number of objects. Better prediction means a better commit ratio, however even with perfect prediction deadlocks appear with small numbers of objects.[1]

With a hot-zone of 1000 (Figure 7.6), increasing the probability of hot-zone access increases the abort rate. Note that at probability 1.0 the rates are significantly smaller than in the uniform random case (with 1000 objects), since with the hot-zone all accesses to the hot-zone go through a single OM that becomes a bottleneck, causing longer queues and hence more aborts. On the bright side, since object access collisions occur mostly in a single shard, the leases prevent deadlocks and result in perfect commit ratio with perfect prediction.

Finally, we investigate the effect of a heavy tail Pareto distribution with varying α. The high collision rate causes the certification rate to decrease. Therefore in this case, we measure rates, normalized by the transaction arrival rate, rather than the commit ratio. The results are shown in Figure 7.7. As the alpha parameter increases, the distribution produces higher contention. Avoiding prediction results in a full certification rate, but many aborts. Using prediction produces a better commit rate, until the contention becomes so high that the leases decrease the certification rate. At this point, despite a high abort rate, the commit rate without prediction becomes better.

Slack

As noted earlier, we define slack to be the ratio of the size of the predicted object access set to the actual object access set. In Figure 7.8 we compare the effect of using a perfect predictor (slack=1) with predictors that overpredict by factors of 2 and 4. The impact of overprediction is surprisingly minor, a finding that should make it easier to create a practical predictor.

[1] With our full leasing algorithm these deadlocks would be eliminated; here, we simply abort when they occur.

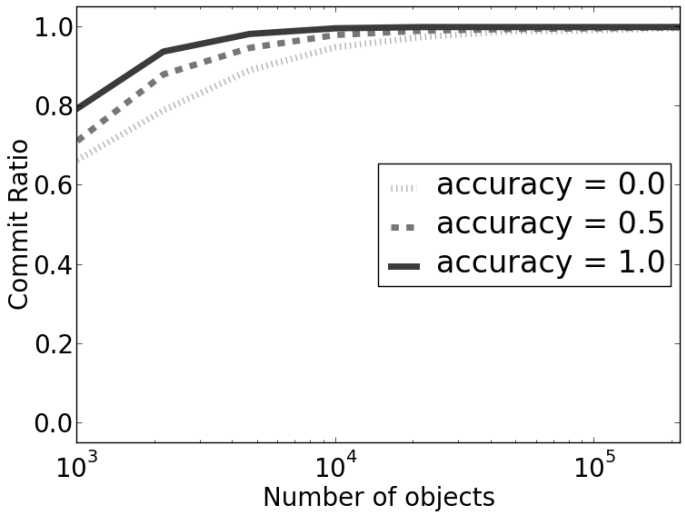


Figure 7.5: System behavior under a uniformly random workload. With poor prediction quality and a small number of objects the system observes high collision rates, and hence high abort rates.

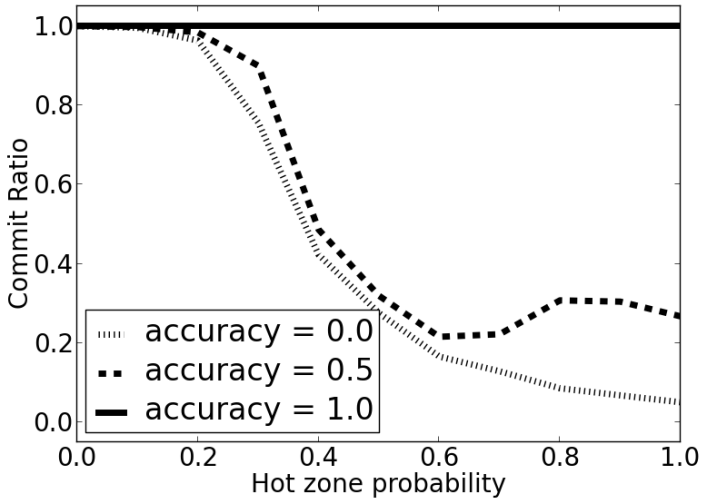


Figure 7.6: System behavior under a hot-zone workload, where a small subset of the objects are accessed with increasing probability. With poor prediction quality we observe high collision rates, and hence high abort rates.

