2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00128

Samya: A Geo-Distributed Data System for High Contention Aggregate Data

Sujaya Maiyya, Ishtiyaque Ahmad, Divyakant Agrawal, Amr El Abbadi
University of California Santa Barbara
Santa Barbara, California
{*sujaya_maiyya, ishtiyaque, agrawal, amr*}*@cs.ucsb.edu*

***Abstract*—Geo-distributed databases are the state of the art tools for managing cloud-based data. But maintaining hot records in geo-distributed databases such as Google's Spanner can be expensive, as it synchronizes each update across a majority of replicas. Frequent synchronization poses an obstacle to achieve high throughput for contentious update-heavy workloads. While such synchronizations are inevitable for complex data types, simple data types such as aggregate data can benefit from reduced synchronizations. To this end, we propose an alternate data management system, *Samya*, to manage aggregate cloud resource usage data. Samya dis-aggregates available resources and stores fractions of these resources across geo-distributed sites. Dis-aggregation allows sites to serve client requests independently without synchronization for each update. Samya incorporates a learning mechanism to predict future resource demands. If the predicted demand is not satisfied locally, a synchronization protocol, *Avantan*, is executed to redistribute available resources in the system. Avantan is a novel fault-tolerant consensus protocol where sites agree on the global availability of resources prior to redistribution. Experiments conducted on Google Cloud Platform highlight that dis-aggregating data and reducing synchronizations allows Samya to commit 16x to 18x more transactions than state of the art cloud geo-distributed systems such as Spanner and CockroachDB.**

1. Introduction

Many small and mid-sized enterprises rely on large cloud providers, such as Amazon AWS, Google GCP, and Microsoft Azure, to provide backend infrastructure. While the cloud's *pay-per-use* strategy along with the elasticity to spawn new resources on demand has many benefits, it comes with a cost: an unexpected traffic spike can drastically increase the consumed resources, leaving the customer with a hefty bill.

To avoid such surcharges, cloud customers can set limits on the amount of resources they consume through a variety of *resource tracking services*. Clients can set limits on resources such as storage capacity, number of deployable VMs, and network bandwidth. Resource tracking services within a cloud provider actively maintain data on current resource usage; this data helps enforce the limits and bill the customer accurately for their usage. A resource can be consumed only if its current usage is below the preset limit of that resource – this translates to a *read-write* transaction at the resource tracking services.

Consider an example where a large cloud provider, *ultraCloud*, has a start-up *eCommerce.com* as a customer. The start-up comprises of many teams such as clothing, electronics, etc, as shown in Figure 1, and the teams consume resources as indicated in the leaf nodes. The resource limit is set by an admin of *eCommerce.com* and is applicable to all teams within the organization. Such a hierarchical structure is widely used by cloud providers to allocate resources, track usage, and accurately bill customers [13, 12, 14].

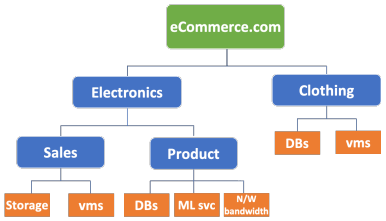


Figure 1: Hierarchical org structure of a cloud customer.

The cloud provider, *ultraCloud*, tracks the number of resources *eCommerce.com* can consume. For example, each VM creation results in a read-write transaction in *ultraCloud*'s resource tracking service to check if the **overall** VMs consumed exceed *eCommerce.com*'s threshold. Only after this transaction succeeds can the actual physical resource be allocated. Any update to an intermediary unit (team) must percolate to the root node, *eCommerce.com*, as the cumulative resource usage by all the teams in the hierarchy is tracked at the root level. Typical update rates for a single node in the hierarchy may be in the hundreds of transactions per second, but the aggregate load on the root for a moderately sized enterprise hierarchy may easily be in thousands of transactions, causing the root node's data to become a ***hotspot***.

In a cloud setting, data – including a tracking service's data – are stored on *multiple* servers across multiple data centers to ensure high availability and fault-tolerance, for example Google's Spanner [4] and Amazon Aurora [27].

Consider the design choices of a Spanner-like database: each data item is replicated across multiple sites, one of which acts as a leader. For each update, the leader replicates the change onto a set of replicas using consensus protocols such as Paxos [20]. While this is a good choice for high

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00128

availability, it aggravates the hot-spot problem in two ways: (1) *Sequential execution*: for hot-spots, where many transactions access the same data, conflicting transactions are processed by the leader sequentially; and (2) *High Latency*: each update is propagated to geographically distant sites, incurring high latency. Spanner commits a transaction with a mean latency of 17ms and a tail latency of 75ms [4]; hence for a single data item, *Spanner can commit on average 58.8 transactions per second (tps) and a tail throughput of 13.3 tps*. For a customer such as eCommerce.com (Figure 1), perhaps 60 tps is enough for an individual node, but for the aggregate root node with hundreds of teams in the hierarchy, this throughput value becomes problematic.

Our observation is that while geo-distributed databases are a good choice for supporting complex forms of data, they are not ideal for simple aggregate data types where the operations are mostly limited to additions or subtractions, such as maintaining resource usage data. Spanner-like solutions provide high scalability but fail to provide the high throughput necessary for hot-spot data. Based on this observation, our research objective is to *design an alternate system that manages simple data types and provides high throughput for update heavy workloads in a cloud setting.*

This objective has been addressed for traditional non-cloud databases in many works such as [23, 2, 19, 9]. Escrow transactions [23] introduced the notion of concurrent transactions updating different 'chunks' of aggregate data, albeit in a non-distributed database. Barbara et al. [2], Kumar et al. [19], and Golubchik et al. [9] introduced the idea of partitioning aggregate data onto multiple sites, allowing each site to independently update its portion of the data value (e.g., multiple sites independently selling airline tickets). The problem is made non-trivial by introducing a constraint while updating the distributed data (e.g., not selling more airline tickets than the available seats). The solution proposed in this paper is motivated by these works, adapted for the radically different settings of large scale geo-distributed cloud infrastructures.

If partitions of available resources are stored on different sites, the next logical question to ask is: how to distribute the available resources among these sites? Advancements in machine learning and deep learning as well as the abundance of cloud resource demand data collected by cloud providers can aid in answering this question. In fact if resource demand can be predicted and resources can be allocated to sites accordingly, most client requests can be served locally in a decentralized manner, without incurring expensive cross-datacenter communications.

In this paper, we propose ***Samya***[1] – a data system that stores and tracks resource usage data across geo-distributed sites. Samya avoids the high latency and low throughput of Spanner-like databases by allowing sites to serve client requests locally, without constant expensive synchronizations.

Overview: To serve client requests locally while still maintaining the global constraint, sites in Samya start with an initial allocation of available resources. We model the resource data as *tokens* (tokens of a specific resource are indistinguishable). A site can serve requests locally as long as it has locally available tokens; once it exhausts its local tokens or if it predicts an increase in resource demand that cannot be satisfied locally, it synchronizes with other sites to *redistribute* any unused tokens in the system. We propose a novel protocol –*Avantan*[2] – to redistribute spare tokens. Avantan is a fault-tolerant consensus protocol, in which, unlike Paxos, the value to agree upon is unknown at the start of the protocol.

Along with providing low latency, the dis-aggregation strategy of Samya increases its availability compared to Spanner-like databases. For a specific resource, Spanner becomes unavailable if a majority of the sites that store the resource information fail, whereas Samya is available as long as at least one site is available.

Other Applications of Samya: Although Samya is motivated and presented as a service that stores and tracks resource usage, it can be used as a data managing system to maintain *any* aggregate data in the cloud. Examples of applications consisting of aggregate data are: rate limiting services to manage quotas and policies; inventory management such as online shopping, car rentals, etc.; airline ticket booking; advertisement campaigns tracking; billing services; etc,. For ease of exposition, in this paper we focus on one application: maintaining resource usage data.

