Blockplane:
A Global-Scale Byzantizing Middleware

Faisal Nawab
Department of Computer Science and Engineering
University of California, Santa Cruz
fnawab@ucsc.edu

Mohammad Sadoghi
Exploratory Systems Lab
Department of Computer Science
University of California, Davis
msadoghi@ucdavis.edu

***Abstract*—The byzantine fault-tolerance model captures a wide-range of failures—common in real-world scenarios—such as ones due to malicious attacks and arbitrary software/hardware errors. We propose Blockplane, a middleware that enables making existing benign systems tolerate byzantine failures. This is done by making the existing system use Blockplane for durability and as a communication infrastructure. Blockplane proposes the following: (1) A middleware and communication infrastructure to make an entire benign protocol byzantine fault-tolerant, (2) A hierarchical locality-aware design to minimize the number of wide-area messages, (3) A separation of fault-tolerance concerns to enable designs with higher performance.**

I. Introduction

A byzantine failure model [11] is a model of arbitrary failures that includes—in addition to crashes—unexpected behavior due to software and hardware malfunctions, malicious breaches, and violation of trust between participants. It is significantly more difficult to develop byzantine fault-tolerant protocols compared to benign (non-byzantine) protocols. This poses a challenge to organizations that want to adopt byzantine fault-tolerant software solutions. This challenge is exacerbated with the need of many applications to be globally distributed. With global distribution, the wide-area latency between participants amplifies the performance overhead of byzantine fault-tolerant protocols.

To overcome the challenges of adopting byzantine fault-tolerant software solutions, we propose pushing down the byzantine fault-tolerance problem to the communication layer rather than the application/storage layer. Our proposal, *Blockplane*, is a communication infrastructure that handles the delivery of messages from one node to another. Blockplane exposes an interface of log-commit, send, and receive operations to be used by nodes to both persist their state and communicate with each other.

Blockplane adopts a locality-aware hierarchical design due to our interest in supporting efficient byzantine fault-tolerance in global-scale environments. Hierarchical designs have recently been shown to perform well in global-scale settings [15]. Blockplane optimizes for the communication latency by performing as much computation as possible locally and only communicate across the wide-area link when necessary.

In the paper, we distinguish between two types of failures. The first is *independent byzantine failures* that are akin to traditional byzantine failures which affect each node independently (the failure of one node does not correlate with the failure of another node). The second type of failures is *benign geo-correlated failures*. In geo-correlated failures, the nodes that are deployed on the same datacenter (or datacenters close to each other) may experience a failure together. In this case, the failure of a node is no longer independent from other nodes. This distinction between the two types of failures is important due to the frequency of datacenter-scale outages [6].

To summarize, the paper proposes the following:

- An approach to make an entire benign protocol tolerate byzantine failures through Blockplane.
- Blockplane proposes a locality-aware design to reduce wide-area communication.
- A separation of fault-tolerance concerns of *byzantine failures* and *benign geo-correlated failures*.

The rest of this paper begins with a background in Section II. We propose Blockplane in Sections III to VII. Section VIII presents the experimental evaluation. Section IX presents an overview of related work. The paper concludes in Section X.

II. Background

Blockplane is a permissioned blockchain system [7], [14] that targets applications where: (1) Participants are globally-distributed, and (2) Byzantine failures need to be tolerated. We distinguish between *byzantine failures* that model independent arbitrary behavior of nodes and *geo-correlated failures* that model an benign outage of a whole datacenter. To clarify the distinction between the two types of failures, we introduce the notations: f_i to represent the number of tolerated independent byzantine failures and f_g to denote the number of tolerated geo-correlated failures. (When the notation f is used without a subscript, then it should be interpreted as the number of tolerated independent byzantine failures f_i.)

Byzantine Agreement Byzantine agreement is the problem of reaching consensus between nodes in the presence of byzantine failures. This includes benign crash failures, hardware/communication malfunctions, software errors, and malicious breaches. The PBFT protocol is a widely-known leader-based byzantine agreement protocol. To tolerate f byzantine failures, $3f+1$ nodes are needed (*i.e.*, $n = 3f+1$). We discuss the normal-case operation of PBFT next, since it is used in the design of Blockplane.

In PBFT, the leader drives the commitment of new commands through three consecutive phases: *pre-prepare, prepare, and commit*. The life-cycle of committing a command begins with a user sending a request to commit a command to the leader. The leader, then, broadcasts a pre-prepare message to

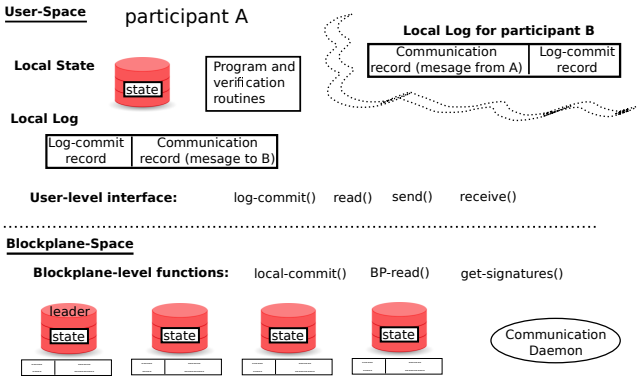


Fig. 1. An illustration of a Blockplane deployment in participant A distinguishing between user-space and Blockplane-space and showing an example of Local Logs in a scenario with two participants. Each log contains log-commit and communication records.

all other nodes. A node that receives a pre-prepare message accepts it if it is authentic the node has not accepted another request with the same sequence number.

If a node accepts a pre-prepare message from the leader, it proceeds to the next phase and the node broadcasts a prepare message to all other nodes, including the leader. Each node waits to collect $2f$ prepare messages in addition to the pre-prepare sent by the leader. Once these messages are received, the node enters the *prepared* phase. The significance of being prepared is that the node now knows that all non-faulty nodes agree on the contents of the message sent by the leader for that view and sequence number. This is because even with f faulty nodes, $f+1$ other nodes have sent prepare messages with these contents—and they intersect with every other group of $f+1$ non-faulty nodes.

Once a node enters the prepared state, it broadcasts a commit message to all other nodes. A prepared node waits for $2f+1$ commit messages (including its own) to enter the *commit* phase. Once in the commit phase, the node—assuming it is not faulty—considers the request committed and logs this information in its local storage. Then, it responds with a reply message to the client. The client waits for identical reply messages from $f+1$ nodes before it considers the command committed (because up to f nodes might be faulty.)

III. Blockplane System And Programming Model

In this section, we present the system and programming models of Blockplane.