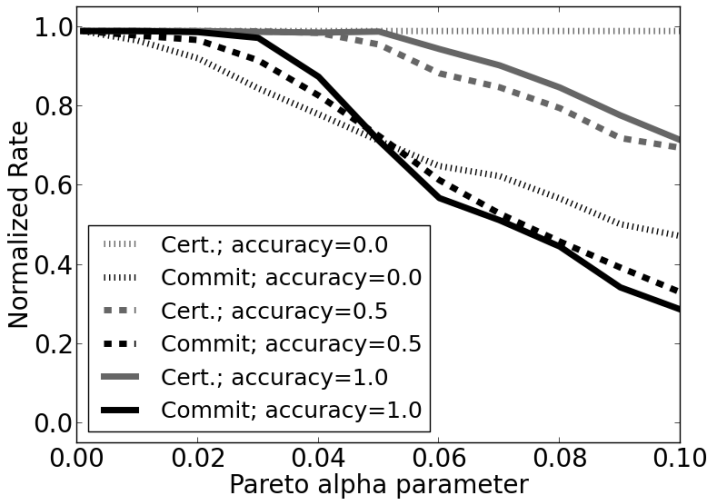


Figure 7.7: System behavior under a Pareto workload. With poor prediction quality, as the Pareto parameter increases, we observe high collision rates, and hence high abort rates.

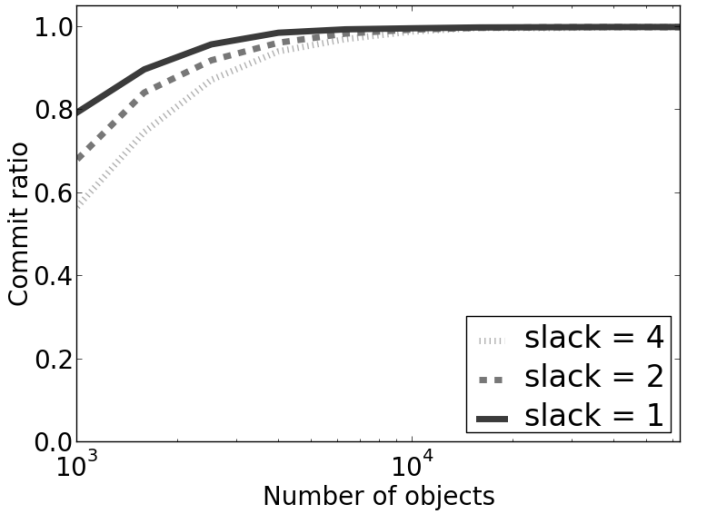


Figure 7.8: Commit ratio for an increasing number of objects and predictors with different slack values, predicting the correct access sets, twice and four times the required objects. Even with potentially high collision rates (few objects), commit ratios mostly remain high. Only for small numbers of objects, and with high slack, does the commit rate fall significantly.

7.5.4 OM Crashes

The efficiency of the system is inversely linear in the time it takes to replace a failed server. Until it is replaced, the objects for which the server was responsible will be inaccessible, blocking any transactions that touch them. This is demonstrated in Figure 7.9, where a single OM crashes, and is replaced. We use 5 OMs so that the unavailability of one has a visible effect, and each transaction touches 3 objects. The figure shows a sliding window average of the commit rate, normalized by the incoming rate. Before and after the failure event, this average fluctuates around the incoming rate, i.e. 1.0. While the OM is down, transactions that access the unavailable objects abort, resulting in a commit rate of about 60%. Note that the drop and rise are rather gentle due to the size of the averaging window — the OM crashes instantly and the replacement is ready to serve instantly (once it restored its state). A shorter window yields a steeper decline and incline, but a noisier output.

Since ACID-RAIN has no real locks, servers may be switched swiftly on suspicion of failure, shortening the interval required to restore a failed machine compared to competing systems with such locks. We experimented with this interval and, as seen in the illustration, shorter recovery delays linearly decrease the number of transactions that must wait.

