Stabilizer: Geo-Replication with User-defined Consistency

Pengze Li[†], Lichen Pan[†], Xinzhe Yang[‡], Weijia Song[⋆], Zhen Xiao[†], Ken Birman[⋆]

[†]*School of Computer Science, Peking University*
[‡]*Pure Storage*
[⋆]*Department of Computer Science, Cornell University*

lipengze@pku.edu.cn, plch368@pku.edu.cn, xyang@purestorage.com
ws393@cornell.edu, xiaozhen@pku.edu.cn, ken@cs.cornell.edu

***Abstract*—Geo-replication is essential in reliable large-scale cloud applications. We argue that existing replication solutions are too rigid to support today's diversity of data consistency and performance requirements. *Stabilizer* is a flexible geo-replication library, supporting user-defined consistency models. The library achieves high performance using control-plane / data-plane separation: control events do not disrupt data flow. Our API offers simple control-plane operators that allow an application to define its desired consistency model: a *stability frontier predicate*. We build a wide-area K/V store with *Stabilizer*, a Dropbox-like application, and a prototype pub/sub system to show its versatility and evaluate its performance. When compared with a Paxos-based consistency protocol in an emulated Amazon EC2 wide-area network, experiments show that for a scenario requiring a more accurate consistency model, *Stabilizer* achieves a 24.75% latency performance improvement. Compared to Apache Pulsar in a real WAN environment, *Stabilizer*'s dynamic reconfiguration mechanism improves the pub/sub system performance significantly according to our experiment results.**

***Index Terms*—Data Synchronization, Geo-Replication, Stability Frontier, Consistency**

I. Introduction

Large scale cloud applications are often geographically distributed, with data replicas hosted on multiple data centers for reliability and performance. Data synchronization across those replicas is challenging due to the limited bandwidth and high latency of the wide-area network (WAN). Applications may differ in their needs for consistency and performance. For example, a banking system might tolerate higher latency to achieve a stronger safety property. An online shopping cart might prefer eventual consistency for responsiveness. A data backup service could provide multiple service level agreement (SLA) options with different reliability and cost. However, existing geo-replication solutions are too inflexible to support such diverse demands. For example, the RedBlue consistency options in Gemini [1], a widely popular replication tool, support only strong and eventual consistency semantics. Although PNUTS [2], Pileus [3], and WheelFS [4] offer more choices than Gemini, the options are still limited. *Stabilizer*, a flexible geo-replication library supporting a very wide range of user-defined consistency models with exceptional performance, responds to this need.

Distributed consistency has two aspects: ordering and durability. In a mirrored data replication system, ordering can be preserved by guaranteeing lossless FIFO data transport. Durability, however, can be interpreted in many ways [3]. *Stabilizer* introduces a simple yet flexible model: the application specifies a desired level of data stability. Options include availability zone stability (data exists at its primary site and at least one nearby data center), k-threshold WAN stability (there are at least k independent data centers with copies), and full WAN stability (all have a mirror copy). The concept of "having a copy" is also flexible, and can include acknowledgment of receipt, persistent logging, or application-supplied validation of the incoming records. These closely match the real needs of WAN replication applications seen in the cloud, distributed banking, financial, and other WAN settings [5]. In the rest of the paper, we mainly focuses on durability semantics.

Stabilizer assumes a primary-site approach where each data item has a primary owner and only the primary can update the data item. In *Stabilizer*, each site owns a pool of data that it updates and possesses mirrored read-only copies of data from other sites. The client can access data only after the desired level of stability is assured. We argue this model, coupled with a flexible notion of stability, yields a simple but powerful tool. The primary-site approach is also used in Pileus [3] and Azure [6] to avoid conflicts in concurrent writing.

Stabilizer streams control information continuously, but separately from data, using an approach inspired by the shared state table (SST) in the Derecho [7]. The basic idea is to send both data messages and control messages aggressively as long as data or receive buffering capacity is available. We treat each message as a separately sequenced object and provide a basic reliability mechanism that ensures lossless FIFO delivery. Control information is required to be monotonic: counters or other monotonic data types in which a newer value can overwrite a prior value. The incoming stream of control information drives the re-evaluation of *stability frontier predicates*, with each WAN site independently evaluating its predicates as they

evolve over time. The *stability frontier predicate* is one of our main contributions, which is the domain-specific language (DSL) of defining the consistency model.

For maximum flexibility, *Stabilizer* provides basic stability data, and allows the user to define new types of control data (from our perspective, as uninterpreted byte vectors). We support some basic consistency properties, such as "every WAN site has received all messages up to number 18", but the user can extend these with new ones coded in a simple but expressive DSL. To illustrate the value of this approach, we create a file backup service similar to DropBox [8], and a prototype pub/sub system.

This paper makes the following main contributions:

- We design an expressive DSL with a compact operator set to allow application users to define consistency models for geo-replicated data. We accelerate the DSL with a just-in-time compiler, making it extremely efficient.
- We design and implement the *Stabilizer* library for large scale cloud applications requiring various consistency models.
- We build a WAN K/V system, a DropBox-like service, and a pub/sub system using *Stabilizer*, each offering flexible consistency levels.
- We evaluate those systems both in real deployment and in an emulated Amazon EC2 WAN environment. *Stabilizer*'s stability frontier predicate model brings no additional overhead and achieves both performance improvements in end-to-end latency (as high as 24.75%) and faster stabilization when WAN reconfiguration occurs.

II. Background

A. Motivation

Many cloud applications replicate their data in geographically distributed locations. Reliable storage systems like Amazon S3 replicate data in availability zones, then create secondary geo-replicas in data centers hundreds or thousands of miles away, ensuring data survival even in the event of natural disasters. Geo-replication can support quick responsiveness in read-mostly applications such as Facebook and Twitter. However, these uses bring a variety of consistency needs. For some, a best-effort model suffices. In social media applications, it is important to preserve the "happens-before" property [9], [10]. A financial system may require a total global order in which all replicas advance in lock-step order [11], [12].

Existing options for consistency often introduce significant delays that may be sensitive to worst-case WAN latencies, and may also be difficult to describe in any single model. For example, a client might want confirmation as soon as at least one server receives/persists its data in each of the remote regions. It is hard to express this goal with popular options such as WheelFS [4], Gemini [1], or Pileus [3], forcing applications to use a stronger but perhaps a more costly model.

We argue that the geo-replication library should expose a more expressive API, enabling applications to define a more flexible consistency model that precisely matches the need and by doing so, reduce excessive user-visible delays.

Users access this flexible infrastructure through dynamically compiled modules coded in the *Stabilizer* DSL. The DSL language allows users to specify new predicates that consist of monotonic logical tests against a local repository of control information received from other WAN nodes. Each WAN node independently and concurrently evaluates its own predicates. The DSL also has an API with which it can report its own stability data, supplying a type name and an associated value. Enabling the programmable ability to enhance the functionality of systems like KV storage system is also used in EventWave [13] and Malacology [14].