2. Related Work

The hotspot problem for aggregated data is an important practical problem studied extensively by the database community. Data partitioning is the most predominantly adopted solution for the hotspot problem, generally present in two main forms: (i) *Key partitioning* where data items are partitioned into different, non-overlapping sets based on their keys and the sets are stored across multiple sites. (ii) *Value partitioning* where the same data item, irrespective of its key, is partitioned into different *values* and these values are stored across multiple sites. Samya adopts the value partitioning approach to store fractions of an aggregate value (i.e., available tokens) across different sites.

The idea of value partitioning has been studied extensively, starting with the seminal paper by O'Neil [23]. In [23], O'Neill introduced escrow transactions where different transactions operate on different fractions of the same data, thus allowing concurrency; this was proposed for a non-distributed database. In [19], Kumar and Stonebreaker extended [23] to site escrows: sites serve transactions locally as long as they have non-zero escrow quantity. In [11], Harder extended the idea of escrows and introduced hierarchical escrows to reduce coordination to dynamically update escrows of multiple sites. In [18], Krishnakumar and Bernstein proposed Generalised Site Escrow to dynamically allocate parts of aggregate data (i.e., resources) to different sites using quorum locking and gossip messages.

The demarcation protocol [2] introduced by Barbara and Garica-Molina partitions an individual data value to be

1. Samya is the Sanskrit word for equilibrium or equality.

2. Avantan in Sanskrit means allocation.

stored on separate machines; the protocol explains how to maintain constraints on the data when the data is distributed. In [1], Alonso and El Abbadi extend the demarcation protocol to store the value partitions across more than two sites and formalize the theory of partitioned data. In [9], Golulbchik and Thomasian assume the incoming request pattern to follow Poisson distribution and allocate tokens to different sites based on this distribution.

In essence, the above discussed works aim to partition a data item based on its value, store the partitions on multiple sites, and update them concurrently, while maintaining a global constraint. These protocols are proposed for radically different environments where typically the sites are not geo-distributed, the networks are assumed reliable, and the results presented are typically simulations. Samya brings the basic idea – *dis-aggregate the aggregate data to increase concurrency* – from these works into the more modern context of cloud computing and geo-distributed data management systems.

A related approach that supports local operations without global synchronization is proposed by Shapiro et al. [25] as Conflict-free Replicated Data Types (CRDTs). CRDTs support conflict freedom by using eventually consistent replication. The eventual consistency guarantees and the semantics of the data types allows replicas to be updated locally and synchronized eventually. CRDTs, or rather systems that use CRDTs, differ from Samya in that the replicas of CRDTs do not dis-aggregate the value of a data item nor maintain a global and distributed constraint, which are important aspects of Samya. The replicas of a data item in CRDT systems have identical values, without a notion of re-balancing the values stored in each replica, as performed in Samya. Another major difference between the two is data in CRDT systems typically need to be commutative whereas data in Samya can be non-commutative.

With the rise of the cloud paradigm, many new database designs opt for geo-distribution to provide high availability [4, 6, 27, 26]. The data in these systems are key partitioned and replicated across geo-distributed sites. Google Spanner [4], Amazon Aurora [27], and CockroachDB [26] all use replication protocols such as Paxos [20] or Raft [24] to consistently replicate each update to a quorum (typically majority). While Amazon's Dynamo [6] chooses eventual consistency and is hence less stringent on replicating each update, it may suffer from inconsistent data.

In general, these approaches differ from Samya in that they employ key partitioning and aim to efficiently execute distributed transactions across these partitions, which are often also replicated. First, due to replicating each update to a quorum of geo-distributed sites, these systems are prone to hot-spot problems for update heavy and contentious workloads. Second, the design mantra common across these works is to build a general data management system that can store varied and complex forms of data. While the general approach has many benefits, it fails to take advantage of application specific data forms (such as aggregate data) to optimize performance. This forces applications such as cloud resource tracking services to inevitably design their own data managing systems. Samya is designed to take advantage of aggregate data by providing high performance support for hot-spots without compromising availability.

3. Samya Architecture

3.1. System Model

This section discusses the system model of Samya. Samya consists of *sites* and *application managers*.

Sites: To enhance the performance and for high availability, the aggregate data is dis-aggregated into different partitions and the partitions are stored across multiple sites, typically geo-distributed. Sites in Samya act as *data shards* that store fractions of dis-aggregated resource availability and usage data. For simplicity, we assume that all sites store information about all resources and in most of the paper the focus is on managing a single resource. Changing this design choice to allow only some sites to store information of specific resources is fairly straightforward; a run-time library can provide lookup and directory services to identify the sites that maintain a specific resource data.

Application Managers: These are stateless processes that relay the messages between a client and the sites. App managers mask the network topology and individual site availability from external clients. As the sites and the clients are geo-distributed, multiple geo-distributed app manager processes exist to reduce the communication latencies between clients and sites. Being stateless, app managers can easily scale on demand depending on the request load.

Samya assumes an underlying asynchronous communication network where messages can be delayed, dropped, or reordered. The sites and the application managers can fail by crashing but do not exhibit malicious behavior. Unless they crash, the sites and application managers execute the designated protocol correctly. Samya further assumes that a site, which stores the data, *does not crash indefinitely*; when a crashed site recovers, it reconstructs its previous state (typically stored on stable storage). If an application manager crashes, since they are stateless, a new process can be spawned easily and plugged into the system.

3.2. Data Model

Abstractly, we term each resource stored in Samya as an *entity*. Multiple instances of an entity are viewed as a set of indistinguishable *tokens*. The clients (i.e., cloud customers) acquire or release these tokens and Samya tracks client actions to maintain up-to-date resource usage and resource availability information for each entity. The application owner (e.g., admin of an enterprise) configures a preset maximum $\mathcal{M}_e$ – the limit on available tokens for entity e – and the application users can acquire or release specific quantities of tokens for the entity.

Samya maintains the following system level constraint: at no point does the system allow the clients to collectively acquire more than $\mathcal{M}_e$ tokens for an entity e.

$$0 \leq total\_acquired\_tokens \leq \mathcal{M}_e \tag{1}$$

The state of an entity e, as maintained by each site S in the system, refers to specific details as presented in Table 1a: id is a unique identity to identify the type of entity e (or resource, such as VM or storage); $TokensLeft$ indicates the number of tokens of entity e available at site S; $TokensWanted$ indicates the number of tokens of entity e that site S needs during a redistribution.

Transactions: Clients perform two types of transactions:

- *acquireTokens(e, n)*: A client asks for n tokens of entity e, where n is a positive integer.
- *releaseTokens(e, m)*: A client returns m tokens of entity e, where m is a positive integer. These tokens can later be acquired by other clients. An individual client never returns more tokens than what it has acquired.

4. Samya

In this section we discuss how Samya efficiently manages and tracks resource usage. Samya is a highly available distributed data management system proposed as an alternative to manage resource data in geo-distributed databases. If a client consumes any resource such as creating additional VMs, traditional geo-replicated databases update all the replicas to reflect the resource usage. Samya, on the other hand, chooses a single site to update the resource usage data, thus avoiding the expensive cross data-center communications for each update. To cope with varying resource demands at different sites, Samya relies on learning based predictions and dynamic reallocation of resources.

4.1. Overview

In this section, we provide an overview of Samya's request serving approach.

4.1.1. Components of a Site in Samya. Each site in Samya consists of 4 components as shown in Figure 2.

- **Request Handling Module**: This module communicates with app managers and serves client requests locally. This module also triggers redistributions.
- **Prediction Module**: This is a learning module, trained on existing resource demand data, that predicts future resource demands in terms of number of tokens.
- **Protocol Module**: If the Request Handling module triggers a redistribution, this module executes a multi-round fault-tolerant protocol that collects token information from other sites for redistribution.
- **Redistribution Module**: Once the redistribution protocol completes, based on the responses from other sites, this module re-allocates the spare tokens.