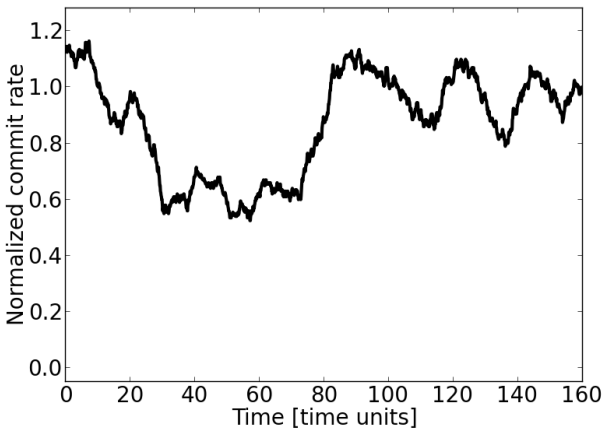


Figure 7.9: Effect of an OM crash and replacement. A moving average of the normalized commit rate is shown as a function of time. The OM of one out of 5 shards crashes at time 20, and is replaced (restoring from the log) at time 70. The average commit rate (over a sliding window) drops after the crash and rises once the replacement OM is in place.

We note that on TM crash, new transactions are simply routed to the available TMs, and efficiency is affected to the extent these form a bottleneck.

Chapter 8

Conclusion

This dissertation presented several works that address contemporary and future challenges in large scale distributed systems. We detail here relevant future directions that follow directly from this work.

Sensor Networks In Chapter 3, we have shown how to perform live monitoring of the average in a sensor networks with LiMoSense, and in Chapter 4 we have shown how to learn a one-shot clustering of the data. Combining the two, to perform live monitoring of a clustering, is a challenging task. Unlike with average monitoring, the merge operation of clusters is a unidirectional operation. Merged clusters cannot be divided to their original components. It is therefore, in general, impossible to remove old read-values that have propagated through the system as clusters. Division of clusters may be done heuristically, and its accuracy depends on the properties of the input. In certain scenarios, it may therefore be possible to split clusters and therefore achieve live monitoring. This should be studied with real-world data, to evaluate the accuracy with the division heuristic.

Cloud Storage In Chapter 6 we described the construction of a reliable regular register with multiple atomic key-value stores. However, production key-value store services typically offer only eventual consistency, where the state of the service may be arbitrary for short periods. On top of such service it may only be possible to build eventually consistent data structures. Defining the properties of eventual consistency and designing an algorithm that works with such base objects remains future work.

Some cloud providers offer APIs different than KVSs, such as queues and logs. Learning what robust data structures can be constructed with different base objects can prove useful for multi-cloud applications.

Finally, security risks call for byzantine fault tolerant algorithms that can withstand arbitrary, possibly malicious, behavior by clouds due to bugs or security breaches.

In Chapter 7 we presented the ACID-RAIN architecture and demonstrated its efficiency through simulation. In order to evaluate the ACID-RAIN architecture and compare it against other approaches to large scale atomic transaction systems, it has to be implemented and run on a datacenter. This would entail several challenges that arise due to its unique structure, e.g., implementing effective failure detectors, both accurate for the reliable logs, and inaccurate for the rest of the servers, together with a responsive load balancer.

Bibliography

[1] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk Paxos: Optimal resilience with Byzantine shared memory. *Dist. Comp.*, 18(5):387–408, 2006.

[2] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: a case for cloud storage diversity. In *SoCC'10*, 2010.

[3] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58:7:1–7:32, April 2011.

[4] M.K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[5] P. Almeida, C. Baquero, M. Farach-Colton, P. Jesus, and M. A. Mosteiro. Fault-tolerant aggregation: Flow updating meets mass distribution. In *OPODIS*, 2011.

[6] Amazon S3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html, retrieved Dec. 6, 2011.

[7] Amazon gets 'black eye' from cloud outage. http://www.computerworld.com/s/article/9216064/Amazon_gets_black_eye_from_cloud_outage, retrieved Dec. 6, 2011.

[8] Amazon Simple Storage Service. http://aws.amazon.com/s3/.

[9] Eric Anderson, Xiaozhou Li, Arif Merchant, Mehul A. Shah, Kevin Smathers, Joseph Tucek, Mustafa Uysal, and Jay J. Wylie. Efficient eventual consistency in Pahoehoe, an erasure-coded key-blob archive. In *DSN-DCCS'10*, 2010.