For efficiency, these DSL modules are compiled on first use, then invoked at low overhead as needed. *Stabilizer* uses the experimental just-in-time (JIT) compilation support in C/C++ called *libgccjit* [15], a compiler backend creating and linking binary code at run-time. We implement the compiler front end for our DSL and then build its corresponding binary using *libgccjit*. Costs of the approach are evaluated in section VI.

B. Related Work

Consistency protocols and related systems: Our flexible approach to consistency has some similarities to Brewer's argument that a system should consider relaxing consistency if this can yield improved availability, performance, and partition tolerance [16]. Existing studies mostly focus on the trade-off between consistency and availability and propose consistency guarantees for different scenarios, whereas our approach allows individual designers to hand-craft desired consistency predicates. Examples of prior work include solutions that favor consistency and partition tolerance, but with rigid semantics [11], [12]. Examples of platforms supporting eventual consistency include [2], [17]. Causal consistency is offered by [9], [10]. Readers interested in the tradeoffs are referred to the survey in [18], which reviews common consistency protocols from the weakest to the strongest consistency , looking at semantic and performance tradeoffs, but concluding that different applications may need different solutions, an option *Stabilizer* introduces.

Systems that provide more consistency options: Pileus [3], WheelFS [4], and Wiera [19] provide more consistency options, which are closer to *Stabilizer*. The main difference between *Stabilizer* and them lays in the usability and flexibility of the system in providing user-defined consistency models rather than the performance of these systems. Consequently, the direct comparison with experiments between *Stabilizer* and the related work is actually comparing the mechanism of providing user-defined consistency behind them, which will raise an apples-to-oranges concern. Moreover, as illustrated in the later section, some consistency models can only be expressed by *Stabilizer*.

In both the related work and *Stabilizer*, users choose the desired consistency model and use it via the system's interface: Pileus defines the consistency model selected by the user in the SLA. Wiera defines the consistency model in code implementation. WheelFS defines it in the file path. *Stabilizer* enables its users to define the consistency model with our

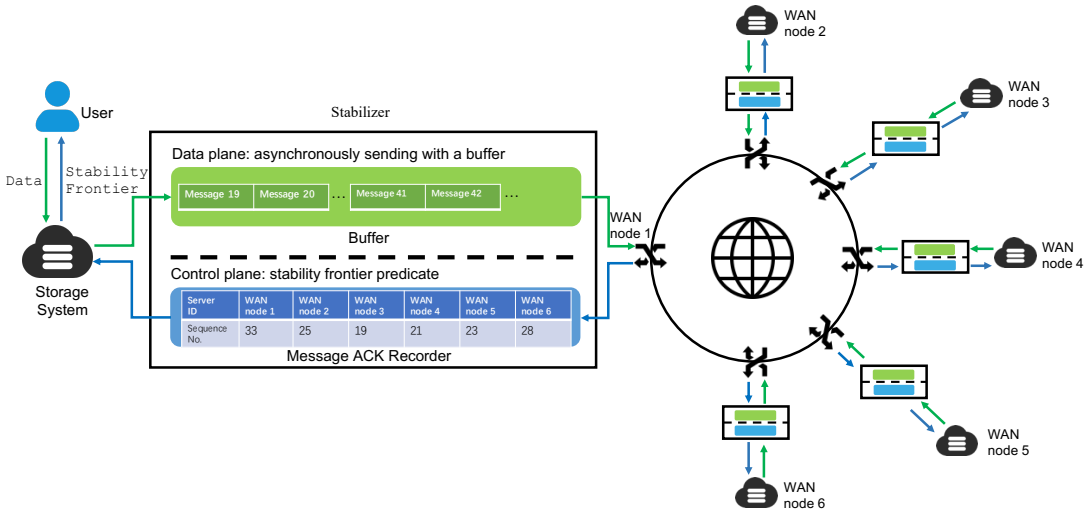


Fig. 1. *Stabilizer* system overview: structure, workflow, data plane, and control plane

designed DSL in the configuration file at start and to change or to register new consistency models with APIs at run-time.

In terms of the user-defined consistency, Pileus and WheelFS cannot allow users to customize the consistency model, while Wiera requires users to implement their own consistency model in Java with their consistency model superclass, which is limited in flexibility. As an example, the user with a special region-based goal mentioned in the later section IV would not be able to express that goal with either of these approaches, whereas *Stabilizer* can easily and accurately support that property.

To enable the selected consistency model: Pileus designs an algorithm to choose the best matching sub-SLAs: consistency and latency requirement pairs defined by users. In Wiera, users can select four types of writing consistency in its policy profile. WheelFS allows users to define information such as consistency (whether to use the default consistency or the eventual consistency) and replication level in the file path. *Stabilizer* provides more consistency model options than them, since users can use the DSL to flexibly create new consistency models and register them with *Stabilizer*'s interfaces. More details of the design and the usage of *Stabilizer* are introduced in later sections.

III. *Stabilizer* Design

A. System Overview

Modern storage systems such as Derecho's object store [7], etcd [20], and leveldb [21] have been shown to achieve high levels of performance by leveraging data center technologies such as RDMA and NVMe, but lack geo-replication support. Accordingly, *Stabilizer* is designed as an add-on for existing solutions that currently run purely in a single data center, giving them the read-only mirroring capability. For example, our WAN K/V store allows full read-write access to a pool of locally owned keys, and then offers read-only access to K/V pairs owned by other WAN nodes, updating them to track the evolution of those remote K/V objects.

Fig. 1 shows a typical workflow of the enhanced geographically distributed storage system, focusing on a simple case: monotonic acknowledgment messages (ACKs) that confirm the arrival of mirrored data. Users directly interact with some applications on the local WAN node. Each update triggers *Stabilizer* thread, which sends the data to its remote counterparts in sequenced messages, delivering data to the remote application via upcall. The progress of data through the pipeline corresponds to a series of levels of stability: received, persisted within *Stabilizer* storage layer, delivered to the application. Besides the received and persisted stability, *Stabilizer* also supports users to define their desired stability.

The remote system might not receive upcalls for each value that is sent. Suppose that WAN node A is streaming data at high speed, including messages X and Y. On the remote system, it is possible that a stability report for X is overwritten by the report for Y. Here, our monotonicity requirement enters the picture: the application is expected to understand that the upcall for Y "implies" the stability of messages prior to Y. For example, if X had message sequence 17, and Y has sequence 91, the "received" stability property for Y implies that X was also received. Internally, *Stabilizer* is single-threaded, and this feature permits the system thread to perform a batch of actions, then report them via stability upcalls, rather than doing upcalls message by message. Besides, since all write operations are issued at the primary node, *Stabilizer* only needs to keep one set of the sequence number from the primary node's view.

Fig. 1 demonstrates the two key building components of *Stabilizer*, namely *data plane* and *control plane*. The *data plane* sends and receives data aggressively to saturate WAN bandwidth (sending throughput can get close to the physical bandwidth limit), while the *control plane* exploits stability frontier predicate to track data synchronization. As shown in Fig. 1, the control information is stored in a message

ACK recorder and get updated when receiving an ACK from other nodes. Each WAN node detects stability independently and asynchronously, but all WAN nodes reach the same conclusions eventually. This asynchronous stability detection design guarantees the data plane and control plane separation.