Each of these modules is pluggable and can be easily replaced with an upgraded version, if and when needed.

4.1.2. Life-cycle of a client request. A step-wise overview of how sites in Samya serve client requests is presented in Figure 2:

1) Client request: A client generates and sends either an *acquireTokens(e, n)* or *releaseTokens(e, m)* request, where e is the entity id, and n, m are number of tokens. This request reaches the closest app manager to the client (this can be achieved using a load balancer).

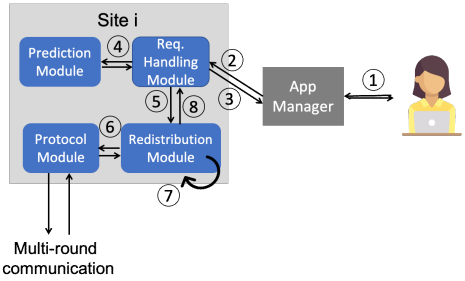


Figure 2: Life-cycle of a client request.

2) App Manager: Typically, an app manager relays the client request to the closest site. But if the closest site has failed or is overloaded, an app manager may relay the client request to another site. As a result, a single client's *acquireTokens* request may be sent to a site S_k whereas a *releaseTokens* request may be sent to a different site S_j. This is acceptable because sites in Samya only store the resource usage data; Samya is not responsible for the actual physical resource allocation, which is the function of higher level applications.

3) Site serving request: A site S that receives a client request attempts to serve the request locally; if successful, it updates its local token state based on the type of request:

$$TokensLeft_S = TokensLeft_S - n \quad (2)$$

if the client acquired n tokens; or

$$TokensLeft_S = TokensLeft_S + m \quad (3)$$

if the client released m tokens. The site then responds to the client, which is relayed via an app manager.

4) Demand prediction: After serving a client request, the site predicts the future demand (e.g., next 5 minutes).

5) Trigger redistribution: If the predicted value indicates a decrease in demand, the site simply continues to serve more requests. However, if the predicted value indicates an increase in demand that cannot be satisfied locally, the site triggers a redistribution.

6) Execute protocol: If the site triggers a redistribution, it communicates with other sites to collectively execute a fault-tolerant protocol to share with each other the state of entity *e*, which includes the information shown in Table 1a.

7) Reallocate tokens: Based on the shared information, each site independently reallocates the overall spare tokens using a deterministic reallocation procedure.

8) Update tokens state: Depending on the outcome of the reallocation, the site that triggered the redistribution may acquire more tokens, upon which it updates its state of entity e and serves any pending or future requests.

In the following sections, we elaborate on: i). when is a redistribution triggered? ii). how is the redistribution protocol executed? and iii). once the protocol terminates, how are the spare tokens reallocated? Table 1b defines the variables that will be used in the following sections.

4.2. Triggering Redistribution

The cloud computing literature has many works that highlight the predictability of resource demands in the cloud,

item	*description*
id	identity of resource type
$TokensLeft_S$	Tokens left at site S
$TokensWanted_S$	Tokens wanted by site S

(a) State variables of an entity e.

Symbol	Meaning
N	Number of sites
TL_t	Tokens Left at t^{th} redistribution
TW_t	Tokens Wanted at t^{th} redistribution
$\mathcal{S}_t$	Total spare tokens in t^{th} redistribution
$\mathcal{R}_t$	Set of sites in t^{th} redistribution
$\mathcal{L}_t$	List of state variables of the sites in $\mathcal{R}_t$

(b) Symbols used in explaining the redistribution strategy.

Variable	init values
BallotNum	initially $< 0, s >$
InitVal	state of entity e (Table 1a)
AcceptVal	initially $\perp$ (Null)
AcceptNum	initially $< 0, s >$
Decision	initially False

(c) Avantan variables at site s during each protocol execution.

TABLE 1

e.g., [10, 5, 22, 7, 17]; as well as discuss various techniques to predict resource demand. The common underlying idea is to collect a large amount of actual demand data, analyze this data to uncover any periodicity or patterns, and develop mathematical models that can learn from the past data to predict future demands.

Samya adopts a similar approach where application-specific historical resource demand data is collected to train a learning model. Once this model is trained and tuned to predict future demands, it is used as the Prediction Module (Figure 2). The Prediction Module is a pluggable module wherein the application developers using Samya are free to choose the best prediction technique suitable for their workload. This module can be replaced even after deployment, if a better learning approach is found or if the application workload changes. We discuss the prediction methods used in evaluating Samya in Section 5.

An *epoch* is defined as the look-ahead time duration used during prediction. This dictates how far ahead in the future to predict resource demand (e.g., 5 or 10 minutes) depending on the workload pattern. Samya supports two ways of triggering a redistribution.

• *Proactive redistribution*: After a site serves an *acquireTokens(e,n)* request, in a background thread, it predicts resource demand for the next epoch. If demand is decreasing or if the increase is below its current tokens availability, the site continues to serve client requests; otherwise, the site triggers a redistribution by updating its state of entity e:

$$TokensWanted = PredictedValue - TokensLeft \quad (4)$$

• *Reactive redistribution*: Since prediction techniques are rarely 100% accurate, Samya allows for *reactive* redistributions. When a site receives a client request that cannot be served locally due to insufficient locally available tokens, the site triggers a reactive redistribution by updating its state of entity e:

$$TokensWanted = m \quad (5)$$

4.3. Executing Redistribution Protocol

Once a site decides to trigger redistribution, it executes *Avantan*: a novel fault-tolerant consensus protocol used to redistribute available resource tokens. This section presents two different versions of *Avantan* differing primarily in their failure assumptions and failure recoveries. Sites execute multiple instances of Avantan either sequentially or concurrently; a single execution instance is presented in this section. For each instance, Avantan aims to reach agreement on the list of sites participating in that instance. The protocol is designed to tolerate arbitrary crashes, message losses, and network partitions, while making a best effort to provide liveness. The two versions of Avantan are:

• **Avantan[$\frac{n+1}{2}$]**: Requires a majority ($\frac{N}{2}+1$) of sites to be alive and communicating during protocol execution. This version is suitable when individual network links are highly unreliable (prone to message drops) and servers crash frequently but network partitions are infrequent. In this version *all* sites execute one redistribution after another.

Avantan[$*$]: No requirements on majority of sites being alive to execute the protocol; it tolerates network partitions of arbitrary sizes and allows different partitions to execute redistribution concurrently. But this version is sensitive to message losses during the execution of the protocol.

The two versions also differ in their failure recovery mechanism, which will be discussed later. In developing the protocol, we follow the abstractions defined in the Consensus and Commitment (C&C) framework [21]. Avantan abstractly consists of the following phases: the first phase executes *Leader Election* and *Value Construction*, the second phase makes the value *Fault-Tolerant*, and finally, the third asynchronous phase distributes the *Decision*.

We explain the two versions of Avantan with respect to redistributing the tokens of a single entity e; the protocol can be easily extended to include multiple entities. During protocol execution, sites maintain the variables in Table 1c, which mainly correspond to the standard variables used in Paxos. $BallotNum$ is a tuple of the form $<num, id>$ where *num* is a local, monotonically increasing integer and *id* is site id. Ballot number ensures the total ordering of different redistributions. *InitVal* is the current state of entity e (as defined in Table 1a) at site s when the redistribution starts. *AcceptVal* is the list of state values of the sites participating in the redistribution and *AcceptNum* is the ballot number at which a site updates its *AcceptVal*. Finally, a boolean $Decision$ indicates if the sites reached agreement on *AcceptVal* at ballot $BallotNum$. Unlike in Paxos, *AcceptVal* is a list of values and not a single value.

The redistribution protocol is initiated by a site S either for proactive or reactive reasons. Note that once a site starts participating in the redistribution protocol, it queues all the *acquireTokens* and *releaseTokens* requests from clients until the protocol terminates.