[10] G. Asada, M. Dong, T.S. Lin, F. Newberg, G. Pottie, W.J. Kaiser, and H.O. Marcy. Wireless integrated network sensors: Low power systems on a chip. In *ESSCIRC*, 1998.

[11] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

[12] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[13] Details of the december 28th, 2012 windows azure storage disruption in us south. http://blogs.msdn.com/b/windowsazure/archive/2013/01/16/details-of-the-december-28th-2012-windows-azure-storage-disruption-in-us-south.aspx, retrieved Jan. 23, 2013.

[14] J. Baker, C. Bond, J.C. Corbett, JJ Furman, A. Khorlin, J. Larson, J.M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR*, pages 223–234, 2011.

[15] Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolic, and Ido Zachevsky. Robust data sharing with key-value stores. In *42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2012.

[16] Philip A Bernstein, Colin W Reid, and Sudipto Das. Hyder-a transactional record manager for shared flash. In *Proc. of CIDR*, pages 9–20, 2011.

[17] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. In *EuroSys'11*, 2011.

[18] Yitzhak Birk, Idit Keidar, Liran Liss, and Assaf Schuster. Efficient dynamic aggregation. In *DISC*, 2006.

[19] Yitzhak Birk, Liran Liss, Assaf Schuster, and Ran Wolff. A local algorithm for ad hoc majority voting via charge fusion. In *DISC*, 2004.

[20] K.P. Birman. *Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services*. Springer, 2012.

[21] Stephen P. Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Gossip algorithms: design, analysis and applications. In *INFOCOM*, 2005.

[22] Stephen P. Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52(6):2508–2530, 2006.

[23] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011.

[24] Christian Cachin, Robert Haas, and Marko Vukolić. Dependable services in the intercloud: Storage primer. Research Report RZ 3783, IBM Research, October 2010.

[25] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[26] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, et al. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *SOSP'11*, 2011.

[27] Lásaro Camargos, Fernando Pedone, and Marcin Wieloch. Sprint: a middleware for high-performance transaction processing. *ACM SIGOPS Operating Systems Review*, 41(3):385–398, 2007.

[28] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[29] Gregory Chockler and Dahlia Malkhi. Active disk Paxos with infinitely many processes. *Dist. Comp.*, 18:73–84, 2005.

[30] B.F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[31] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[32] J.C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, JJ Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally-distributed database. *OSDI*, page 1, 2012.

[33] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[34] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic transactional data store in the cloud. *USENIX HotCloud*, 2, 2009.

[35] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. *ACM SOCC*, pages 163–174, 2010.

[36] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment*, 4(8):494–505, 2011.

[37] Souptik Datta, Chris Giannella, and Hillol Kargupta. K-means clustering over a large, dynamic network. In *SDM*, 2006.

[38] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP'07*, 2007.

[39] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *J. Royal Stat. Soc.*, 39(1), 1977.

[40] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley-Interscience, 2nd edition, 2000.

[41] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko Vukolic. Fast access to distributed atomic memory. *SIAM J. Comp.*, 39(8):3752–3783, 2010.

[42] A.J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 international conference on Management of data*, pages 301–312. ACM, 2011.

[43] Burkhard Englert and Alexander A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *ICDCS'00*, 2000.

[44] Patrick Th. Eugster, Rachid Guerraoui, Sidath B. Handurukande, Petr Kouznetsov, and Anne-Marie Kermarrec. Lightweight probabilistic broadcast. In *DSN*, 2001.

[45] I. Eyal, I. Keidar, and R. Rom. Distributed data clustering in sensor networks. *Distributed Computing*, pages 1–16, 2011.

[46] Ittay Eyal, Idit Keidar, and Raphael Rom. Distributed clustering for robust aggregation in large networks. In *HotDep*, 2009.

[47] Ittay Eyal, Idit Keidar, and Raphael Rom. Distributed data classification in sensor networks. In *PODC*, 2010.

[48] Ittay Eyal, Idit Keidar, and Raphael Rom. Limosense — live monitoring in dynamic sensor networks. In *7th International Symposium on Algorithms for Sensor Systems, Wireless Ad Hoc Networks and Autonomous Mobile Entities (ALGOSENSOR'11)*, 2011.