B. Data Plane

A property of our solution is that it can maximize utilization of WAN bandwidth by sending data aggressively as soon as it has been assigned a sequence number, but it can also buffer data for later transmission if needed. When a message has been delivered everywhere, the buffer space is reclaimed. The performance of the WAN bandwidth utilization is evaluated with experiments in section VI. The overall approach is in contrast with classic WAN consistency mechanisms, such as protocols based on Paxos [22] that block message sending when all leaders are busy exchanging control information.

C. Control Plane

The monotonic stream of stability reports enables each WAN node to track the global status of each message. For completeness, we adopt the rule that all stability properties hold for the WAN node that originated a message, including user-defined properties. Thus, when *Stabilizer* first learns of message k, it immediately assumes that k has locally been delivered, verified, etc. With this in mind, we can now focus on the DSL, *stability frontier predicates*. Before giving a formal definition of the predicate, we first introduce its components.

Operators:

- The *MAX* / *MIN* operator returns the maximum/minimum number in a set of integers.
- The *KTH_MAX* / *KTH_MIN* operator returns the k-th largest/smallest number in a set of integers.

Operands: The operand in the predicate has the form of $#, where the # is the WAN node index. For instance, the WAN node 1 in Fig. 1 can be referred to as $1. Data centers have unique names in most cloud settings, hence we map them to numbers. *Stabilizer* configuration file includes a list of data centers where the system has been deployed. Within this list, a subset notation designates availability zones. Thus when *Stabilizer* is launched it can look up its own data center name and convert this to an index number, learning its own rank in the overall list, its availability zone neighbors, etc. Besides, the predicate also supports macros and variables to represent the operand.

Macros and variables:

- **Macros:** $ALLWNODES expands to a list of all WAN nodes, and $MYAZWNODES expands to list all WAN nodes in the availability zone of the WAN node executing the DSL. $MYWNODE expands to the local WAN node.
- **Variables:** $WNODE_# refers to the WAN node with name #. $AZ_# is used to represent the WAN node collection in availability zone #. The WAN node name and availability zone name are from *Stabilizer* configuration file. For example, if there is a node named *Foo* in the availability zone *Wisc*, they can be referred to as $WNODE_Foo and $AZ_Wisc.

Function tools:

- SIZEOF returns the number of WAN nodes in its parameter. Besides, the predicate also supports further calculation with the result of SIZEOF: *SIZEOF($ALLWNODES) / 2 + 1* is the number of WAN nodes required to achieve a majority. This function is only used in the *KTH* operators.
- The binary operator '-' calculates the difference between two WAN node sets. In some situations, it is useful to exclude the sender WAN node from a set; for this purpose, we write *$ALLWNODES-$MYWNODES*. *$ALLWNODES-$MYAZWNODES* is the set of WAN nodes remote from the sender's availability zone.

Suffixes: Within predicates, we use a $#.type notation where # is the WAN node id (operand) to represent the highest sequence number acknowledged by that WAN node for a given type of ACK, where .type could be .received, .persisted or .xxx (here, "xxx" would be some application-defined stability level such as verified, countersigned, etc). If the .type is omitted, we assume .received as the default. This notation also extends to sets. For example, *($MYAZWNODES-$MYWNODE)*.verified expands to a list of WAN node-ids for members of the sender's availability zone other than the sender itself, then selects the "verified" property (an application-defined monotonic property).

A predicate p has the following simple but variadic form:

$$p = O(x) \tag{1}$$

$$O \in \{MAX, MIN, KTH\_MIN, KTH\_MAX\} \tag{2}$$

The x is the parameter list, comprising operands, macros, variables, functions, and other predicates if needed.

The macros and variables will be finally expanded to the corresponding operands. For instance, in Fig. 1, the predicate *MAX($ALLWNODES-$MYWNODE)* would be auto-expanded at WAN node 1 to $MAX(\$2, ..., \$6)$ and returns the highest sequence number acknowledged by any remote WAN node, which is 28 in this case.

Have we written *MAX($MYAZWNODES-$MYWNODE)*, we would be looking at receipt acknowledgments within the local availability zone. Although weak, this level of consistency still could be adequate, as a confirmation of availability-zone replication for backup purposes. In contrast, *MIN($ALLWNODES)* tells us that *every* message up to the computed minimum is stable: a much stronger guarantee. The *KTH_MIN((SIZEOF($ALLWNODES)/2+1), $ALLWNODES)* predicate reports a message to be stable if it has been acknowledged by a majority of sites (the sender included). By combining the operators and leveraging multiple types of ACKs, a stability frontier predicate becomes surprisingly expressive as we show in Section IV.

D. Interfaces

Stabilizer provides many interfaces to utilize its functionality.

- **One-time stability frontier update trigger.** The application can wait for certain stability by calling *waitfor(sequence-number, predicate-key)*. The *predicate-key* is the name of the predicate, and we use a map data structure to store all the predicates in *Stabilizer*. This method will block until the stability result of the desired predicate with name *predicate-key* holds for the message with the specified sequence number.
- **Stability frontier update monitor.** In *Stabilizer*, *monitor_stability_frontier(predicate-key, lambda)* registers the function *lambda* to do the work specified by the user each time the corresponding predicate returns a new stability frontier. The *lambda* will be passed the sequence number that is most recently generated from the desired predicate, and the application-defined additional data object if any (as an uninterpreted byte vector).
- **Predicates registering and changing.** Users define new predicates using a simple yet expressive DSL, placing the code either in *Stabilizer*'s configuration file before launch, or changing them on the fly with *Stabilizer*'s API *register_predicate(predicate-key, predicate-string)* for a new one and *change_predicate(predicate-key)* to update an existing one at runtime. The *predicate-string* is a predicate sentence like *MIN($ALLWNODES)* that the user can understand. *Stabilizer* handles new predicates with a just-in-time checking and compilation sequence. First, we employ *Flex* [23] and *Bison* [24] for lexical and grammatical analysis, then use *libgccjit* to generate a compiled binary to calculate the *stability frontier*. The binary can then be called without significant overhead, even on the critical path.

Since *Stabilizer* is a library, it needs to be integrated with a local storage system to utilize its functionality. So the user (the client of *Stabilizer*) is not the traditional end-user that accesses the storage system but the developer/administrator of the storage system. Consequently, those interfaces will not be called arbitrarily or meaninglessly (like concurrent invocations). They only can be called by the system designer at the code level with proper logic.

E. Fault Tolerance

Recall that all write operations in Stabilizer are performed at the primary node. We need to consider two types of failures: 1) The primary node crashes. 2) Secondary nodes crash. Starting from the simpler second situation: the crash of the secondary node will only affect the advancement of the stability predicate involving this crashed node. The crashed secondary node can be observed by a predicate update timer or the data transmission failure information. The primary can adjust the predicate to eliminate the impact.