A. Motivation

The goal of Blockplane is to provide a framework for efficient and accessible byzantine fault-tolerance in wide-area environments. The system model is hierarchical, where nodes are grouped together to form local units and the units communicate together globally. The aim of this hierarchy is to *mask* byzantine failures locally within the unit. By masking byzantine failures locally, the global coordination can utilize benign (non-byzantine) protocols. The efficiency here is that the global, wide-area communication patterns mimics those of the benign protocol rather than the more expensive byzantine protocol (later in the paper we present an example of applying this to consensus is presented in Section VI-E and correctness is discussed in Section VII.) The programming model aims to provide accessibility to programmers by proposing a new abstraction that—unlike the traditional SMR abstraction—exposes both a commitment and communication interfaces. This allows programmers to follow a design pattern that is closer to the distributed benign protocol rather than adapting the distributed protocol to an SMR-style of programming. Additionally, the separate interfaces for commitment and communication allows the Blockplane infrastructure to handle these two requests differently, leading to a specialized, more efficient design for each.

B. System Model and Notation

The system model consists of n participants. Participants are on different datacenters. We will use the terms *participant*, *datacenter*, and *site* interchangeably. The communication between a pair of participants incurs wide-area communication latency. Participants communicate using a message passing-based distributed protocol, denoted $\mathcal{P}$. All communication is handled by Blockplane. Blockplane establishes and utilizes communication channels between nodes that can incur message drops, corruption and reordering. Blockplane utilizes existing approaches to detect data corruption and reordering such as the TCP protocol.

Each participant maintains $3f+1$ nodes for Blockplane (Each node runs an instance of Blockplane and an instance of $\mathcal{P}$). Launching and managing the Blockplane nodes is performed by the application administrator. For example, if f equals 1, then each participant maintains 4 nodes for Blockplane. The set of Blockplane nodes for participant i are denoted N_i. The set of Blockplane nodes corresponding to a participant, N_i, is called a *Blockplane unit*, or unit for short. The set of nodes and their public keys are known to all nodes.

Each Blockplane unit, N_i, maintains a log of events that represents a SMR log of the corresponding participant. The Local Log of participant i is denoted L_i. The j^{th} event in L_i is denoted $L_i[j]$. There are two types of events in the log:

- *(1) Log-Commit records:* A log-commit record (or commit record for short) represents an event that changes the state of the corresponding participant's protocol, $\mathcal{P}$. A participant writes a commit record via an interface instruction called log-commit that takes an arbitrary string as input. The participant uses log-commit records to persist its state on Blockplane nodes to enable recovery in the case of failure.
- *(2) Communication records:* A communication record represents a message that is sent from the corresponding participant to another participant. The participant writes a communication record by using an interface instruction called send that takes an arbitrary string message and the destination participants. Blockplane also provides a receive interface instruction to receive any incoming messages.

Figure 1 shows an example of a Blockplane deployment. We distinguish between the user-space and the Blockplane-space. The user-space is the abstraction that is seen by the system developer who uses Blockplane. This includes an abstraction of a Local Log and a single copy of an up-to-date state. User-space is where the system developer's code (and verification routines that we will introduce later) reside. The

system code uses Blockplane through the user-level interface functions such as log-commit and other shown interfaces. The Blockplane-space is the underlying infrastructure provided by Blockplane. System developers use the user-level interfaces and are not exposed to Blockplane-space complexities. In Blockplane space there are $3f + 1$ Blockplane nodes, each with a copy of the log and program state. There are various Blockplane-level functions that are not exposed to the system developer. However, we show some of them as they are presented later in the paper as part of Blockplane design. Blockplane-space also includes the communication daemon that is responsible for processing communication records and delivering them to their destinations. The figure also shows an example of the contents of the Local Logs in two participants. Each log consists of log-commit and communication records.

C. Programming Model

Blockplane exposes two types of interfaces: First, an interface for log-commit records that includes a log-commit instruction and a read instruction. This interface is similar to a SMR interface and should be used in the same way SMR interfaces are used. The log-commit instruction guarantees that the committed value will survive the set fault-tolerance level. Also, it guarantees that it is ordered after all previously committed values in the Local Log. The read instruction allows the participant to read the committed records to recover from failures or update replicas. Like systems that use SMR systems in general, a system that uses Blockplane must be deterministic (*i.e.*, an event has a deterministic effect on the state of the system) and all copies of the protocol, $\mathcal{P}$, in the participant's Blockplane unit must start with an identical initial state.

The other type of interface exposed by Blockplane is the *communication interface* that consists of send and receive instructions. The protocol developer of $\mathcal{P}$ must use this interface for any communication between participants.

The protocol $\mathcal{P}$ itself can be written as a benign protocol that does not tolerate byzantine failures. To use Blockplane, the protocol developer must use the commit and communication interfaces:

***Definition** 1:* A protocol that uses Blockplane must use the log-commit and communication interfaces for all the following cases in the protocol:

- If an event changes the state of the protocol, then the event must be committed to the Local Log.
- If an event may lead to one or more communication events, then the event must be committed to the Local Log.
- Any communication between participants must be handled through the communication interface.

Additionally, the developer must write *verification* routines for log-commit and communication instructions. These verification routines are going to be used by the Blockplane replicas to enable them to verify whether a record is a valid state transition. (More details about how verification routines are used by Blockplane is presented in Section IV-B.) The intuition of verification routines is the following: A Blockplane node may propose committing a record to represent a state change in the protocol (*e.g.*, changing the value of a state variable). However, for this record to be committed (as we will see in Section IV-B), other Blockplane nodes must attest to whether the proposed record is a valid state transition given the current state. The verification routines written by the developer will be used for this purpose.