[49] Fabio Fagnani and Sandro Zampieri. Randomized consensus algorithms over large scale networks. *IEEE Journal on Selected Areas in Communications*, 26(4):634–649, 2008.

[50] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2), 1985.

[51] Eli Gafni and Leslie Lamport. Disk Paxos. *Dist. Comp.*, 16(1):1–20, 2003.

[52] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: an active distributed key-value store. In *OSDI'10*, 2010.

[53] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *J. Parallel Dist. Comp.*, 69(1):62–79, 2009.

[54] David K. Gifford. Weighted voting for replicated data. In *SOSP'79*, 1979.

[55] Seth Gilbert, Nancy Lynch, and Alexander Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. *Dist. Comp.*, 23:225–272, 2010.

[56] Gmail back soon for everyone. http://gmailblog.blogspot.com/2011/02/gmail-back-soon-for-everyone.html, retrieved Dec. 6, 2011.

[57] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *ACM SIGMOD Record*, volume 25, pages 173–182. ACM, 1996.

[58] Maxim Gurevich and Idit Keidar. Correctness of gossip-based membership under message loss. *SIAM J. Comput.*, 39(8):3830–3859, 2010.

[59] Maya Haridasan and Robbert van Renesse. Gossip-based distribution estimation in peer-to-peer networks. In *International Workshop on Peer-to-Peer Systems (IPTPS 08)*, February 2008.

[60] Joseph Heller. *Catch-22*. Simon & Schuster, 1961.

[61] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. and Sys.*, 13(1):124–149, 1991.

[62] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Prog. Lang. and Sys.*, 12:463–492, July 1990.

[63] P. Hunt, M. Konar, F.P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, volume 10, 2010.

[64] Navendu Jain, Prince Mahajan, Dmitry Kit, Praveen Yalagandula, Michael Dahlin, and Yin Zhang. Network imprecision: A new consistency metric for scalable monitoring. In *OSDI*, 2008.

[65] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 45:451–500, May 1998.

[66] jclouds — multi-cloud library. http://www.jclouds.org/.

[67] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)*, 23(3), 2005.

[68] P. Jesus, C. Baquero, and P.S. Almeida. Fault-tolerant aggregation for dynamic networks. In *SRDS*, 2010.

[69] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. Fault-tolerant aggregation by flow updating. In *DAIS*, 2009.

[70] F.P. Junqueira, B.C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 245–256. IEEE, 2011.

[71] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E.P.C. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[72] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *FOCS*, 2003.

[73] Wojtek Kowalczyk and Nikos A. Vlassis. Newscast em. In *NIPS*, 2004.

[74] Tim Kraska, Gene Pang, Michael J. Franklin, and Samuel Madden. Mdcc: Multi-data center consistency. *CoRR*, abs/1203.6049, 2012.

[75] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Op. Sys. Review*, 44:35–40, 2010.

[76] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):254–280, 1984.

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

[78] Leslie Lamport. On interprocess communication. *Dist. Comp.*, 1(2):77–101, 1986.

[79] 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. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[80] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[81] Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *FTCS'97*, 1997.

[82] J. B. Macqueen. Some methods of classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[83] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[84] D. Malkhi, M. Balakrishnan, J.D. Davis, V. Prabhakaran, and T. Wobber. From Paxos to CORFU: a flash-speed shared log. *ACM SIGOPS Operating Systems Review*, 46(1):47–51, 2012.

[85] M. Mark Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip based peer sampling. *ACM Transactions on Computer Systems*, 25(3), 2007.

[86] Mezeo: Cloud storage platform. http://www.mezeo.com/.

[87] Damon Mosk-Aoyama and Devavrat Shah. Computing separable functions via gossip. In *PODC*, 2006.

[88] Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *SenSys*, 2004.

[89] D. Ongaro, S.M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[90] C.H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[91] S. Patterson, A.J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *Proceedings of the VLDB Endowment*, 5(11):1459–1470, 2012.

[92] A. Pavlo, E.P.C. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel oltp systems. *Proceedings of the VLDB Endowment*, 5(2):85–96, 2011.

[93] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–15. USENIX Association, 2010.

[94] Rackspace hosting. http://www.rackspacecloud.com/cloud_hosting_products/files/.