If a primary node crashes, it should rely on the integrated system to restart. For instance, *Stabilizer* can make use of the fault tolerance part of Derecho's object store. When a Derecho node restarts, it will invoke the Derecho's *view change* logic. In the *view change* process, the restarted node will rejoin the top-level Derecho group and updates the corresponding *Stabilizer*'s configuration file and initiates *Stabilizer* process. Moreover, the Derecho object store can also persist the stability frontier information, which can be used for *Stabilizer* recovery. Many systems like Orbe [25], COPS [9], etc., also have fault tolerance functionality. When they are integrated with *Stabilizer*, they can restart *Stabilizer* after their recovery logic. Related work like Wiera [19] and Pileus [3] only provide the basic or did not demonstrate the fault tolerance mechanism.

IV. *Stability Frontier Predicate* Use Cases

A. Using Stability Frontier Predicate in AWS Regions

As a first illustration of the approach, we consider the problem of tracking the stability of data in AWS. Amazon's data centers are deployed globally, with small groups of three nearby but fault-independent data centers forming *an availability zone*. Latency is low between data centers in the same availability zone, and the user might wish to leverage this to define a notion of "locally stable". Larger-scale stability would be more costly because long-haul WAN links enter the picture, but offers a higher level of protection. For example, regional disasters could somehow shut down all three availability zone members. As shown in Fig. 2, we select four regions with eight availability zones according to the introduction of AWS [26]. The region and availability zone topology is also used in performance evaluation.

Traditional consistency mechanisms are indifferent to topology, making it very hard to express constraints such as that "the event is fully replicated within the availability zone of the sender, and is also geo-replicated to at least one remote site". With our DSL, this is easily expressed: *MIN(MIN($MYAZWNODES - $MYWNODE), MAX($ALLWNODES - $MYAZWNODES))*. Here, the first part of the predicate checks that the sender's messages have been receive-acknowledged by all WAN nodes other than the sender in the sender's availability zone, and the second part confirms that it has been acknowledged by at least one remote WAN node, not in the sender's availability zone. Such a message is geo-replicated and hence will not be lost, and is also availability-zone replicated hence even if the sender data center fails and rolls over to one of its availability-zone peers, the service does have a copy.

B. Implementing the Quorum protocol with Stabilizer

Similar to the stability frontier predicate in section III-C, which is used to perceive the write progress, we also extend the predicate for the read operations. In this section, we demonstrate how to implement the Quorum protocol [27] with the read and write predicates.

The Quorum protocol is a classical distributed algorithm with a serial consistency guarantee for accessing replicated data. It makes sure that a reader always sees the data committed by a previous non-concurrent write. The quorum refers to a set of replicas that any two of them overlap. A successful read operation returns the latest version of the responses from at least N_r replicas, referred to as a read quorum; and a successful write operation must write to at least N_w replicas,

referred to as a write quorum. The quorum protocol enforces that $N_w + N_r > N$, where N is the total number of replicas. Therefore, at least one replica in the read quorum falls in any write quorum and hence has the latest data.

It is easy to express the quorum protocol using *Stabilizer* predicates. Suppose that we want the write quorum to be a majority of WAN nodes in our system, and a read quorum to be any set overlapping the write quorum. The write predicate is written as *KTH_MIN(SIZEOF($ALLWNODES)/2+1, $ALLWNODES)*, meaning that a write has been completed once ACKs from a majority of the WAN node set (including the sender) are received. Similarly, a read predicate can be written as *KTH_MIN(SIZEOF($ALLWNODES)/2, $ALLWNODES)*. This can easily be varied, for example, to express that a write quorum includes all WAN nodes and that any single WAN node can be used for reads, etc.

V. *Stabilizer* Applications

A. K/V Store

Next, we integrate *Stabilizer* with Derecho object store [7] to show how this library empowers a storage system that otherwise lacks the ability to synchronize data across geographic regions. The modified version can track data synchronization progress based on the user-defined consistency model.

The Derecho object store is an existing system that efficiently leverages modern data center hardware to deliver high-throughput, low-latency, and fault-tolerant distributed key-value storage services. The object store is typically deployed on multiple servers but within a single data center [7]. Our enhanced version offers each WAN node (each data center) the ability to originate K/V updates to local data, but to read K/V data from any WAN node.

The enhanced K/V maintains the original APIs, such as *put, get, get_by_time*, etc. (see [7] for details). When a client calls *put*, the Derecho stores data locally, then *Stabilizer* buffers the new records and starts an asynchronous transfer to mirror the data remotely. Thus, the semantic of *put* is that upon completion the action is locally stable. A client seeking a stronger guarantee would request a *stability frontier* matched to the consistency model. To this end, we add an API *get_stability_frontier*. We also extend the K/V store API with two functions called *register_predicate* and *change_predicate*. A user application call *register_predicate* to register new predicates defined with the DSL in Section III-C. The *change_predicate* function allows the application to switch among pre-registered predicates.

A K/V store is a form of middleware. To evaluate a use case, we implemented a file backup service over our WAN K/V system, designed to offer geo-replication in a manner like Dropbox [8]. This application supports several stability frontier predicates: *OneWNode*, *OneRegion*, *MajorityWNodes*, *MajorityRegions*, *AllWNodes*, *AllRegions*. For example, a new file can be dropped into the system and then the application can wait until the data has reached a majority of WAN data centers before allowing access to the contents. With a traditional Dropbox, the actual semantics of uploading a file are unspecified, and fine-grained control is not possible.

B. A Pub/Sub Service

The pub/sub service is a message bus connecting the distributed components of a scalable application. It supports publishers that produce messages, subscribers that consume messages, and brokers that pass messages from the publishers to the subscribers [28]. To isolate unrelated traffic the publisher and subscriber specify one or more topics that name messages they care about. In a typical pub/sub implementation, when deployed across WAN, each data center has one broker managing local publishers, subscribers, as well as their subscribing state. The brokers synchronize messages with each other to get their local subscribers messages of a topic whose publisher is far away, and send messages of a topic to remote subscribers.

We build a pub/sub service prototype in WAN based on *Stabilizer*. For simplicity, we omit topic management and focus on the performance for a single topic. The solution wraps the *Stabilizer* library with a thin layer to provide broker service interfaces such as *publish* and *subscribe*. The *publish* API merely multicasts the data to remote peer brokers through the asynchronous data plane. The *subscribe* API allows a client to register a callback function to handle incoming messages on the unified topic. After receiving a first subscription request, the broker becomes active as a member of the active broker list. Thanks to *Stabilizer*, the publisher can keep track of the progress of messages transfer using the stability frontier predicate dynamically managed by the broker.

Some pub/sub solutions offer persistence, but our prototype currently lacks this feature (like support for multiple topics, persistence would be easy to introduce, but beyond the needs of our experimental evaluation). Accordingly, in section 6.3, we compare the performance of our pub/sub service prototype with Apache Pulsar [29] using non-persistent topics.