4.3.1. Avantan[$\frac{n+1}{2}$]. The protocol consists of 3 rounds (5 phases) as described in Algorithm 1:

Algorithm 1 Avantan[$\frac{n+1}{2}$] redistribution protocol.

Let $state_{t.i}$ be the state of entity e at site with id i during t^{th} execution of Avantan.

1: **procedure** ELECTION-GETVALUE()

2: BallotNum ← (BallotNum.num+1, selfId)
3: InitVal ← $currState$ /* With an updated $TokensWanted$ */
4: Send *Election-GetValue(BallotNum)* to all

5: **procedure** ELECTIONOK-VALUE()

6: upon receiving *Election-GetValue(bal)* from S
7: **if** bal > BallotNum **then**
8: BallotNum ← bal
9: predictedVal ← PredictForNextEpoch()
10: **if** predictedVal > currState.TokensLeft **then**
11: currState.TokensWanted ← predictedVal - currState.TokensLeft
12: InitVal ← $currState$
13: Send *ElectionOk-Value(BallotNum, InitVal, AcceptVal, AcceptNum, Decision)* to S

14: **procedure** ACCEPT-VALUE()

15: **if** received *ElectionOk-Value(bal, initV, acceptV, acceptN, dec)* from majority **then**
16: **if** at least ONE response with *dec*=True **then**
17: AcceptVal ← *acceptV* of that response
18: Decision ← True
19: **else if** at least one response with *acceptV*$\neq \perp$ /* *dec* is True for none. */ **then**
20: AcceptVal ← *acceptV* of the highest *acceptN*
21: **else**
22: AcceptVal ← (InitVal || all received $initV$s)
23: AcceptNum ← BallotNum
24: Send *Accept-Value(BallotNum, AcceptVal, Decision)* to all

25: **procedure** ACCEPT-OK()

26: upon receiving *Accept-Value(bal, acceptV, acceptN, dec)* from S
27: **if** bal $\geq$ BallotNum **then**
28: AcceptVal ← acceptV
29: AcceptNum ← bal
30: Decision ← dec
31: Send *Accept-ok(BallotNum)* to S

32: **procedure** DECISION()

33: **if** received *Accept-ok(bal)* from majority **then**
34: Decision ← True
35: Send *Decision(BallotNum, Decision)* to all

• ***Election-GetValue***: In the first phase site S attempts to become the leader as well as collect the state values from other sites. Site S increments its ballot number (line 2) and sets its $InitVal$ to its current local state of entity e, i.e., *TokensLeft*, and *TokensWanted*. Site S then sends Election-GetValue(BallotNum) messages to all sites.

• ***ElectionOk-Value***: As shown in lines 6-13, upon receiving the *Election-GetValue* message, a site C (termed as cohort to distinguish from the leader) checks if the received ballot number is greater than its current ballot number. If yes, it updates its ballot number, and predicts the next epoch's demand (line 9). C also sets its $TokensWanted$ field (line 11) only if the predicted value is greater than current tokens left, as C cannot satisfy its increasing demand. C then sets its $InitVal$ to the updated state and sends ElectionOk-value(BallotNum, InitVal, AcceptVal, AcceptNum, Decision) to leader S. The *AcceptVal, AcceptNum*, and *Decision* variables are used in failure recovery; in a failure-free execution, these variables are set to the initial values as defined in Table 1c.

• ***Accept-Value***: The leader S waits until it receives ElectionOk-Value messages from a *majority* of the sites including itself, denoted as $\mathcal{R}_t$. In a failure-free execution (failure recovery explained later), S sets $AcceptVal$ to the concatenated *InitVal*s received in the ElectionOk-Value responses (line 22), and sets $AcceptNum$ to its current ballot number. S then sends Accept-Value(BallotNum, AcceptVal, Decision) to all sites.

• ***Accept-ok***: Upon receiving the Accept-Value message from the leader, indicated in lines 26-31, a cohort C checks whether the received ballot number is at least as high as its current ballot number. If yes, it updates the $AcceptVal$, $AcceptNum$, and $Decision$ variables and sends an Accept-ok(BallotNum) message to the leader.

• ***Decision***: The leader waits for Accept-ok messages from at least a majority of sites (including self), then sets its $Decision$ variable to True, and sends Decision(BallotNum, Decision) messages to all. The protocol terminates for the leader when it sends the $Decision$ messages; the protocol terminates for a cohort once it receives the Decision message. The sites then reallocate the tokens using the information in $AcceptVal$ and respond to any pending client requests that were queued. The sites also reset the variables in Table 1c (except $BallotNum$) once the protocol successfully terminates.

Failure Recovery: If the leader fails (crashes or partitions), the other sites must recover in order to continue serving the clients. The recovery follows the same steps as a failure-free execution: if a site S' times-out waiting for the leader's message, S' attempts to become the new leader and terminate the protocol by sending Election-GetValue messages to all the sites. As shown in Procedure Accept-Value (lines 15-24), S' waits for a majority of ElectionOk-Value messages and checks if at least one of the messages has $Decision$ as True: this implies the previous leader had terminated the protocol and had sent at least one Decision

message before failing; so S' chooses the $AcceptVal$ received in this message (lines 16-18).

If none of the received messages has $Decision$ as True but at least one message has a non-empty $AcceptVal$, this implies the previous leader had constructed the $AcceptVal$ and sent Accept-Value to at least one site before failing; hence the new leader S' chooses this value as $AcceptVal$ (lines 19-20). If multiple sites respond with differing $AcceptVal$s, the new leader chooses the $AcceptVal$ corresponding to the highest $AcceptNum$. Any other case implies the previous leader had either failed to construct $AcceptVal$ or to store it on a majority before failing, and hence, S' is free to construct $AcceptVal$ based on the received $InitVal$s (line 22) (this is also the failure-free behavior). The next steps of fault-tolerantly storing the chosen value and sending the decision are the same as in failure-free executions.

Fault Tolerance: As stated in the FLP impossibility result [8], no consensus protocol can guarantee termination even with a single site failure. Following the impossibility result, Avantan[$\frac{n+1}{2}$] can block if a majority of the sites fail or are unreachable, similar to Paxos.

In spite of the blocking behaviour of Avantan[$\frac{n+1}{2}$], Samya's availability is higher than a system that executes Paxos for each transaction (e.g., Spanner). This is because Avantan[$\frac{n+1}{2}$] does not block if a majority of the sites have failed in the *first* phase of the protocol. To provide liveness, we use timeouts: if a site that wants to be a leader sends out Election-GetValue message but does not receive enough ElectionOk-Value messages within the timeout period, the site terminates the redistribution and continues to serve any client requests that can be served locally. This is acceptable since the leader failed to construct any value before aborting the redistribution.

However, if the leader successfully constructed a value and sent Accept-Value messages to all sites but failed to make the value fault-tolerant, i.e., it did not receive a majority of Accept-Ok messages, then that site and the other live sites are blocked until a majority recover.

Theorem 1*: No two distinct values are both chosen for a given instance of Avantan[$\frac{n+1}{2}$].*

Proof: Proof presented in the extended technical report [16].

4.3.2. Avantan[$*$]. Avantan[$\frac{n+1}{2}$], similar to Paxos [20] and other other consensus algorithms, is restrictive as it requires a majority of sites for redistribution to succeed. However, the tokens required by a site S, indicated in the $TokensWanted$ field, might be satisfied by fewer than a majority of sites. The token redistribution logic imposes no requirements on the minimum number of sites. Hence, we propose an alternative consensus protocol, Avantan[$*$], that allows *any subset of sites* to participate and ensures that all participating sites agree on the same value. We modify Avantan[$\frac{n+1}{2}$] to accommodate these new requirements.

The failure free execution code of Avantan[$*$] is the same as Algorithm 1 but with 3 major changes:

(i). The leader S, rather than waiting for a majority of responses after sending Election-GetValue messages, waits until it receives enough ElectionOk-Value messages (with $TokensLeft$ field set) such that S's token requirements can be satisfied. After a predefined amount of time, if S does not receive enough responses, it aborts the redistribution and notifies other sites and the clients. All the sites whose $InitVal$s were collected form the set $\mathcal{R}_t$ – the set of sites participating in the t^{th} redistribution. In all subsequent rounds, S communicates only with the sites in the $\mathcal{R}_t$, while notifying the other sites to discard this redistribution.