[95] Jason K. Resch and James S. Plank. AONT-RS: Blending security and performance in dispersed storage systems. In *FAST'11*, 2011.

[96] Jan Sacha, Jeff Napper, Corina Stratan, and Guillaume Pierre. Reliable distribution estimation in decentralised environments. *Submitted for Publication*, 2009.

[97] D. J. Salmond. Mixture reduction algorithms for uncertain tracking. Technical report, RAE Farnborough (UK), January 1988.

[98] Cheng Shao, Evelyn Pierce, and Jennifer L. Welch. Multi-writer consistency conditions for shared memory objects. In *DISC'03*, pages 106–120, 2003.

[99] Y. Sovran, R. Power, M.K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM, 2011.

[100] A.S. Tanenbaum. *Computer networks*. Prentice Hall, 2003.

[101] The Apache Software Foundation. Apache hbase, 2012.

[102] R. van Renesse and F.B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, volume 6, 2004.

[103] Paul M. B. Vitányi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware (detailed abstract). In *FOCS'86*, 1986.

[104] Voldemort: A distributed database. http://project-voldemort.com/.

[105] B. Warneke, M. Last, B. Liebowitz, and K.S.J. Pister. Smart dust: communicating with a cubic-millimeter computer. *Computer*, 34(1), 2001.

[106] Yunqi Ye, Liangliang Xiao, I-Ling Yen, and Farokh Bastani. Secure, dependable, and high performance cloud storage. In *SRDS'10*, 2010.

לפיכך, אנו מציגים בפרק 3 את לימוסנס, אלגוריתם עמיד בפני תקלות המבצע ניטור רציף ברשת חיישנים דינאמית. זהו האלגוריתם החסין האסינכרוני הראשון לחישוב ממוצע המבצע ניטור רציף. האלגוריתם מחשב באופן רציף תמונה מדוייקת של המידע, אשר משתנה באופן דינאמי.

למרות שחישוב הממוצע מספיק במקרים רבים, יישומים שונים דורשים תמצית מורכבת יותר של המידע. *בבעיית החלוקה לצבירים (קלאסטרים) המבוזרת*, מספר גדול של חיישנים מחשבים חלוקה לצבירים של המידע הנדגם, כלומר מחלקים את המידע לאוסף צבירים ומתארים כל צביר באופן תמציתי.

בפרק 4 אנחנו מציגים אלגוריתם כללי (גנרי) אשר פותר את בעיית החלוקה לצבירים המבוזרת. ניתן לממש את האלגוריתם בטופולוגיות רשת שונות, ובאמצעות טיפוסים שונים של צבירים. למשל, האלגוריתם הכללי ניתן למימוש על מנת לבצע את החלוקה על בסיס מרחק של הדגימות מהצביר אליו הן שייכות. זו היא בעיית k הממוצעים (k-means) המפורסמת. ואולם, המרחק לא תמיד מפיק חלוקה טובה, לפיכך אנו מציגים גם מימוש המתאר את הצבירים כתערובת של פילוגים גאוסיאניים (Gaussian Mixture). במקרה זה, החלטות החלוקה נעשות בעזרת טכניקות מתחום המערכות הלומדות. בעזרת הדמיה (סימולציה) אנחנו מדגימים כי האלגוריתם עמיד בפני נפילות, מהיר ובר סילום. אנו מוכיחים כי כל מימוש של האלגוריתם הגנרי (העומד בכללים מסויימים) מתכנס בלי תלות בטופולוגיה, באסטרטגיית החלוקה ובייצוג של הצבירים.

עקביות באחסון בענן

בחלקה השני של העבודה נעסוק בשתי בעיות הנוגעות בעקביות (consistency) בעת אחסון מידע במחשוב ענן. ההתפתחות של טכנולוגיות מרכזי הנתונים (datacenters) מובילה לשימוש נרחב במרכזי נתונים לאחסון נפח אדיר של מידע במערכת מבוזרת. שירותי אינטרנט דוגמת רשתות חברתיות, מנועי חיפוש ואתרי קניות מאחסנים נפח מידע אדיר. חברות גדולות משתמשות לרוב במרכזי נתונים משלהן, אך גם חברות גדולות וגם קטנות עושות שימוש בשירותי אחסון ענן חיצוניים. בין אם הנתונים נשמרים במרכז פרטי ובין אם הם נשמרים בענן, למשתמשים ציפיות הולכות וגדלות מהשירות. הם מצפים לאמינות וזמינות גבוהים, לצד יעילות וביצועים מהירים.