VI. Evaluation

We emulate a WAN environment using eight servers with 24GB memory, 16 cores Xeon E5620 CPU in a local cluster. Each server has a Gigabit network card (NIC) connecting to a Gigabit rack top switch. We use those servers to emulate eight WAN nodes in four AWS regions in Amazon EC2, as shown in Fig. 2. Server 1 is the sender and the rest are the receivers. We use *TC* [30] to inject the latency and control the bandwidth between the servers to match Amazon EC2 network performance observations in Table I. To prevent the Gigabit NIC and switch from becoming a bottleneck, we throttle the emulated bandwidth to halves of the observed values.

In addition to the emulated environment, we also use five physical servers in CloudLab [31] as a real-world WAN experiment setup. We use three servers, one in each of the Clemson (CLEM), Wisconsin (WI), and Massachusetts (MA) clusters. We then pick two servers referred to as Utah1 (UT1) and Utah2 (UT2) from the Utah cluster. Because Utah1 is the sender in the following experiments, we list the bandwidth and latency between Utah1 and the rest of the servers in Table II.

TABLE I
NETWORK STATUS BETWEEN NORTH CALIFORNIA AND OTHER REGIONS

	Lat (ms)	Thp (Mbit/s)	Half Thp (Mbit/s)
North California*	3.7	667	333.5
Ohio	53.87	89	44.5
Oregon	23.29	113	56.5
North Virginia	64.12	74	37

*The network status between availability zones in North California region

We use six predicates in Table III for evaluation and to demonstrate how to define consistency models with the (default) .received type. The first three predicates define consistency models at regional granularity. The *OneRegion* predicate returns the maximum acknowledged sequence number from any WAN node in any remote region. In other words, *OneRegion* predicate claims a message is stable if any WAN node in a remote region acknowledges it and all prior messages. The *MajorityRegions* predicate returns the maximum sequence number acknowledged by a majority of the remote regions, while the *AllRegions* predicate returns the minimum out of all the maximum sequence numbers acknowledged by each remote region. Please note that if an ACK from any WAN node in a region is received, we say the message is acknowledged by that region. Similarly, the other three predicates define consistency models at WAN node granularity.

TABLE II
NETWORK PERFORMANCE BETWEEN UTAH1 AND OTHER SERVERS

	Utah2	Wisconsin	Clemson	Massachusetts
Thp(Mbit/s)	9246.99	361.82	416.27	437.11
Lat(ms)	0.124	35.612	50.918	48.083

TABLE III
PREDICATES USED IN OUR EXPERIMENT

Name	Predicate
OneRegion	MAX(MAX($AZ_North_Virginia),MAX($AZ_Oregon),MAX($AZ_Ohio))
MajorityRegions	KTH_MAX(2,MAX($AZ_North_Virginia),MAX($AZ_Oregon),MAX($AZ_Ohio))
AllRegions	MIN(MAX($AZ_North_Virginia),MAX($AZ_Oregon),MAX($AZ_Ohio))
OneWNode	MAX($ALLWNODES-$MYWNODE)
MajorityWNodes	KTH_MAX(SIZEOF($ALLWNODES)/2+1, ($ALLWNODES-$MYWNODE))
AllWNodes	MIN($ALLWNODES-$MYWNODE)

In this section, we evaluate *Stabilizer* to answer four questions with experiments:

1) How does *Stabilizer* behave in the microbenchmarks like the reading latency and the DSL compilation cost?
2) How effective is *Stabilizer* in reducing the end-to-end latency when a more precise or flexible consistency granularity is needed?
3) When a pub/sub system is implemented with *Stabilizer* as the module for sending data, how does *Stabilizer* perform compared to a popular pub/sub system in throughput and latency?
4) What benefits in user-perceived latency does the flexibly changing consistency level mechanism bring?

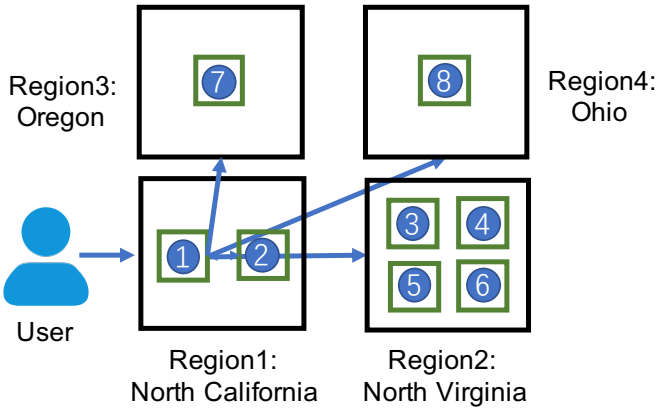


Fig. 2. Experimental topology of the servers in four regions

A. Microbenchmarks

Quorum read experiment: We measure the latency of the Quorum read operation using four servers in CloudLab. The three quorum server processes run on Utah1, Wisconsin, and Clemson servers, respectively. We set both the read and the write quorums to two. We then run a writer in Utah2 and a reader process in Utah1. We evaluate the read latency, measured from the time a message is sent from the sender to the time it is received by the reader. In Fig. 3, the black dashed line represents the Round-trip time (RTT) latency measured from Utah1 to each site using the ping command. The quorum read latency is comparable to the RTT of Wisconsin. This is because the Wisconsin server is the second-fastest server among the three quorum members when reading or writing from the Utah cluster. Therefore, it is usually the response from the Wisconsin server that satisfies a quorum read request. As the messages become larger, the time required to transfer the data increases, resulting in a slight increase in latency. The result of the Quorum read experiment proves that we implement the Quorum semantics correctly with *Stabilizer*.

The performance overhead of the user-defined consistency mechanism: We evaluate the overhead of compiling and computing predicates with 1 to 5 operators and 5 to 20 operands. The operand is the WAN node's index number illustrated in section III. Since each operand can represent one WAN node, this experiment setup can reflect the geo-replication factors for small to large cloud applications. The maximal computing and compilation time comes with five *KTH_MIN* operators and 20 operands, which are about 0.2 ms and 30 ms, respectively. We argue that such a one-time compilation overhead is acceptable.

B. Dropbox-like Application Experiment

Next, we evaluate our Dropbox-like file geo-replication backup application, which starts by extending the Derecho

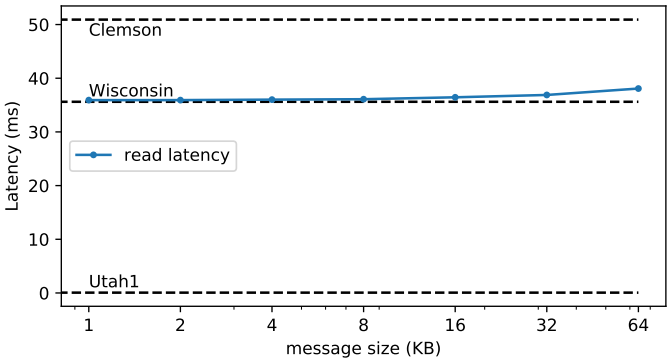


Fig. 3. Latency of read operation

object store into a geo-replicated K/V store, and then layers simple file storage and retrieval API over the K/V API, with user-customizable WAN consistency. The predicates we use are shown in Table III. To establish a baseline, we also compare *Stabilizer* with a state-of-the-art industrial implementation of the Paxos protocol, the PhxPaxos [32]. The topological structure of the servers on which the Dropbox-like file backup application is deployed is shown in Fig. 2. We assume that all user write requests will be sent to server No. 1 which will synchronize data to other WAN nodes.