Algorithm 1: An example of a program using Blockplane (the code in red is what is added or modified to use Blockplane)

```
1: c := a counter initially set to 0
2: on event UserRequest (in: destination) {
3:    log-commit(request info)
4:    send (to: destination)
5: }
6: on event StartServer () {
7:    while (true)
8:       receive ()
9:       log-commit(increment-counter)
10:      c++
11: }
```

Summary and discussion. In summary, given this interface of Blockplane, a system developer is not exposed to the complexities of tolerating byzantine failures. Rather, a developer only needs to modify the system to use the log-commit and communication interfaces and implement verification routines. The intention of exposing both a commit and a communication interfaces—unlike SMR that exposes a commit interface only—is two-fold: (1) to make the programming pattern closer to ones used in distributed algorithms which model the communication between different entities as direct or broadcast messages. SMR, on the other hand, would require transforming distributed algorithms that rely on a direct or broadcast message abstraction to utilize an ordered-log coordination abstraction. (2) to enable Blockplane to handle commit and communication requests differently which leads to opportunities to enhance locality as we detail later. These opportunities are more challenging with a SMR abstraction. However, Blockplane's abstractions are new and more complex than traditional SMR (since it requires dealing with two types of interfaces and potentially more complicated verification routines.) These factors will affect the potential for adoption in real scenarios, compared to existing methods that are familiar to developers. Also, Blockplane might not offer an advantage compared to byzantine SMR for protocols that are easily transformed to use SMR-based coordination and for applications where nodes are not separated by wide-area networks. Another obstacle to adopting Blockplane—as well as SMR systems like PBFT—is that the application must be deterministic. Otherwise, an ordered log of events is not guaranteed to produce the same output from a common initial state. The complexity of verification routines depends on the application, for example, a transaction processing application would have verification routines to check whether a transaction can commit. Byzantine SMR systems share this complexity, however, Blockplane verification routines may be more complex due to the need to verify both commit and communication events.

Example. The following is an example of using the Blockplane programming interface (Algorithm 1 shows the modified algorithms in the example.) The example is based on a simple distributed counting protocol, $\mathcal{P}$. In the counting protocol, each

participant maintains a counter that is initially set to 0. A user can trigger an event to send a message from a participant A to another participant B. When a participant receives a message, it increments the counter. In this simple example, the only state that a participant needs to maintain is the counter value. Therefore, the protocol $\mathcal{P}$ calls the log-commit instruction whenever a message is received. This commits the event to the SMR log. In the case of a failure, the protocol $\mathcal{P}$ reads the log using read instructions to recover the state of the counter. The other change that is needed is to use Blockplane's send and receive instructions to send and receive messages.

In addition to using the commit and communication instructions, the system developer must also provide verification routines for each log-commit and send instruction. In the case of the program in Algorithm 1, three verification routines must be written, for the log-commit and send instructions in the UserRequest event and for the log-commit instruction in StartServer. The verification routines are provided to Blockplane in the form of callbacks. (When these callbacks are used is discussed in Section IV-B.) The following is a description of possible verification routines for these three instructions:

- The log-commit instruction in the UserRequest event: the verification routine in this case may verify that the user request is from a trusted user/source.
- The send instruction in the UserRequest event: the verification routine in this case validates that the corresponding user request has been actually received and was not processed before. (This is to avoid the case of a malicious node trying to send messages to other participants without having received a user request to perform that communication.)
- The log-commit instruction in StartServer: the verification routine in this case validates that the corresponding received messages has been actually received from another participant. (We provide more details about verifying received messages later in Section IV-C. The intuition of this verification routine is that the Blockplane node checks whether the received message has been signed by $f+1$ nodes in the source participant.) The purpose of this verification routine is to prevent a malicious node from proposing to commit a record that increments the counter without having received a message from another participant.

We present an example of using Blockplane to byzantize a more complex protocol in Section VI-E.

IV. Blockplane Design for Independent Failures

In this section, we present the design of Blockplane. We focus here on tolerating byzantine failures that affect machines independently. In Section V, we augment this design to tolerate benign geo-correlated failures.

A. Blockplane Operation Overview

Assume that the number of tolerated failures f_i is 1. In this case, each participant maintains 4 Blockplane nodes. In each Blockplane node is a replica of the protocol $\mathcal{P}$ and a copy of the Local Log.

The log-commit instruction takes an arbitrary value as input. The log-commit instruction causes the value to be appended to the Local Log in a fault-tolerant manner. At this point the replicas maintained by Blockplane nodes can read the new committed record and incorporate it to their state.

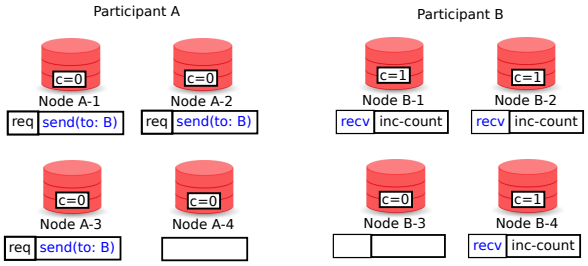


Fig. 2. An example of the state of Blockplane nodes.

Similar to committing an event, the send instruction appends the message and destination to the Local Log in a way that tolerates byzantine failures. After it is appended, $f_i + 1$ Blockplane nodes sign the message denoting that they testify for its correctness. The set of signatures is called a *proof*. Then, the message with its proof are sent to the Blockplane nodes in the destination and it is appended to the destination's Local Log. When the protocol $\mathcal{P}$ at the destination calls the receive instruction it returns the next received unread message in its Local Log.

Figure 2 shows an example of Blockplane state. Consider a Blockplane scenario that is running the counter program in Algorithm 1. The figure shows the state of participants A and B. Each participant has four Blockplane nodes, each labeled X-i, where X is the participant and i is the node's number. The figure shows the state of the node, hence the value of c. Also, it shows the instance of the Local Log. In the log, black entries denote commit records and blue entries denote communication records. The scenario shows the state produced after a request from A is sent to B. Nodes at A have the state of the request and the send event, and nodes at B have the state of receiving the request and processing the counter increment operation.

B. Committing to the Local Log

Committing to the Local Log (called *local commit*) is the most common operation in Blockplane. It is used in three types of events: (1) Logging a state change using log-commit, (2) Declaring an event that may lead to communication with other participants using log-commit, and (3) Communication events using send. Whenever, log-commit or send are called, a commit routine commits the event to the Local Log. We call the instruction to the commit routine local-commit. The instruction takes an arbitrary string as input.

In Blockplane, local commit is performed using the PBFT byzantine fault-tolerant protocol. Specifically, when a commit routine starts, the event value to be committed is sent to the current PBFT leader. Then, the current leader commits the event to the next available log position in the Local Log using the PBFT protocol. (Reads, recovery, and failure cases are handled in the same way as PBFT.) PBFT requires responses from $2f + 1$ nodes to make progress.

In our deployment of PBFT as the component for local commitment, we make some changes to accommodate the Blockplane protocol. The first change is that every value has a type annotation that represents the type of the record. The type of the record can be a commit record or a communication record. The other change is in the voting process in the commit phase. When a node enters the *prepared* phase, rather

Algorithm 2: Blockplane communication algorithms

```
1: on send (in: message) {
2:   local-commit (in: message, communication record annotation)
3: }
4:
5: on start-communication-daemon (in: destination){
6:   p = first entry in the Local Log
7:   loop {
8:     if p is a communication record to destination then
9:       if p was sent to destination then
10:        continue
11:      end if
12:      P = p with a pointer to the previous communication record to destination
13:      Get signatures for the validity of P from f + 1 local nodes
14:      Send P and f + 1 signatures to Blockplane nodes in destination
15:    end if
16:    p++
17:  }
18: }
```

than broadcasting a PBFT commit message immediately, the corresponding verification routine is called. (The verification routine is described in Section III-C.) The verification routine checks the validity of the value.