(ii). If a cohort responds with ElectionOk-Value message to one leader, it rejects all other Election-GetValue messages from concurrent leaders (even if they have higher ballot) until the former instance of Avantan[$*$] is complete. This ensures that a site participates in one instance of redistribution at a time.

(iii). Rather than waiting for any majority of sites to respond with Accept-ok messages (as shown in line 33), the leader S waits to receive Accept-ok from *ALL* the sites in $\mathcal{R}_t$ before it sends out the decision.

Different sets of sites can execute parallel redistributions but an individual site participates in one redistribution at a time.

Failure Recovery: Since Avantan[$*$] does not require a majority quorum to proceed, its failure recovery differs from that of Avantan[$\frac{n+1}{2}$]. The leader S or other participants can fail at any point during the execution of the protocol. Sites that did not even receive the Election-GetValue message are free to participate in other redistributions. If the leader fails (crash or network partition), sites that participated in the redistribution, must be able to recover.

A cohort C that participated in the redistribution detects leader failure using time-outs. Upon timeout, C checks the progress of the protocol execution. using the variables defined in Table 1c. If site C's $Decision$ variable is True, this implies the protocol had terminated and so C reallocates the tokens. If the $Decision$ is not true, C decides its next action based on the value of $AcceptVal$:

i). If $AcceptVal = \perp$: This implies C did not receive AcceptVal from the leader S before S failed; thus, C is free to abort this redistribution because the previous leader could not have proceeded to the *Decision* phase without the Accept-ok from C.

ii). If $AcceptVal \neq \perp$: This implies the leader had chosen a value but may not have decided on it if it had failed to receive all Accept-oks from $\mathcal{R}_t$ before failing. C then contacts all sites in $\mathcal{R}_t$ and waits for their responses (note that C knows all the sites in $\mathcal{R}_t$ based on the list of $InitVal$s in $AcceptVal$). If any site responds with $Decision$ as True, this implies the previous leader fault-tolerantly stored the value but failed before sending Decision to all. Site C sends the Decision message to all sites in $\mathcal{R}_t$.

Otherwise, if any site responds with $AcceptVal = \perp$, this is same as case (i), and C can safely abort the redistribution (and perhaps notify other sites in $\mathcal{R}_t$). If all sites in $\mathcal{R}_t$, except the previous leader S, respond with identical $AcceptVal$, this implies S successfully stored the value on all sites in $\mathcal{R}_t$ but failed before sending any Decision message. Hence, site C decides on that value, sets $Decision$ to True, and sends Decision messages. Finally, if C cannot

communicate with all the other blocked sites in $\mathcal{R}_t$, it is blocked until the communication is fixed.

Fault tolerance: Similar to Avantan[$\frac{n+1}{2}$], failures during protocol execution can cause sites in $\mathcal{R}_t$ participating in that execution of Avantan[$*$] to be blocked. But since Avantan[$*$] allows fewer number of sites to participate in a redistribution compared to Avantan[$\frac{n+1}{2}$], the set of sites not participating in the t^{th} instance of Avantan[$*$] are free to serve client requests or execute another redistribution instance. The experiments in Section 5 analyze and contrast the fault tolerance of the two versions of Avantan in practical settings.

Theorem 2*: No two distinct values are both chosen by the set of sites participating in a given instance of Avantan[$*$].*
Proof: Proof presented in the extended technical report [16].

While Avantan seems similar to Paxos, they differ in two major ways: (i) Paxos aims to reach agreement on a single, client provided value whereas Avantan collects partial values from each site and aims to reach agreement on the aggregated values, and (ii) the redistribution correctness condition (Equation 1) does not require a majority – a fact that is exploited in designing Avantan[$*$]– which is stringent requirement of Paxos.

4.4. Reallocating Tokens

Once the execution of Avantan terminates, the sites execute a deterministic procedure, discussed in this section, to reallocate the tokens. A successful execution of either versions of Avantan ensures agreement on the $AcceptVal$, which is a list of $InitVal$s, i.e.,:

$$\mathcal{L}_t =< e, TL_t, TW_t > \forall\, i \,\epsilon\, \mathcal{R}_t \tag{6}$$

The reallocation logic defined in Algorithm 2 takes $\mathcal{L}_t$ as input and reallocates the spare tokens among the set of sites in $\mathcal{R}_t$. *The redistribution algorithm ensures the constraint in Equation 1* that at no point does the token allocation count across all sites exceed the maximum limit $\mathcal{M}_e$ for a given entity e. For ease of exposition, we again focus of reallocating the tokens for a single entity e and use the variables defined in Table 1b to explain Algorithm 2.

Redistributing tokens: The RedistributeTokens procedure takes $\mathcal{L}_t$ as input. The spare tokens and the total tokens wanted (sum of tokens in the $TokensWanted$ field of each site) across all sites in $\mathcal{R}_t$ are computed as shown in lines 4-6. If the spare tokens are more than the total tokens wanted, all pending client requests can be satisfied, and the AllocateTokens procedure is called.

Rejecting requests: If the tokens wanted are more than the spare tokens available, some requests must be rejected as defined in the RejectSomeRequests procedure of Algorithm 2. We adopt a greedy approach to *maximise overall token usage* rather than maximise the number of requests satisfied. This is achieved by first sorting the list $\mathcal{L}_t$ in ascending order of tokens wanted (line 11); we choose the ascending order so that the algorithm can reject requests with least tokens wanted first. Requests are rejected by setting tokens wanted to 0 (line 13) and increasing the spare quantity, $\mathcal{S}_t$, (line 14) until the number of spare tokens exceeds the total tokens wanted (lines 15-16).

Algorithm 2 Procedures to re-allocate spare tokens after a successful redistribution

1: **procedure** REDISTRIBUTETOKENS($\mathcal{L}_t$)
2: $\mathcal{S}_t \leftarrow 0$ /* Spare tokens */
3: $TotalTW \leftarrow 0$ /* Total tokens wanted*/
4: **for** i in $\mathcal{R}_t$ **do**
5: $TotalTW \leftarrow TotalTW + \mathcal{L}_t[i].TW_t$
6: $\mathcal{S}_t \leftarrow \mathcal{S}_t + \mathcal{L}_t[i].TL_t$
7: **if** $TotalTW > \mathcal{S}_t$ **then**
8: $\mathcal{L}_t, \mathcal{S}_t \leftarrow$ RejectSomeRequests($\mathcal{L}_t, \mathcal{S}_t$)
9: AllocateTokens($\mathcal{L}_t, \mathcal{S}_t$)
10: **procedure** REJECTSOMEREQUESTS($\mathcal{L}_t, \mathcal{S}_t$)
11: $sorted\mathcal{L}_t \leftarrow \mathcal{L}_t$ sorted in ascending order of TW_t
12: **for** i in $sorted\mathcal{L}_t$ **do**
13: $sorted\mathcal{L}_t[i].TW_t \leftarrow 0$
14: $\mathcal{S}_t \leftarrow \mathcal{S}_t + sorted\mathcal{L}_t[i].TL_t$
15: **if** $TotalTW \leq \mathcal{S}_t$ **then**
16: break
17: **return** $sorted\mathcal{L}_t, \mathcal{S}_t$
18: **procedure** ALLOCATETOKENS($\mathcal{L}_t, \mathcal{S}_t$)
19: **for** i in $\mathcal{R}_t$ **do**
20: $\mathcal{L}_t[i].TokensGranted \leftarrow \mathcal{L}_t[i].TW_t$
21: $\mathcal{S}_t \leftarrow \mathcal{S}_t - \mathcal{L}_t[i].TW_t$
22: **for** i in $\mathcal{R}_t$ **do**
23: $\mathcal{L}_t[i].TokensGranted \leftarrow \mathcal{L}_t[i].TokensGranted + \frac{\mathcal{S}_t}{len(\mathcal{R}_t)}$

Allocating spare tokens: Finally, AllocateTokens (line 18) is called with the updated list $\mathcal{L}_t$ and spare tokens $\mathcal{S}_t$. At this point, the redistribution satisfies all requests with non-zero tokens wanted (as the requests that cannot be satisfied are already rejected). A tokens request is granted as shown in line 20 and for each granted request, the spare quantity is updated (line 21) After satisfying requests, if any more tokens are left, they are equally distributed among all the participating sites (line 23).
The redistribution procedure is pluggable; an application can plug in any other algorithm that better suits their need.