Trace used in the experiment: To drive our experiment, we obtain a file storage trace from six cloud storage services in the real world [33]. It records the users' sync request activities in different countries. This trace includes several files classified by cloud storage service type and service users. A characteristic of this trace is that most of the sync requests in each day are concentrated within one hour or several minutes. Therefore, we selected a Dropbox [8] trace with a relatively high rate of sync requests. This period describes the users' sync activity from 16:40:45 to 16:57:08 in 2012-09-20 and includes a total of 3.87GB of data. Fig. 4 shows the time statistics of the trace we use in the experiment.

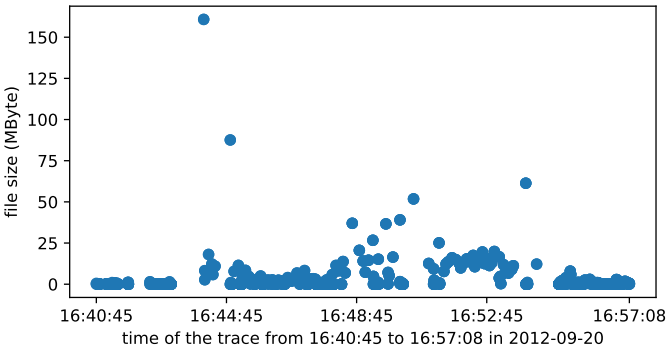


Fig. 4. Dropbox file size distribution in 17 minutes

Trace-driven experiment: We use the records in the Dropbox trace as input to evaluate the Dropbox-like file backup application. In the trace, each record specifies when a file sync request is submitted and the size of the file; our emulation issues the same request to the file backup application using files filled with random bytes. *Stabilizer* splits big writes into smaller packets whose upper bound is 8KB, so we get 517,294 messages in total to be sent. Each time an ACK arrives, *Stabilizer* recalculates the stability frontier in a predicate-specific manner and records the time when each message's synchronization progress satisfies the consistency model specified by each predicate. For example, when the WAN node sees ACKs from all other WAN nodes for message No.10, *Stabilizer* records this timestamp for the predicate *AllWNodes*. Fig. 5 shows the first timestamp when a message's synchronization progress meets each predicate since the message is sent out in the trace-driven experiment.

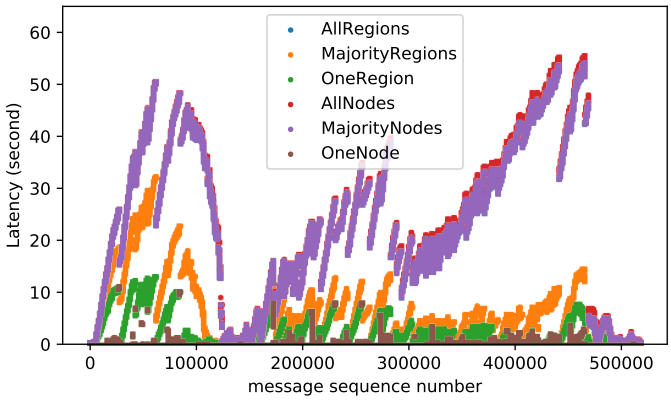


Fig. 5. Stability Frontier latency

As shown in Fig. 5, we can notice three spikes for each stability frontier predicate. To understand this, we need to refer to Fig. 4. Recall that we split files into packets up to 8KB each, so that the arrival of a huge file or dense arrival of small files both bring a lot of messages to send, resulting in a tremendous burden for WAN. It turns out that the three spikes observed in Fig. 5 correspond to the transmission of records associated with huge files. The weaker levels of consistency are less impacted, while the stronger ones, which depend on ACKs from more WAN nodes, are more sensitive to delays. Thus in the figure, we see that *MajorityWNodes* is more vulnerable to load spikes than *MajorityRegions*. A user who needs geo-replication but does not need to wait for every WAN node to receive the file would benefit from lower delay, while a user who really does want every WAN node to have the file can configure the system to match that specific need.

File-based experiment: Next, we construct an experiment that compares our performance to that of a widely-used industrial Paxos product. The interesting case here involves situations the user has a topological basis for customizing the consistency predicate. The Paxos is typically indifferent to topology and has a fixed obligation to respect the quorum-based log update rule built into the protocol. In the topology showed in Fig. 2, the Paxos protocol needs to ensure the data synchronization is done on at least four servers: server No.7, No.8, and any two of No.3-No.6. In contrast, a user content

with the *MajorityRegions* predicate should have an advantage: for this predicate, we only need to await data synchronization with respect to two of the three servers: No.7, No.8, and any single server in the region of North Virginia. The Paxos product we select, PhxPaxos [32], is considered to be a state-of-the-art technology for such uses.

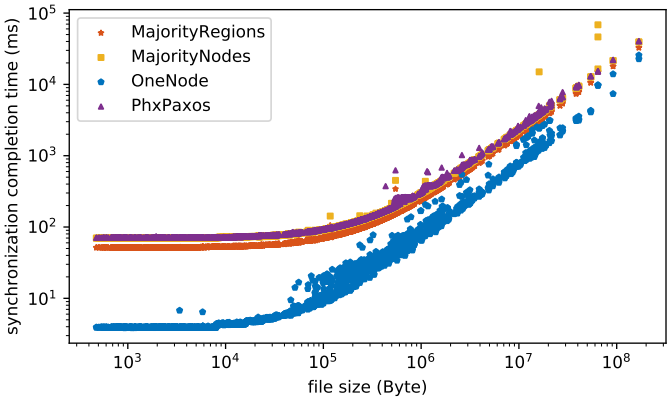


Fig. 6. File synchronization time

Our experiment records the time to synchronize each file from the trace under different predicates semantics. The recorded time includes the time for all messages divided from each file to be synchronized. Simultaneously, to avoid the possible queuing caused by multiple files arriving in a short period, we only test the time required for one file to complete synchronization at a time. Results in Fig. 6 show that when the PhxPaxos is used as the consistency model, the time required to complete the synchronization of files is similar to the performance of using *MajorityWNodes* predicate (the two curves mostly overlap). Moreover, it is longer than the time required to use *MajorityRegions* predicate, and this difference becomes larger as the file becomes larger. According to our analysis, using *MajorityRegions* predicate to describe the majority of regions' data synchronization progress can bring an average of 24.75% end-to-end latency improvement over using PhxPaxos. Furthermore, if the difference in the number of WAN nodes and bandwidth between regions becomes more extensive, the difference between *MajorityRegions* predicate and PhxPaxos will be more significant. Clearly, the selection of the consistency model best matched to the needs of the application can avoid unnecessary waiting.