C. Communication

Send instruction. Algorithm 2 shows the steps in the send instruction and the communication daemon responsible for delivering the message to the other participant. A send instruction takes as input a message (string of bytes) and a destination. When the send instruction is called, the message is committed to the local log, via local-commit.

A *communication daemon* takes care of communicating messages that were committed to the Local Log. Each participant runs a communication daemon that continuously reads the log to detect new send entries. For every other participant, a separate communication daemon handles its communication. Algorithm 2 shows a simplified pseudo code of the operation of a communication daemon. First, the daemon maintains a pointer p, that points initially to the first entry in the Local Log. Then it reads the entries in the Local Log one-by-one until it finds an entry that is a communication record to the daemon's destination. If that communication record has been already sent to the destination, then it is skipped. Otherwise, the communication demon constructs a *transmission record*, P, which consists of the contents of the message in addition to a pointer to the previous communication record to the same destination.

When P is ready, the communication daemon collects $f+1$ signatures from local Blockplane nodes. A Blockplane node signs the transmission record if it agrees that its contents and meta-information are accurate.

At any point in time, it is possible that there are more than one communication daemon. In such a case, the same communication record might be sent more than once to the intended participant. This is, however, is not problematic because the receiving end will verify the validity of the message and that duplicates are dropped. (We present more details about this in the rest of this section in the description of the receive instruction.) In fact, running more than one communication daemons is necessary, because a single communication daemon might be faulty (*e.g.*, malicious) and may pretend maliciously to send messages. Running more than one communication daemon might stress network I/O with a lot of redundancy. For this reason we deploy a *communication daemon reserve*, or reserve for short. A reserve is a collection of $f + 1$ Blockplane nodes. A reserve node periodically send requests to nodes at other participants asking them about the most recent communication record they have received. If there is a substantial gap between the two participants, then the reserve node transforms to a regular communication daemon. This is because a significant gap between the two nodes may signal that the current communication daemon is maliciously delaying the communication of messages.

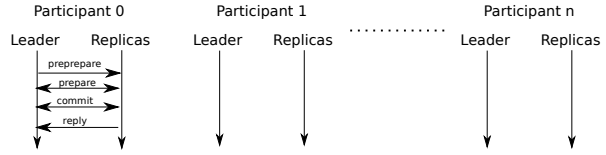
Receive instruction. When a node receives a transmission record P from another participant's communication demon, it tries to commit the received message to its participant's Local Log using log-commit. However, in order to commit the received record, enough nodes in the destination participant must participate in committing it. Blockplane has a special verification routine for received message that is provided by Blockplane. This verification routine, called the *receive verification routine* checks the following:

- The transmission record has $f + 1$ signatures from the source.
- The transmission record has not been received before.
- There are no previous transmission records that were not received. This is checked by verifying that the previous log position—if any—in the received transmission record has been already received.

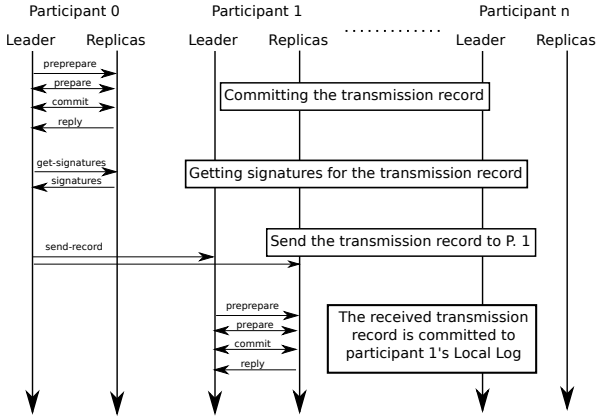
A node receiving the transmission record calls the local-commit instruction to commit it. Blockplane nodes use the receive verification routine to ensure the conditions above apply. After the received message is committed to the Local Log, nodes read the entry and process the received message the same way other records are read from the log.

In Blockplane, receiving messages is implemented by exposing an interface called receive which takes the source participant as input (we omit the input in the discussions and pseudo code if it can be implied.) This is intended to make the communication interface similar to ones typically in use in networked systems. The Blockplane nodes, as they are reading the Local Log and incorporating new events to the node's state, they process received transmission records in the following way: Each node maintains a reception buffer for every other participant. When a received transmission record is encountered in the Local Log, it is appended to the corresponding reception buffer. When a receive instruction is called, the corresponding reception buffer is polled and the next transmission record is returned.

In addition to the transmission records, a Blockplane node may receive requests for information from reserve nodes at other participants. A node respond to these requests with the log position of the most recently committed transmission record received from the participant corresponding to the reserve node. (Note that the returned log position is the one that was sent along with the transmission record and not the one at the receiver's Local Log.) A reserve node must account for the possibility of malicious nodes at the destination participant nodes. Therefore, it should send requests to at least $f + 1$ nodes. The response with the smallest log position is



(a) The communication needed to commit a value using local-commit in participant 0



(b) The communication needed to communicate a message from participant 0 to participant 1 where send and receive instructions are involved

Fig. 3. The normal-case operation of commitment and communication in Blockplane.

guaranteed to be true. This is because at least one out of the $f + 1$ participants is non-faulty. In the worst case, the non-faulty node is the one that responded with the smallest log position. However, it is also possible that a faulty node would respond with a very small log position to make the reserve node suspect that its participant's communication daemon is faulty. To overcome this case, the reserve node can send the request to more than $f+1$ nodes in the destination participant. If any group of $f+1$ nodes agrees that the log position is larger than a certain number, then this can be taken as the current received log positions. The reserve node chooses the set of $f + 1$ nodes that maximizes that lowest log position.

D. Normal-Case Performance

In this section, we present the communication patterns resulting from calling the user-level log-commit instruction and then the communication pattern resulting from calling the user-level send instruction.

Figure 3(a) shows the communication needed to commit a record to the Local Log in participant 0 when the user-level code calls log-commit. (All this communication is performed by Blockplane-level code and is not implemented by the program developer in user-level code.) Committing a record involves the three phases of the PBFT protocol: pre-prepare, prepare, and commit phases. Finally, reply messages are sent from the replicas to the user that called local-commit. Committing a record to the local log takes four intra-datacenter communication latencies.