5. Experimental Evaluation

This section discusses the experimental evaluation of Samya, specifically the performance of two versions of Samya that uses Avantan[$\frac{n+1}{2}$] and Avantan[$*$] to handle any redistributions during the experiments. Samya's performance is compared with two baselines we implemented in Go and one open-sourced database:

i) ***MultiPaxSys***: A Spanner-like geo-distributed database that executes multi-Paxos [3] for each transaction.

ii) ***Demarcation/Escrow***: A value-partitioned system that captures the underlying mechanisms proposed in [2,

	Random Walk	ARIMA	LSTM
MAE (no. of tokens)	1212.19	609.13	259.21

(a) Mean Absolute Error (MAE) - in units of number of tokens - of resource demand prediction for three different prediction models.

percentiles	Samya w/ Av.$[\frac{n+1}{2}]$	Samya w/ Av.$[*]$	Dem./ Escrow	MultiPaxSys	CockroachDB
90^{th}	1.40 ms	2.9 ms	3.5 ms	126.8 ms	158.7 ms
95^{th}	10.2 ms	37.3 ms	59.6 ms	172.7 ms	184.2 ms
99^{th}	65.1 ms	97.3 ms	213.9 ms	276.3 ms	351.4 ms

(b) Different latency percentiles of Samya and baseline systems.

TABLE 2

19, 1, 18]. Specifically, Demarcation/Escrow extends the Demarcation protocol proposed by Barbara et al. [2] to more than 2 sites, similar to [1] by Alonso et al., and integrates the notion of site escrows used in [19] by Kumar et al. All sites start with an equal 'escrow' of an entity e (maximum limit, $\mathcal{M}_e$, divided equally among all sites), and the sites serve requests locally until they exhaust the spare escrow locally, upon which a site i borrows escrows from one or more sites. A stringent requirement of this baseline, inherited from [2] and [19], is it requires the network to be reliable; a message loss may lead to blocking.

(iii). ***CockroachDB*** (v20.2.3): A state-of-the-art open sourced geo-distributed database [26] that uses Raft[24] to replicate any changes to the data.

In evaluating Samya, the experiments focus on two performance aspects: *commit latency* – time taken to commit a transaction measured by the client; and *throughput* – the number of transactions successfully committed per second, i.e, only granted *acquireTokens* and *releaseTokens* requests are counted in throughput.

5.1. Resource Demand Data and Its Prediction

Samya is evaluated on a VM workload dataset published by Microsoft Azure [15]. The dataset, consisting of roughly 2 million data points, contains a representative trace of Azure's VM workload in a single geographical region collected over a month. Along with other information, it includes VM creation and deletion requests reported at discrete 5-minute intervals. A detailed description and an analysis of the dataset published by Cortez et al. [5] discusses several interesting patterns of the dataset. A noteworthy observation among them is the VM requests have nearly periodic properties over time. The authors conclude that for such requests, "history is an accurate predictor of future behavior".

5.1.1. Resource Demand Prediction. The original Azure data was pre-processed such that the number of VM creations and deletions represent demand for VMs, as depicted in Figure 3a. The figure shows periodically increasing and decreasing demand patterns in the data, indicating that a learning model can learn these patterns to predict future demands. Although the cloud computing literature consists of many sophisticated learning methods, we picked 3 simple options for resource prediction: random walk model as the baseline model, ARIMA (autoregressive integrated moving average) as a linear regression model, and LSTM, a type of recurrent neural network, as a non-linear regression model.

To choose the best predictor of the VM demands in the Azure dataset, the original one month data was split into 80% of training data and 20% of testing data. The result of our evaluation is shown in Table 2a. LSTM predicted the resource demands with highest accuracy, and hence, was chosen as the prediction module for Samya.

5.1.2. Data Processing. Since Samya is proposed as a solution for the *hot-spot* problem of aggregate data, the dataset used to evaluate Samya needs to have a high request-arrival-rate. To achieve this, we modified the original data's sampling interval of 5 minutes to 5 seconds. As a result, the same number of requests that arrived in a span of 5 minutes in the original dataset now arrived in a span of 5 seconds, generating a workload with high request-arrival-rate. Due to the shrinking of the sampling interval, the original duration of 30 days of the entire dataset was reduced to 12 hours.

Samya is a geo-distributed system with sites across different time zones while the Azure dataset corresponds to only a single geographical region in a single time zone. To generate the client requests at different regions, the original dataset is phase shifted based on the time difference between the regions. For example, if the demand in the original dataset peaks at 10 AM Tuesday and drops at 1 AM Wednesday, in our experiments, clients in North America generate peak demand load at 10 AM Tuesday *at the same time* as clients in Asia generate the reduced demand of 1 AM Wednesday – demand is phase-shifted corresponding to the time difference between North America and Asia. The phase shifting retains the periodicity in each region while avoiding peak demand in all regions at the same time. Clients in different regions generate respective phase-shifted transactional workloads where the VM creation and deletion requests from the dataset are transformed to *acquireTokens(VM, 1)* and *releaseTokens(VM, 1)* requests respectively.

5.2. Experimental Setup

The clients and servers were deployed on Google Cloud Platform where each VM was an n1-standard VM with 8 vCPUs and 30 GiB RAM. For most experiments, the VMs were placed in 5 different regions: US-West1, Asia-East2, Europe-West2, Australia-Southeast1, and SouthAmerica-East1. 3 or 5 is the typical default number of replicas used in current state-of-the-art databases [4, 26].

To simplify the evaluation, we merged the application managers and clients into a single machine. Thus, each region consisted of one VM as the client generating token acquire or release requests and another VM as the server serving client requests. In the experiments, *all five clients generated transactions simultaneously* and a client's requests were served by the closest site closest.

For MultiPaxSys, since systems such as Spanner [4] place a majority of the sites in close-by regions to achieve

faster replication time, we placed 3 out of 5 sites in different regions within the US, and 2 others in Asia and Europe.

All the experiments focused on entity VM and the maximum global limit, $\mathcal{M}_e$, was set to 5000, indicating that each site in Samya and Demarcation/Escrow starts with 1000 tokens. Note that the start allocation can also be an uneven token distribution, based on historic data. Furthermore, as indicated by the resource demand depicted in Figure 3a, a single region's demand can go beyond 1000, ensuring that sites in Samya would require redistribution.

5.3. Latency and Throughput

The first set of experiments evaluate the commit latency and throughput of the two versions of Samya and the three baselines: Demarcation/Escrow, MultiPaxSys and CockroachDB by generating load for one hour (corresponding to 60 hours in the original dataset), creating roughly 820,000 transactions, each transaction either acquiring or releasing one token of VM. The goal here is to study the behavior of the systems over extended periods of time when the workload is highly contentious.

Latency: Latency incurred at different percentiles for all four systems are tabulated in Table 2b. Since each transaction in MultiPaxSys and CockroachDB executes a replication round before responding to the client, and the workload is contentious, they incur significantly higher latencies compared to Samya. For Demarcation/Escrow, although most requests are served locally, due to the lack of prediction and an efficient escrow redistribution strategy, a resource demand peak causes latency spikes, causing its overall latency to be higher than Samya.