ואולם, ספקים של שירותי אחסון בענן, גם הגדולים ביותר, לעתים כושלים. במקרה כזה, הם עלולים לאבד מידע, או לכל הפחות לא להיות זמינים לפרק זמן מוגבל. לפיכך, המשתמשים בשירותים אלה חייבים לשכפל את המידע שלהם אצל מספר ספקים שונים על מנת להשיג זמינות גבוהה ורציפה. ואולם, טכניקות שכפול קלאסיות [11] לא יישימות בתרחיש זה. הסיבה היא ששירותי האחסון מספקים בדרך כלל מנשק מפתח-ערך הכולל רק פונקציות לשמירה ושליפה של ערך על פי מפתח, ואינו מספיק כאבן בניין של הטכניקות הללו. בפרק 6 אנחנו מציגים אלגוריתם המספק מנשק של אחסון אמין באמצעות כמה שירותי אחסון מהעולם האמיתי. האלגוריתם יעיל וללא עיכובים (wait-free) ומספק מנשק של רגיסטר עם מספר קוראים ומספר כותבים, באמצעות מאגרי מפתח-ערך לא אמינים, כל זאת בסביבה אסינכרונית. מימשנו את האלגוריתם ובדקנו אותו באמצעות ספקים קיימים. אנו מדגימים בעזרת הדמיות המבוססות על המימוש הזה את התקורה הנמוכה של השכפול הן בנפח האחסון הנדרש והן במהירות.

לאחר מכן, בפרק 7, אנחנו מציגים את אסיד-ריין. זוהי ארכיטקטורה חדשנית למימוש יעיל של תנועות (טרנזקציות) אטומיות במאגרי נתונים מבוזרים. תנועות אטומיות רצויות מאוד במאגרי נתונים מבוזרים, אך קשה לממשן באופן יעיל. כמו ארכיטקטורות אחרות, מערכת אסיד-ריין בנויה ממספר שכבות, כאשר כל שכבה מספקת שירותים לשכבה שמעליה (הלקוח מתקשר עם השכבה העליונה). אסיד-ריין משתמשת ביומנים (לוגים) באופן חדשני המאפשר להגביל את הדרישה לאמינות אל שכבה יחידה של המערכת. אוסף גדול ובר סילום של צמתים בלתי תלויים מהווה שכבה חיצונית המועדת לנפילות. אלה מגובים על ידי אוסף בלתי תלוי של יומנים אמינים. אסיד-ריין מצמצמת התנגשויות בין תנועות בעזרת חיזוי, המאפשר לסדר את התנועות לפני ביצוע פעולות העלולות להוביל לכשלונן.

תקציר

טכנולוגיות ייצור ובניה של מערכות מחשב מתפתחות במהירות ומובילות לשגשוג של מערכות מבוזרות בסדרי גודל שהיו נחלת הדמיון אך לפני שנים מעטות. בעבודה זו נעסוק שני סוגים של מערכות כאלה – רשתות חיישנים ואחסון במחשוב ענן. אנחנו מתכננים מערכות חסינות ובנות סילום (סקאלאביליות, ניתנות להגדלה) בסביבות אלה, מוכיחים את נכונותן ומנתחים את התנהגותם על ידי הדמיה (סימולציה).

תכנון של מערכות מבוזרות גדולות צריך להתבצע על פי מספר עקרונות.

חסינות במערכת בעלת אלפי צמתים, מטבע הדברים צמתים מדי פעם מתקלקלים. בעקבות כך, מפעיל המערכת יירצה להוסיף צמתים חליפיים שצריכים להשתלב במערכת. כמו כן, במערכת גדולה יש לקחת בחשבון אפשרות של אובדן הודעות ונפילת קשרי תקשורת. על המערכת להיות עמידה בפני כל אלה.