Figure 3(b) shows the communication involved in sending a message from participant 0 to participant 1. This communication is triggered by a send instruction in user-level code and proceeds in four steps:

- (1) The first step occurs when the send instruction is called. The send instruction commits the message by calling the commit-local instruction. The message is forwarded to the leader (this step is not shown in the figure to account for the typical case where the instructions are called at the leader). The message is then committed to the Local Log of participant 0 in the same way we show in Figure 3(a).
- (2) After the message is committed to the Local Log, it becomes readable to the communication daemon. Assume that the communication daemon is co-located with the current leader. The communication daemon constructs a transmission record P and then collects $f + 1$ signatures for its validity.
- (3) The transmission record and the collected signatures are sent to the destination, participant 1. The transmission record can be sent to one or more nodes in participant 1.
- (4) After receiving the transmission record, a node in participant 1 calls commit-local to commit it to the Local Log. Committing to the Local Log in participant 1 proceeds in the same way we describe in Figure 3(a). The special Blockplane verification routine (the receive verification routine) ensures that the received transmission record is valid.

V. Geo-Correlated Failures

In the previous sections, we showed Blockplane design that tolerates f_i byzantine failures. The type of failures considered in the previous section is the typical byzantine failure that affects nodes independently (*i.e.*, a failure does not affect more than one node). In this section, we introduce the notion of a geo-correlated benign failure, where nodes in the same geographic locality experience a crash failure due to a natural disaster. Remember that we distinguish between the two types of failures: f_i (used interchangeably with f) denotes the number of tolerated independent byzantine failures and f_g denotes the number of tolerated geo-correlated benign failures.

Modeling geo-correlated failures is important because it captures cases where a single failure affects multiple nodes collectively. We focus on geo-correlated failures that are due (non-byzantine) datacenter-scale outages [6].

To overcome geo-correlated failures, Blockplane nodes must coordinate with $f_g + 1$ participants (out of a chosen set of $2f_g + 1$ participants) in order to commit or communicate with other participants. In the case of committing a new value using local-commit, the following changes are introduced: participants maintain mirrors of each others' states on $3f_i + 1$ nodes (these nodes can co-locate with the Blockplane nodes used for local commitment.) If a participant A wants to commit a new value, it must collect proofs from f_g other participants. Obtaining these proofs starts after locally committing the value (in the same manner introduced in the previous section.) The committed value is then sent to f_g other participants. Each participant would locally commit the value on its mirror of participant A. If the value commits, then the participant responds with $f_i + 1$ proofs from its local nodes. These responses are maintained by all nodes in participant A as an annotation of the proved entry.

The same changes are applied to the send instruction. Additionally, the transmission record would also include the proofs from other participants and a node receiving a transmission record would only accept it if the proofs of the source participant and the other f_g participants are valid.

Recovery in the case of a failure is done in a similar way to recovery in primary-copy replication. A secondary that suspects the failure of the current primary, starts processing requests and using other participant as its secondary nodes. (The new secondary nodes must be ones in the originally chosen set of $2f_g+1$ nodes.) This transfer of the primary role does not threaten the integrity of data since any entry must be replicated to f_g+1 nodes prior to commitment, which ensures intersection between any two primaries.

VI. Design Considerations and Examples

A. Read Operations

Committing to the Local Log is performed to persist a state change or communication event. Read operations, on the other hand, do not generally have to be committed or persisted. However, in some instances, an application may want stronger guarantees for its read operations, which may lead to having to commit some read operations. There are different read strategies in Blockplane (Here, we refer to operations that read an entry of the Local Log.) The default read strategy in Blockplane is a read-1 strategy, where the read is served from the closest node to the client. The node provides a proof of the entry's validity which is the set of commit messages corresponding to the entry. Another read strategy is to wait for $2f+1$ identical responses from different nodes. This strategy overcomes the scenario where a malicious node returns that an entry is not committed when in reality it is committed. The last read strategy, and strongest, is a linearizable read that requires committing the read operation to the log in the same way an entry is committed.

B. Recovery from failures

Within a participant replication group, there are two types of nodes that can fail, replica nodes and leader nodes. When a replica node fails, operation is not brought to a halt because enough other replicas are responsive to the leader's requests. When the replica becomes non-faulty again, it reads the state of the Local Log from other nodes to catch up with the current state. A failure of the leader node causes operation to stop until a new leader is elected. In Blockplane, a leader controls the Local Log of a single participant using the PBFT protocol. To elect a leader, we use the same method used in PBFT, which is a view-based leader election. A view is like an epoch, where all nodes start at view 0. Each view has a predetermined leader (*e.g.*, the leader can be the node with id equal to the view number modulo the number of nodes). If nodes suspect that the current view's leader is faulty, then they start moving to the next view. This is repeated until a non-faulty leader is found. (More details can be found in PBFT [4].)

Another type of failure is the failure of a whole participant. If $f_g > 0$, Blockplane recovers from this failure by starting to process the requests for the failed participant at one of the secondary participants. This is akin to how primary-copy replication handles the failure of a primary.

C. Batching and Group Commit

To increase throughput, Blockplane utilizes batching and group commit, which are typical approaches used in many data management and transaction processing systems. Blockplane utilizes batching in a similar manner to SMR-based systems, where transactions (or requests) are batched together. At any given point in time, a leader only attempts to commit a single batch and does not start the next one until the current one is committed. The transactions in a batch are ordered in a way that preserves any dependencies between them, *i.e.*, if a transaction t_1 reads from t_2, then t_1 is ordered before t_2. The leader and replicas perform the validation routines for each transaction and vote positively to commit a batch only if all the transactions are validated successfully. Once the batch is committed, its transactions are applied according to their order in the batch.

D. Performance and Monetary Costs

Byzantizing a system using Blockplane incurs performance and monetary costs due to the additional resources and communication needed to run the protocols. In terms of storage and compute resources, Blockplane requires $3f_i$ additional nodes for each participant. In terms of communication, Blockplane adds the local commitment overhead for all commit and communication requests that incur three phases of communication across f_g+1 participants. Additionally, the application developers need to perform non-trivial effort to transform their application to use Blockplane API and write verification routines. All these overheads—and other smaller ones that are covered in the paper—increase monetary costs as well.

Compared to a SMR approach, many of these overheads exist at varying degrees. For example, byzantine SMR requires a smaller number of nodes and less communication, however it incurs higher latency which may decrease throughput. Given these added costs, Blockplane targets applications that wish to tolerate byzantine failures. This is especially the case for applications that handle finances and mission critical operations, such as e-commerce and banking applications.