The interesting behaviour here is the contrast in latency numbers for Avantan[$*$] and Avantan[$\frac{n+1}{2}$]. We suspected Avantan[$*$] to outperform Avantan[$\frac{n+1}{2}$], since the latter needs to wait for a majority of responses to execute a redistribution, unlike Avantan[$*$], which can proceed with any number of responses. But the latencies in Table 2b indicate the opposite – Avantan[$\frac{n+1}{2}$] has lower latencies than Avantan[$*$] across all percentiles.

The difference in the first phase of the two versions during redistribution explains this counter-intuitive result. Avantan[$\frac{n+1}{2}$] requires a majority of sites to respond with their local token values, which the leader concatenates into a single value (i.e., $AcceptVal$). The redistribution rebalances the tokens between a majority of sites. Whereas, Avantan[$*$] collects just enough responses (consisting of local token values) to satisfy its token needs, and immediately proceeds to the fault-tolerance phase. While this greedy approach may benefit specific transactions, Avantan[$*$] rebalances the tokens between a small number of sites, causing more sites to trigger subsequent redistributions. Hence, in the long run, Avantan[$\frac{n+1}{2}$] is better at re-balancing the tokens and causing fewer redistributions. In the experiments, for the same client workload, Avantan[$\frac{n+1}{2}$] required 208 redistributions (proactive and reactive combined) whereas Avantan[$*$] required 792 redistributions.

Throughput: Figure 3b shows the throughput of all four systems when five clients concurrently send requests. Since MultiPaxSys and CockroachDB serve these requests sequentially (as they all update the same data entry), their throughput is roughly **16-18x** worse than Samya and **11x** worse than Demarcation/Escrow. This result highlights the *benefits of dis-aggregating a value to allow executing concurrent transactions*.

Between Demarcation/Escrow and Samya, the demand prediction and a more efficient redistribution strategy of Samya causes its throughput to be almost **1.3x** better than Demarcation/Escrow. The performance difference between Avantan[$\frac{n+1}{2}$] and Avantan[$*$] is due to the increased number of redistribution in the latter, which slows the rate with which client requests are served.

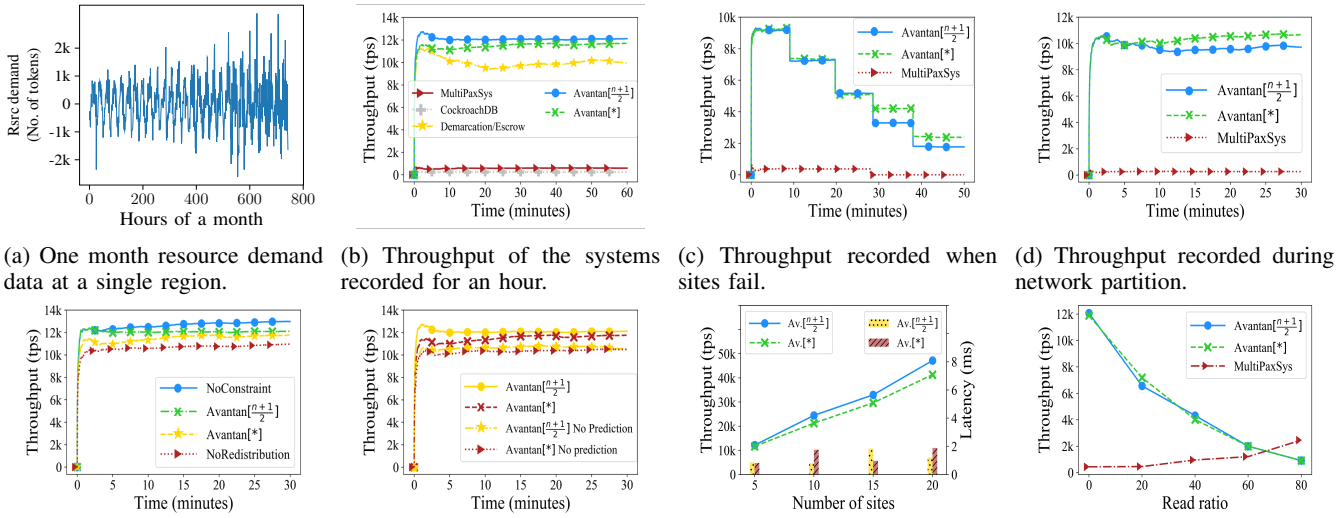
This experiment establishes the performance of MultiPaxSys and CockroachDB are comparable, hence we use MultiPaxSys as the baseline in the following experiments.

5.4. Failure Experiments

5.4.1. Crash Failures. This set of experiments evaluate Samya and MultiPaxSys when crash failures occur (Demarcation/Escrow is not evaluated in failure experiments since it requires reliable networks and hence is not fault-tolerant). The experiment starts with five regions and roughly every 10 minutes, both the site and the client in a region is crashed, until only one region remains alive, recording the throughput throughout the experiment. As indicated in the Figure 3c, once three sites crash, the throughput of MultiPaxSys drops to 0, since no transaction can be committed once a majority of the sites fail.

For the two versions of Samya, the performance is roughly the same up to 2 site failures (note that the performance is similar for both and not worse for Avantan[$*$] because in the first few minutes, the number of redistributions are low due to low resource demand in the Azure dataset; when the number of redistributions are low, the two versions perform comparably). When 3 sites fail, Avantan[$\frac{n+1}{2}$] attempts redistribution, times-out, and fails to perform any redistribution due to the failed majority. However, sites continue to serve requests that can be served locally. Meanwhile, Avantan[$*$] can successfully redistribute tokens even if only a minority of the sites are alive, thus causing its performance to be higher than Avantan[$\frac{n+1}{2}$]. when failures occur.

5.4.2. Network Partitions. This experiment measures the performance of Samya and MultiPaxSys during a 3-2 network partition, i.e., one partition consists of 3 sites and the other consists of 2 sites, and clients send transactions for thirty minutes. The results are indicated in Figure 3d. In MultiPaxSys, only the partition with 3 replicas continues to serve client requests and are up-to-date while the other two replicas are rendered stale. Its performance is significantly low compared to Samya. For Samya, although both Avantan[$\frac{n+1}{2}$] and Avantan[$*$] start off with comparable performance, once the sites exhaust local tokens and trigger redistributions, Avantan[$*$] outperforms Avantan[$\frac{n+1}{2}$], since Avantan[$\frac{n+1}{2}$] cannot redistribute tokens in the smaller network partition whereas Avantan[$*$] can.



(a) One month resource demand data at a single region.

(b) Throughput of the systems recorded for an hour.

(c) Throughput recorded when sites fail.

(d) Throughput recorded during network partition.

(e) Throughput of Samya with no constraints and no redistributions vs. Samya with redistributions.

(f) Throughput of Samya with and without predictions.

(g) Average throughput (line graphs) and latency (bar graphs) for increasing number of sites.

(h) Average throughput measured with increasing ratios of read-only transactions.

Figure 3: Various performance evaluations of Samya

The two failure experiments highlight that between the two versions of redistribution strategies for Samya, Avantan[$*$] performs better in a failure prone environment; but in an infrequent failure settings, Avantan[$\frac{n+1}{2}$] performs better as indicated in Section 5.3.

One advantage of MultiPaxSys over Samya in both failure scenarios is that MultiPaxSys can allot more tokens as long as a majority of replicas are alive, because the synchronous replication makes sure that the entire quota limit can be used. Whereas some unclaimed tokens in Samya are inaccessible temporarily until recovery.

5.5. No Constraint vs. No Redistribution

The remaining experiments focus on contrasting the two version of Avantan in Samya. In this experiment, we explore whether redistribution is worthwhile and its cost on throughput. This experiment compares Samya's performance with two baseline versions: i) *No Constraints*: there is no upper-bound on the number of resource tokens allotted, hence every requests (acquire or release) succeeds locally at a site; ii) *No Redistribution*: a constraint on the limit exists but once a site exhausts its local quota, it simply rejects the client request, rather than triggering a redistribution (neither proactive nor reactive). The results are shown in Figure 3e.