The experimental results of trace-driven and file-based show that it is feasible to use our DSL to construct different consistency models. Moreover, the difference between the different stability frontiers reflected in the trace-driven and file-based experiments shows that if the application does not select the most suitable consistency model, end users may waste time unnecessarily. The flexibility of the DSL predicate model we employ allows applications to craft elaborate stability frontier predicates that closely match needs, a possibility that opens new options relative to today's very limited choices.

C. Pub/Sub Service Experiment

In the pub/sub service experiment, we compare our pub/sub service prototype based on *Stabilizer* with Pulsar. Pulsar [29] is a Java distributed pub/sub messaging system that supports geo-replication, non-persistent messaging, and that uses non-blocking IO to transmit data on WAN links. We compare Pulsar with our pub/sub application of *Stabilizer*, for cases with identical workloads and requested consistency semantics.

In the design of the pub/sub comparison experiment, a client can publish messages at a range of frequencies. Queues in Pulsar have a fixed size limit, and the experiment would not exceed this limit. Accordingly, our experiments send 10,000 8KB messages each, a volume of data that will not cause the Pulsar queue to overflow. We deploy the resulting performance test on the real environment described in Table II.

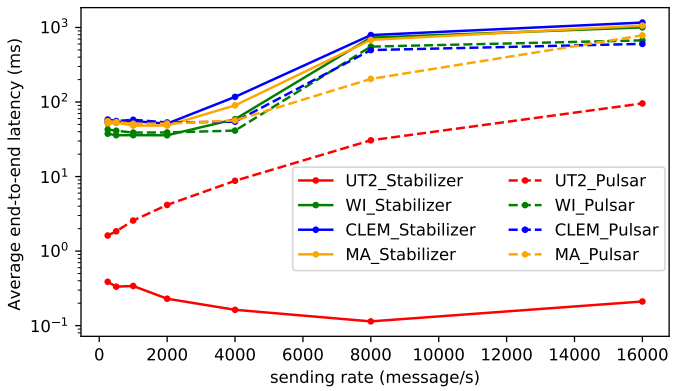
Recall our prototype omits persistence, and that we also configure Pulsar to run in its volatile mode as well. Here, a very small change to Pulsar is necessary, to prevent the broker from discarding messages if a WAN link is transiently slow. In the original non-persistent topic geo-replication of Pulsar, if the local broker finds that the link to the remote broker is temporarily inaccessible it turns out that the local broker will silently abandon sending the message. Our change introduces buffering and ensures that Pulsar continues to try, eventually sending all messages and preserving sender order.

To measure the end-to-end latency, we focus on publisher streaming data, and on the ACKs that stream back as data arrives and can be delivered to remote subscribers. Using a suitable predicate, the publisher can calculate the end-to-end latency by tracking ACK arrival times and subtracting the corresponding message send times. Our experiment measures throughput at all WAN nodes: we are curious to see whether certain sources experience unusual behavior. Bandwidth is computed as the total size of the messages divided by the time from the beginning to the arrival of the last message. Please note that latency and throughput are for each pair of publishers and subscribers.

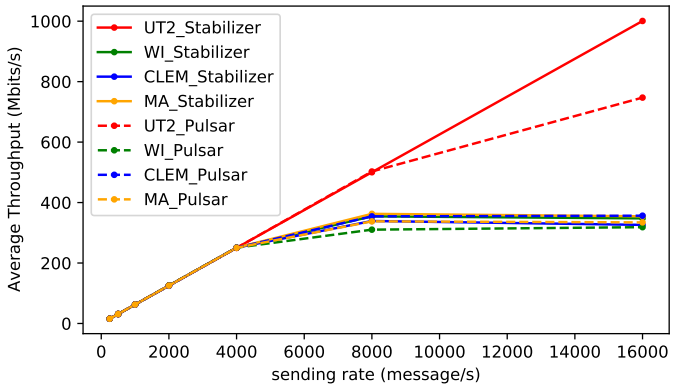
Fig.7 shows the measured throughput and latency with varying message rates from 250 to 16000 messages per second. Note that when the sending rate exceeds the bandwidth, queuing delay is incurred and the latency rises sharply. All lines except red ones in Fig.7 reveal that both systems bottleneck at the same throughput, with comparable latency. This shows that when there is enough sending data, *Stabilizer* can send data on WAN "at full speed" (close to the physical bandwidth bottleneck). The red lines, on the other hand, represent the performance on LAN, between Utah1 and Utah2. Here, bandwidth is so high (10Gb) that at the transmission rates we use, it is not possible to create a backlog. Nonetheless, Pulsar shows growth in latency. We believe this is associated with garbage collection within its JVM. As seen in the figures, our prototype is as fast or faster than Pulsar in all scenarios we explore.

D. Dynamic reconfiguration experiment

We make a reliable broadcast application based on the pub/sub prototype system in this experiment. The reliable



(a) Latency



(b) Throughput

Fig. 7. *libgccjit* Latency and throughput performance in pub/sub experiment

property requires that the broker at the publisher has to ensure that every broker with any subscriber will receive the message. However, a subscriber in this application can subscribe or unsubscribe at any time. Sometimes, a remote data center might not have any subscribers for a topic, say T. In such a case, a publisher does not care whether the messages in T reach that data center or not. *Stabilizer* allows dynamic predicate reconfiguration so that the application will not wait unnecessarily. We show *Stabilizer*'s dynamic reconfiguration in the environment shown in Table II.

We assume a subscriber on the slowest site who subscribes and unsubscribes on the local broker from time to time. Accordingly, *Stabilizer* will add/remove the slowest site from the observation list for data synchronization via changing predicate. In this experiment, we send 1600 8KB messages at 80 messages per second and count the end-to-end latency of each message under the particular predicate. The simulated client subscribes/unsubscribes every five seconds, and accordingly, *Stabilizer* adjusts its current predicate every five seconds. We run the experiment with two predicates: the *all_sites* predicate report the sequence number acknowledged by all remote sites, and the *three_sites* predicate reports that are acknowledged by at least three remote sites. As the two predicates report on different stability frontiers, there might be a gap when the predicate shifts. For example, when the stability frontier is 1000 with *three_sites*, the stability frontier with *all_sites* is still 900. We argue that the user should be responsible for handling such a gap.