E. Paxos Example

In this section, we present how the paxos protocol [10] can be augmented to use Blockplane. We choose paxos for this section due to its popularity and because agreement is one of the problems that has been studied extensively with the byzantine fault-tolerance model, thus allowing us to compare the transformed paxos protocol with existing specialized byzantine agreement protocols. However, Blockplane can be applied to distributed protocols in general, given that they are deterministic and start from the same initial state.

The paxos protocol has two main routines: Leader Election and Replication. In the Leader Election routine, a node attempts to become a leader by getting a majority of votes from other nodes. In the Replication phase, a leader commits a new value by getting a majority of votes accepting it. In the following we show the details of the two routines and how they can be implemented using the Blockplane interface.

Algorithm 3 shows the routines for Leader Election and Replication. The state of the protocol consists of three variables: (1) r, which is the proposal number, initially set to some unique number. (2) l, which is a boolean variable denoting whether the participant is a leader. (3) *max-val*, which is a value used in the paxos protocol as we describe next.

Algorithm 3: Paxos routines using Blockplane (code in red corresponds to Blockplane interfaces).

```
1: r := proposal number, initially set to 𝒰
2: l := am I a leader, initially false
3: max-val := maximum accepted value, initially null
4: on LeaderElection () {
5:   log-commit(Leader Election)
6:   for every m∈M, where M is a at least a majority of participants
7:     send (msg: paxos-prepare(r), to: m)
8:   while responses pending
9:     responses ← receive ()
10:  if responses include a majority of positive votes
11:    l = true
12:    max-val = maximum-accepted-value(responses)
13:    log-commit (l, max-val)
14:  else
15:    r = next unique proposal number
16:    log-commit (r)
17: }
18:
19: on Replication (in: value) {
20:   log-commit(Replication, value)
21:   if l == false
22:     return
23:   for every m∈M, where M is a at least a majority of participants
24:     send (msg: paxos-propose(r,value), to: m)
25:   while responses pending
26:     responses ← receive ()
27:   if responses include a majority of positive votes
28:     log-commit (value committed)
29:   else
30:     r = next unique proposal number
31:     l = false
32:     log-commit (r, l)
33: }
```

The Leader Election routine starts off by committing that the event has started. Then, a paxos prepare messages, denoted paxos-prepare is sent to at least a majority of participants using the send instruction. Responses, in the form of paxos-promise messages, are collected via the receive instruction. Once collected, the node checks whether there is a majority of positive votes. If this is the case, then the node declares that it is the new leader by changing the l variable to true. Also, the received paxos-promise messages may include previous values that were accepted by the participant. If that is the case, the variable max-val is updates with the value with the largest proposal number. (This value must be the one used in a subsequent Replication phase.) Then, the new values of l and max-val are committed using log-commit. If a majority of paxos-promise messages was not attained, then the node updates the proposal number to the next available unique proposal number and commits that change using log-commit.

The Replication routine also starts with committing that the event has started. Then, the node verifies that it is the leader before proceeding. The node then sends a paxos-propose message to at least a majority of nodes with the proposal number and value to be committed. The node receives responses, in the form of paxos-accept messages, using the receive instruction. If a majority of nodes votes positively, the value is considered committed and the node commits that event using log-commit. Otherwise, the node is no longer a leader and updates the proposal number to the next available unique proposal number. It also commits this event using log-commit.

For brevity, we do not show the other algorithms such as the ones to react to receiving paxos-prepare and paxos-propose messages in addition to the verification and recovery code. However, similar changes are applied to them to use Blockplane.

VII. Correctness

In this section, we provide a proof sketch of Blockplane's correctness. (The proof sketches consider the basic design of Blockplane as well as the geo-correlated fault-tolerance extensions.) We discuss three properties in the following lemmas:

Lemma 1: All honest (non-malicious) nodes corresponding to a participant agree on the content of any entry in the Local Log.

Proof sketch. The safety can be proven by contradiction. Assume that two honest nodes, a and b, disagree on the value of an entry in L_A. This means that a and b have each collected a set of $f_i + 1$ reply messages from $f_g + 1$ participants. This means that both a and b have collected a set of $f_i + 1$ reply messages from at least one common participant P (this is because the set of Local Log replication across participants is limited to $2f_g + 1$ participants.) At least one of the reply messages collected by a is from an honest node and at least one of the reply messages collected by b is from an honest node (potentially different from the one collected by a.) Two honest nodes in P cannot disagree about the contents of an entry because all the honest nodes in P followed the PBFT protocol. This disagreement is a contradiction. ■

Lemma 2: An honest node in a participant A only receives a communication message from participant B if honest nodes in participant B agree about its content and order.

Proof sketch. An honest node a runs the receive verification routine. The received message correctness (that it is agreed upon by the honest nodes of the sender participant B) can be proven by contradiction. Assume that the nodes in the sender B do not agree about the content of the sent message. However, the node a has a set of $(f_g + 1)(f_i + 1)$ signatures from B, at least one of which is from an honest node b. Since the honest nodes in B follow the PBFT protocol in committing to their Local Log, then the signature from b is a guarantee that all nodes in B agree on the content of the message. This is a contradiction, and proves the agreement of nodes in B on the content of the message.

The other part we need to prove is that the message is received in order, *i.e.*, there are no earlier messages that were maliciously delayed or dropped. This is guaranteed by tracking the earlier log positions of the sender by the receiver. ■

Lemma 3: A benign protocol that uses Blockplane to commit SMR logs and communicate with others cannot make illegal state transitions (an illegal state transition is an entry in a Local Log that is not the result of correct execution of the protocol in respect to all the previous Local Log entries.)

Proof sketch. The intuition of this proof is that Blockplane masks any byzantine failures locally, and thus allows following the benign protocol's communication pattern globally (such is the case in the presented example in Section VI-E.) This can be proven by contradiction. Assume that an illegal state transition of the benign protocol at a participant A has been made. This illegal state transition is caused by one of two cases:

	C	O	V	I
C	0	19	61	130
O	19	0	79	132
V	61	79	0	70
I	130	132	70	0

TABLE I
THE AVERAGE ROUND-TRIP TIMES IN MILLISECONDS FOR EVERY PAIR OF THE 4 USED DATACENTERS.

- An event that is processed in A: in this case, malicious nodes in A, A_m have caused the illegal state transition. However, for A_m to influence the content of the Local Log, they will have to be more than f_i of them, which is a contradiction.
- A malicious message received from another participant B: The message includes signatures from $f_i + 1$ nodes in B. Since the message is malicious, the number of malicious nodes in B must be higher than f_i, which is a contradiction.