Comparing the baselines: i) Samya with no constraints is the best case scenario with optimal performance, and as seen in Figure 3e, Samya with constraints and redistributions has only 3.5-4% less throughput than the optimal throughput. ii) Samya with both versions of Avantan has about 14% higher throughput than Samya with no redistributions, i.e., 14% of the transactions would be rejected if Samya did not perform redistributions. This indicates that although executing global redistribution is expensive, the system performs better with the redistributions.

5.6. Proactive vs. Reactive Redistributions

This experiment measures the significance of predictions in Samya. The performance of four variants of Samya are measured: Avantan[$\frac{n+1}{2}$] with and without prediction, and Avantan[$*$] with and without prediction. The clients execute transactions for thirty minutes for each variant. As indicated in Figure 3f, Samya performs about **1.4x** better with predictions (for both versions). Predictions proactively prepare a site for the incoming demand and allows a site to indicate its token requirements with higher precision. This experiment highlights the advantages of using predictions in building distributed systems such as Samya.

5.7. Increasing number of sites

This set of experiments evaluate the scalability of Samya by increasing the number of sites from 5 to 20, with additional sites spawned in each of the 5 regions in which previous experiments were conducted. In this experiment, for each configuration, the clients generate transactions for 10 minutes. Figure 3g depicts the average latency and average throughput for each configuration. Samya shows a roughly linear increase in throughput as the number of sites increase, while keeping the latency constant for both versions of Avantan. This experiment highlights that Samya is highly scalable as more clients can concurrently acquire or release tokens when the number of sites increase.

5.8. Read-Write workload

This experiment compares the average throughput of the two versions of Avantan with that of MultiPaxSys, when the ratio of read-only transactions increases, as shown in Figure 3h. For Avantan, when a client issues a read request to a site S, S communicates with all the other sites to learn their current token availability, aggregates the received values, and responds to the client with a global snapshot

of the total available tokens. For MultiPaxSys, the current available tokens is read at a single leader site. This experiment highlights that when the read ratio increases roughly past 65%, the throughput of MultiPaxSys increases more than Avantan. Since reads are performed at a single site in MultiPaxSys and most writes are performed at a single site in Samya, one would expect the crossover point to be at 50%, which is not the case. The reason is that in our experimental setup, five geo-distributed clients generate requests in parallel and for MultiPaxSys, all client requests are sent to one single leader site, which sequentially processes the requests, thus incurring high latency. Whereas for Avantan, due to the decentralised design choice, write requests are typically served locally by the sites closest to the clients, in parallel. Hence, as long as an application's write load is 35% or more, it can benefit by choosing Samya.

5.9. Further Evaluations

Two additional experiments are discussed in the extended technical report [16]:

(i) Varying the maximum limit $\mathcal{M}_e$: The experiment shows that Avantan's throughput increases roughly **5x** when the maximum limit is increased from mean (600 tokens) to max demand (16000 tokens) for the specific Azure VM demand data, thus highlighting that Samya's performance improves with higher maximum limit.

(ii) Varying the request arrival rate: As mentioned in Section 5.1.2, to generate a high request arrival rate, the original data sampling interval was modified from 300 seconds to 5 seconds. This experiment compares Samya and MultiPaxSys when the request arrival interval varies from 5 seconds to the original scale of 300 seconds. The main conclusion is that even at the original request arrival rate, Avantan commits 43% more transactions than MultiPaxSys.

6. Conclusion

In this paper, we propose *Samya* – a geo-distributed data management system to store aggregate resource usage data. Samya is intended for high contention, update heavy (35% or more) workloads. Samya dis-aggregates the data and stores fractions of available resources on multiple geo-distributed sites. The dis-aggregation allows concurrent updates to hotspot resources, in contrast to sequentially ordering all concurrent and contentious updates at a leader site as in traditional geo-distributed databases such as Google's Spanner. A site in Samya serves client requests independently until, based on a learning mechanism, it predicts an increase in its resource demand that cannot be satisfied locally. This triggers a novel synchronization protocol *Avantan* to redistribute the available tokens, after which, sites continue to serve client requests independently. The experimental evaluation of Samya's performance highlights that it can commit 16x to 18x more transactions than a Spanner-like database. Evaluating Samya for applications other than resource tracking is an interesting future work.

Acknowledgements

Sujaya Maiyya is partially supported by IBM PhD Fellowship. This work is funded in part by NSF grants CNS-1703560 and CNS-1815733.

References

[1] Gustavo Alonso et al. "Partitioned data objects in distributed databases". In: *Distributed and Parallel Databases* 3.1 (1995), pp. 5–35.

[2] Daniel Barbara et al. "The Demarcation Protocol: A technique for maintaining linear arithmetic constraints in distributed database systems". In: *International Conference on Extending Database Technology*. Springer. 1992, pp. 373–388.

[3] Tushar D Chandra et al. "Paxos made live: an engineering perspective". In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. 2007, pp. 398–407.

[4] James C Corbett et al. "Spanner: Google's globally distributed database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), pp. 1–22.

[5] Eli Cortez et al. "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 153–167.

[6] Giuseppe DeCandia et al. "Dynamo: amazon's highly available key-value store". In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[7] Sheng Di et al. "Host load prediction in a Google compute cloud with a Bayesian model". In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–11.

[8] Michael J Fischer et al. "Impossibility of distributed consensus with one faulty process". In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382.

[9] Leana Golubchik et al. "Token allocation in distributed systems". In: *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*. IEEE. 1992, pp. 64–71.

[10] Zhenhuan Gong et al. "Press: Predictive elastic resource scaling for cloud systems". In: *2010 International Conference on Network and Service Management*. Ieee. 2010, pp. 9–16.

[11] Theo Härder. "Handling hot spot data in DB-sharing systems". In: *Information Systems* 13.2 (1988), pp. 155–166.

[12] *Amazon AWS Account Hierarchy*. Accessed: 2020-02-17.

[13] *Google Cloud Enterprise Hierarchy*. Accessed: 2020-02-17.

[14] *Microsoft Azure Resource Hierarchy*. Accessed: 2020-02-17.

[15] *Azure Public Dataset*. 2019 (accessed June 30, 2020).

[16] *Samya Technical Report*. Accessed: 2020-10-13.

[17] Yexi Jiang et al. "Asap: A self-adaptive prediction system for instant cloud resource demand provisioning". In: *2011 IEEE 11th International Conference on Data Mining*. IEEE. 2011, pp. 1104–1109.

[18] Narayanan Krishnakumar et al. "High throughput escrow algorithms for replicated databases". In: *VLDB*. Vol. 1992. 1992, pp. 175–186.

[19] Akhil Kumar et al. "Semantics based transaction management techniques for replicated data". In: *ACM SIGMOD Record* 17.3 (1988), pp. 117–125.

[20] Leslie Lamport et al. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[21] Sujaya Maiyya et al. "Unifying consensus and atomic commitment for effective cloud data management". In: *Proceedings of the VLDB Endowment* 12.5 (2019), pp. 611–623.

[22] Hiep Nguyen et al. "{AGILE}: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service". In: *Proceedings of the 10th International Conference on Autonomic Computing*. 2013, pp. 69–82.

[23] Patrick E O'Neil. "The escrow transactional method". In: *ACM Transactions on Database Systems (TODS)* (1986), pp. 405–430.

[24] Diego Ongaro et al. "In search of an understandable consensus algorithm". In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 305–319.

[25] Marc Shapiro et al. "Conflict-free replicated data types". In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[26] Rebecca Taft et al. "CockroachDB: The Resilient Geo-Distributed SQL Database". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 1493–1509.

[27] Alexandre Verbitski et al. "Amazon aurora: Design considerations for high throughput cloud-native relational databases". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 1041–1052.