אסינכרוניות צומת יחיד במערכת עלול לסבול מתקלה זמנית המאטה את פעולתו באופן ניכר. כמו כן, הודעות בין צמתים עלולות להתעכב אם עליהן לעבור בין נתבים. אלגוריתמים סינכרוניים מניחים כי משך העיכובים במערכת חסום מלמעלה. נקיטת הנחה שמרנית (חסם גבוה) משמעה שהמערכת מתקדמת באטיות, היות שצמתים חייבים להמתין זמן ממושך כדי לוודא שכל ההודעות שנשלחו (והתשובות עליהן) אכן הגיעו. במערכת גדולה אסור להניח חסם נמוך, כיוון שעיכובים ממושכים מתרחשים מדי פעם. לפיכך, אלגוריתמים סינכרוניים אינם יעילים במערכת גדולה.

סילומיות על מנת שמערכת תהיה בת סילום, יש להמנע לחלוטין משימוש ברכיב ריכוזי יחיד לפעולתה השוטפת. רכיב שכזה בהכרח יהפוך צוואר בקבוק של המערכת ברגע שהיא תגדל מעבר לגודל מסויים.

עבודה זו מתארת ארבעה אלגוריתמים בני סילום אשר פועלים על פי עקרונות אלה. כולם אסינכרוניים, חסינים לנפילה של צמתים ואובדן הודעות, ואינם תלויים בשום רכיב יחיד בפעולתם השוטפת.

הצרפה ברשתות חיישנים

בחלקה הראשון של העבודה נטפל בשתי בעיות בתחום *רשתות החיישנים*. על מנת לנטר שטחים נרחבים, אנו צפויים לראות בשנים הקרובות רשתות חיישנים המורכבות מאלפי חיישנים המנטרים פעילות סיסמית, לחות, טמרפטורה וכיוצא באלו. כל אחד מהחיישנים ברשתות אלו מורכב מהרכיב המודד, רכיב תקשורת אלחוטית להתקשרות עם צמתים קרובים, יחידת עיבוד ונפח אחסון מצומצם.

רשתות כאלה, הפרושות בשטח נרחב, מטבען לא מאפשרות לאסוף את כל המידע שנצבר למקום אחד. שידור כל המידע ברשת צורך משאבי סוללה ניכרים, והטופולוגיה הדינאמית של הרשת מקשה להפיץ את המידע. למרבה המזל, לעתים קרובות המידע הגולמי גם אינו נחוץ, כי אם הצרפה (אגרגציה) של המידע, אותה ניתן לחשב באופן מבוזר. לדוגמא, במדידת משקעים מעוניינים בכמות הגשם הכוללת באיזור נתון, ולא במדידות של כל אחד מהחיישנים; במדידת טמפרטורה, נתעניין לעתים קרובות בממוצע ולא בטמפרטורה הנמדדת בכל חיישן.

מספר עבודות [72, 21, 87, 83] טיפלו בגרסה החד-פעמית של בעיה זו. בגרסה זו, כל חיישן דוגם את הסביבה פעם אחת, ואז כל החיישנים מחשבים יחד את ממוצע הדגימות. ואולם, כדי לבצע ניטור רציף, יש לחשב באופן רציף את התמצית של המידע המשתנה באופן דינאמי. אם נפתור את הבעיה על ידי הרצה חוזרת ונשנית של האלגוריתם החד-פעמי, או שנבזבז משאבים (אם נריץ בתדירות גבוהה) או שנאבד דיוק (אם נריץ בתדירות נמוכה, או לא ניתן לריצות די זמן להתכנס).

המחקר נעשה בהנחייתם של פרופ' רפאל רום ופרופ' עדית קידר בפקולטה להנדסת חשמל.

אני מודה לטכניון ולמכון האסו פלטנר על התמיכה הכספית הנדיבה בהשתלמותי.

לסביי וסבתותיי, ליזה, יעקב, ברכה ואפרים

אלגוריתמים חסינים ובני סילום עבור אחסון בענן ועבור רשתות חיישנים

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת תואר דוקטור בפילוסופיה

איתי אייל

הוגש לסנט הטכניון – מכון טכנולוגיה לישראל
סיוָן התשע''ג - חיפה - מאי 2013

אלגוריתמים חסינים ובני סילום עבור אחסון בענן ועבור רשתות חיישנים

איתי אייל