■

In addition to the properties above, Blockplane inherits the availability characteristics of PBFT within a participant (due to the Local Commit protocol) and inherits the availability characteristics of primary-copy replication across participants when $f_g > 0$.

VIII. EXPERIMENTAL EVALUATION

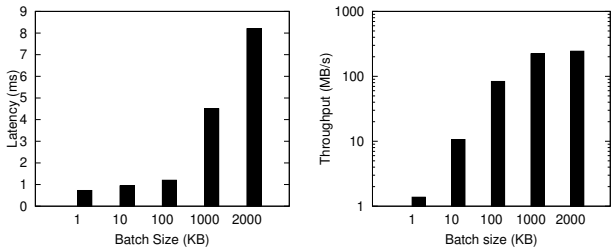
We present a performance evaluation of Blockplane in this section. The evaluation is conducted across four Amazon AWS datacenters in Virginia (V), Oregon (O), California (C), and Ireland (I). The Round-Trip Times (RTTs) between the four datacenters range between 19ms and 132ms (Table I.) In each datacenter we use Amazon EC2 m5.xlarge. Each machine runs Linux and have 4 virtualized CPUs and 16 GB memory. The communication bandwidth between two nodes in the same datacenter—as measured using iperf—is 640 MB/s.

For the evaluation, we compare with (non-byzantine) Paxos [10] and PBFT [4], and a hierarchical variant of PBFT. The prototypes developed for this evaluation aim to highlight the communication overhead effect on the overall normal-case performance. Some design elements that dealt with aspects beyond the evaluation are not implemented, such as recovery, independent code bases via techniques such as n-version programming, and creating and checking signatures and digests. These design aspects either do not play a role in normal-case operation scenarios we evaluate or are negligible compared to the wide-area latency cost and thus their absence is not going to affect the validity of the presented results.

In the evaluations, unless we mention otherwise, we set the fault-tolerance level f_i to 1 and f_g to 0. We also present evaluations with geo-correlated byzantine fault-tolerance where f_g is set to 1. In the Blockplane evaluations, we run 4 machines in each datacenter to represent the nodes of a single organization. In all experiments, we batch commands together to form big batches. Each experiment is the average of committing 1000 batches after a warm-up period of committing 100 batches. The size of a batch is 1000 bytes. (We vary the size of the batches in some of the evaluations.) The contents of each batch is an arbitrary set of commands.

A. Local Commit Performance

In this section, we present a set of experiments to test the performance of local commitment, which is the performance of running the log-commit instruction. The log-commit instruction triggers a three-round byzantine protocol inside the datacenters. Thus, there are no wide-area communication involved in this set of experiments. Figure 4 shows the results of these experiments that were conducted in the datacenter in Virginia. In the experiments, we vary the size of the batches between 1 KB and 2000 KBs. The network bandwidth between two nodes is 640 MB/s. Since we have four nodes in the deployment, a leader must send the payload to at least two other nodes. Therefore, we cannot expect the throughput to be larger than 320 MB/s. Due to other communications (to prepare, commit and apply), the practical throughput becomes even lower.



(a) Latency (b) Throughput
Fig. 4. The performance of local commitment while varying the block size

Figure 4(a) shows the latency of local commitment. With small batch sizes (up to 100 KBs), the latency is within 1 ms. However, larger batch sizes incur larger latency. Committing 1000 KBs batches incurs a latency of 4.5 ms and committing 2000 KBs batches incurs a latency of 8.2 ms. This signals that there is a stress on the system's resources (*e.g.* network I/O) with batch sizes bigger than or equal to 1000 KBs.) This is validated by the throughput results that we show in Figure 4(b). As we increase the batch size, the throughput increases. For small batch size, the increase is more significant. Increasing the batch size from 1 KB to 100 KBs results in a 60x increase in throughput. However, increasing the batch size from 100 KBs to 1000 KBs results in only a 160% increase in throughput. Beyond this point, the increase in throughput is minimal. Increasing the batch size from 1000 KBs to 2000 KBs results in a 10% increase in throughput. This mirrors the effect we observe on latency, where stressing the system resources leads to increasing latency and reaching a throughput plateau.

In summary, with the right choice of batch sizes, Blockplane local commitment can achieve low latencies around 1 ms while reaching a throughput of up to 83 MB/s.

Number of nodes (f_i)	4 (1)	7 (2)	10 (3)	13 (4)
Throughput (MB/s)	83	51	28	25
Latency (ms)	1.2	1.9	3.5	4

TABLE II
LOCAL COMMITMENT PERFORMANCE WHILE VARYING THE NUMBER OF NODES AND f_i.

Another set of experiments we performed measures the scalability of Local Commitment by varying the number of nodes from 4 to 13 nodes (corresponding to f_i values from 1 to 4.) The results are shown in Table II for a batch size 100 KB which yielded the best balance of throughput and latency in the previous experiment. As we increase the number of

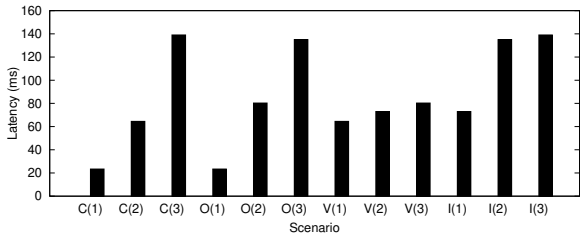


Fig. 5. The performance of committing with geo-correlated byzantine fault-tolerance

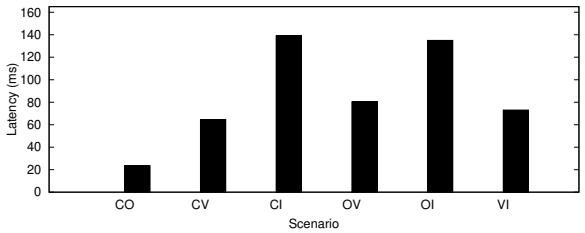


Fig. 6. The performance of communication between participants

nodes (and consequently f_i) the pressure on the network I/O increases because the batches need to be sent to more replicas. This leads to decreasing the throughput of local commitment from 83 KB/s with 4 nodes to 25 MB/s with 13 nodes. The increase in the number of nodes and pressure on the network I/O also leads to increasing the latency of committing a batch from 1.2 ms with 4 nodes to 4 ms with 13 nodes. In summary, increasing the resilience of local commitment incurs a significant overhead and should only be done when necessary.