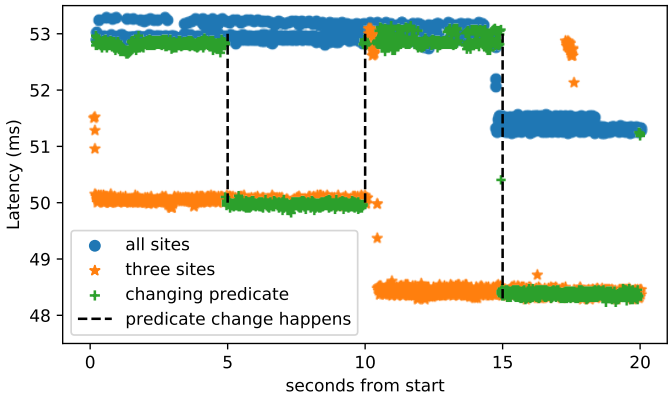


Fig. 8. Latency in predicate dynamic reconfiguration and baselines

Fig. 8 shows that *Stabilizer* can reduce the end-to-end latency from message sending to achieving the requirement of the stability frontier by dynamically shifting the predicate. After the slowest site is moved out of the data synchronization observation list, the latency drops significantly. The latency difference between *all_sites* and *three_sites* is only three milliseconds because the second slowest site (Massachusetts) is only three milliseconds faster than the slowest site (Clemson). It is possible that some stragglers holding a large system slow, where such a benefit could be more obvious. This experiment is just an example of how *Stabilizer* dynamically adjusts the predicate. But *Stabilizer* is not limited by predefined predicates. It can create new predicates according to the application requirements to optimize system performance.

VII. Conclusion

Large scale cloud computing applications depend on geo-replication for safety, reliability, and read-only query performance enhancement. To satisfy various geo-replication consistency requirements, we propose *Stabilizer*, a cloud application library allowing the user-defined consistency model. In *Stabilizer*, we design a new mechanism to let users customize their consistency models with our DSL. We introduce the concept of a user-specified stability frontier, enabling applications to describe their consistency requirements accurately. Because consistency must be detected on a high-rate critical path, we introduce a simple yet expressive DSL to describe new predicates. In the DSL, we support macros, variables, functions and suffixes, making this mechanism flexible and easy to use.

With *Stabilizer*, we extend an existing K/V object store, and then implement a variety of applications over the geo-replicated object store. Besides, a pub/sub service prototype is built to compare with the well-known pub/sub system. In the K/V store experiments, comparisons establish that *Stabilizer* outperforms Paxos-based solutions and that our asynchronous

streaming of control data reduces latency. Moreover, in the pub/sub service prototype experiment, results show that *Stabilizer* can make good use of the bandwidth on WAN and that dynamic reconfiguration can be supported efficiently.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their thoughtful suggestions. This work was supported by the National Natural Science Foundation of China under Grant No.61872397. The contact author is Zhen Xiao.

REFERENCES

[1] C. Li, D. Porto, A. Clement, J. Gehrke, N. M. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, 2012, pp. 265–278.

[2] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, 2008.

[3] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, M. Kaminsky and M. Dahlin, Eds. ACM, 2013, pp. 309–324.

[4] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. T. Morris, "Flexible, wide-area storage for distributed systems with wheelfs," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*. USENIX Association, 2009, pp. 43–58.

[5] A. Khelaifa, S. Benharzallah, L. Kahloul, R. Euler, A. Laouid, and A. Bounceur, "A comparative analysis of adaptive consistency approaches in cloud storage," *Journal of Parallel and Distributed Computing*, vol. 129, pp. 36–49, 2019.

[6] 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*, 2011, pp. 143–157.

[7] S. Jha, "Corrigendum to "derecho: Fast state machine replication for cloud services, " by jha et al., ACM transactions on computer systems (TOCS) volume 36, issue 2, article no. 4," *ACM Trans. Comput. Syst.*, vol. 36, no. 4, p. 15:1, 2020.

[8] (2021) Dropbox. [Online]. Available: https://www.dropbox.com/

[9] 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 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, T. Wobber and P. Druschel, Eds. ACM, 2011, pp. 401–416.

[10] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, "I can't believe it's not causal! scalable causal consistency with no slowdown cascades," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 453–468.

[11] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. USENIX Association, 2012, pp. 251–264.

[12] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.cidrdb.org, 2011, pp. 223–234.

[13] W.-C. Chuang, B. Sang, S. Yoo, R. Gu, M. Kulkarni, and C. Killian, "Eventwave: Programming model and runtime support for tightly-coupled elastic cloud applications," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2523616.2523617

[14] M. A. Sevilla, N. Watkins, I. Jimenez, P. Alvaro, S. Finkelstein, J. LeFevre, and C. Maltzahn, "Malacology: A programmable storage system," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 175–190. [Online]. Available: https://doi.org/10.1145/3064176.3064208

[15] M. David. (2021) Just-in-time compilation (libgccjit.so). [Online]. Available: https://gcc.gnu.org/wiki/JIT

[16] S. Gilbert and N. A. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[17] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. ACM, 1995, pp. 172–183.

[18] P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems," *ACM Comput. Surv.*, vol. 49, no. 1, jun 2016. [Online]. Available: https://doi.org/10.1145/2926965

[19] K. Oh, A. Chandra, and J. Weissman, "Wiera: Towards flexible multi-tiered geo-distributed cloud storage instances," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 165–176.

[20] (2021) etcd. [Online]. Available: https://etcd.io/

[21] S. Ghemawat and J. Dean. (2021) Leveldb, a fast key-value storage library. [Online]. Available: https://github.com/google/leveldb

[22] L. Lamport, "Paxos made simple, fast, and byzantine," in *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002*, ser. Studia Informatica Universalis, A. Bui and H. Fouchal, Eds., vol. 3. Suger, Saint-Denis, rue Catulienne, France, 2002, pp. 7–9.

[23] (2021) Flex scanner. [Online]. Available: https://github.com/westes/flex/

[24] (2021) Bison, a general-purpose parser generator. [Online]. Available: https://www.gnu.org/software/bison/

[25] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–14.

[26] (2021) AWS region and availability zone. [Online]. Available: https://aws.amazon.com/about-aws/global-infrastructure/regions_az

[27] D. K. Gifford, "Weighted voting for replicated data," *Proc.acm Symp.oper.syst.princip.dec*, 1979.

[28] R. Baldoni, M. Contenti, S. T. Piergiovanni, and A. Virgillito, "Modeling publish/subscribe communication systems: towards a formal approach," in *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems, 2003.(WORDS 2003)*. IEEE, 2003, pp. 304–311.

[29] Apache. (2021) Pulsar. [Online]. Available: https://pulsar.apache.org/

[30] (2021) TC, the traffic control in the linux kernel. [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html

[31] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14. [Online]. Available: https://www.flux.utah.edu/paper/duplyakin-atc19

[32] J. Zheng, Q. Lin, J. Xu, C. Wei, C. Zeng, P. Yang, and Y. Zhang, "Paxosstore: High-availability storage made practical in wechat," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1730–1741, 2017.

[33] Z. Li, C. Jin, T. Xu, C. Wilson, Y. Liu, L. Cheng, Y. Liu, Y. Dai, and Z. Zhang, "Towards network-level efficiency for cloud storage services," in *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, C. Williamson, A. Akella, and N. Taft, Eds. ACM, 2014, pp. 115–128.