B. Geo-Correlated Fault-Tolerance

In this section, we test the performance of Blockplane with geo-correlated byzantine fault-tolerance by varying the tolerance level, f_g, from 1 to 3. Figure 5 shows the results of this set of experiments. The x-axis denotes the label of the scenario and the level of geo-correlated byzantine fault-tolerance. For example, the label $C(1)$ denotes the latency of committing at California with the level f_g set to 1. Each datacenter is represented with three data points, one for each f_g level from 1 to 3. Increasing the level of f_g always lead to an increase in latency. This is because increasing the f_g level means that there is a need to coordinate with more datacenters. However, the magnitude of this increase varies from one datacenter to another. The difference in the latency increase magnitude depends on many factors—most important is the wide-area latency between datacenters. For example, increasing the level of f_g from 1 to 2 in California leads to a 176% increase in latency, whereas increasing the level of f_g from 1 to 2 in Virginia leads to only a 13% increase in latency. This difference in magnitude can be inferred from observing the RTT latencies in Table I. California incurs a low latency (19 ms) to communicate with its closest datacenter (Oregon) compared to the latency (61 ms) to communicate with the second closest datacenter (Virginia). On the other hand, the RTT latencies between Virginia and all other datacenters are close to each other (between 61 ms and 79 ms). Likewise, achieving the same level of f_g differs from one datacenter to the other for the same reason. The latency of commitment depends on the RTT latency between the datacenter and the f_g closest datacenters. For the level f_g equals to 1, California and Virginia perform much better than other datacenters. For a level f_g equals to 2, all datacenters achieve a close latency between 64 and 80 ms latency, except Ireland that incurs a 135 ms latency. For a level f_g equals to 3, all datacenters incur a latency over 135 ms, except for Virginia that achieves a latency of 80 ms.

In summary, the effect of the level of geo-correlated byzantine fault-tolerance on performance depends on the topology of the network and the datacenter where the commitment occurs. This means that placement is an important aspect of deployment to achieve a good performance. Also, knowing the sufficient level of fault-tolerance is essential in order to not incur unnecessary latency overhead.

C. Communication Performance

In this section, we perform a set of experiments to test the performance of Blockplane's communication interface (*i.e.*, the send and receive instructions.) Figure 6 shows the results of sending a message from one datacenter to another datacenter through the Blockplane interface. We show six data points, one for each pair of datacenters. The latency for each data point, is the time to send a message through the send interface, then receiving using the receive instruction and finally acknowledging the receipt of the message back at the source. The latency of communication varies across pairs of datacenters. This is because the RTT latencies between pairs of datacenters are different, and the RTT latency between a pair of datacenters directly influence the latency of sending a message through Blockplane. For example, sending a message from California to Oregon requires 23.4 ms. Sending a message between the following pairs of datacenters, California-Virginia, Oregon-Virginia, and Virginia-Ireland, incurs a latency between 64 ms and 80 ms. The highest communication latency (over 135 ms) is incurred between the following pairs of datacenters: California-Ireland and Oregon-Ireland.

An important observation to make is the overhead that Blockplane adds to normal communication (communication without byzantine fault-tolerance.) This overhead is caused by the need to locally commit the communication records at both the source and destination. We expect that the overhead is not going to be significant because the local commit overhead (intra-datacenter latency) is negligible in comparison to the wide-area latency between datacenters. We quantify this overhead by comparing the RTT latencies in Table I with the numbers we obtained in Figure 6. The overhead incurred by Blockplane communication is between 1% and 7% for all pairs of datacenters except for California-Oregon where the overhead is 23%. The reason for this is that the RTT latency between California and Oregon is the lowest (19 ms only). Therefore, the effect of the additional intra-datacenter communication is more significant in comparison to other datacenters.

In summary, the performance of Blockplane communication varies across pairs of datacenters depending on the RTT latency between each pair. Also, Blockplane communication introduces an overhead that is caused by the additional intra-datacenter communication to commit communication records. The overhead is more significant between nearby pairs of datacenters (such as California and Oregon where the overhead is 23%) but is negligible between pairs of datacenters with

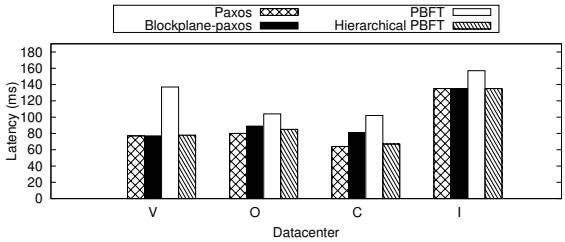
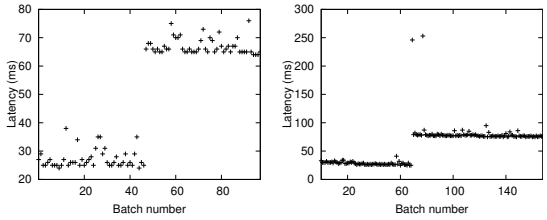


Fig. 7. The performance of Blockplane-paxos in comparison with paxos, PBFT, and Hierarchical PBFT.



(a) Backup failure (b) Primary failure

Fig. 8. The performance of reacting to two types of node failures

higher RTT latency (the overhead can be as low as 1% for some pairs of datacenters.)

D. Performance of a Global Consensus Use Case: Byzantized Paxos

In this section, we evaluate the use of Blockplane to transform a non-byzantine protocol and making it tolerate byzantine failures. Specifically, we take the paxos protocol and augment it with Blockplane commit and communication interfaces. Therefore, we transform a typical deployment of paxos on four datacenters where there is a single machine on each datacenter to a Blockplane deployment on four datacenters where there are four Blockplane machines in each datacenter. In the rest of this section, we present the performance of the byzantized paxos protocol using Blockplane that we call *Blockplane-paxos*. We also compare with three other protocols. The first is paxos that will provide a benchmark to measure the overhead of byzantizing that is incurred in Blockplane-paxos. The second is PBFT that will provide a benchmark to measure the performance of a byzantized paxos with a protocol that is specifically designed to solve byzantine agreement. Finally, we compare with a third variant that we call Hierarchical PBFT that uses PBFT in a hierarchical way similar to how it is done in Blockplane but without the API separation.

Figure 7 shows the results of this set of experiments. Each data point represents the latency of the Replication phase of paxos if the leader is at the corresponding datacenter in the x-axis label. Paxos requires polling the votes of a majority to perform the Replication phase. This means that the expected latency at each datacenter is the RTT latency between that datacenter and the closest majority to that datacenter. The results in the figure confirm this expectation, where the Replication phase latency is within 10% of the RTT latency to the majority as derived from Table I. The performance of Blockplane-paxos should also be close to the latency of a RTT to the majority. This is because—like paxos—the leader needs to hear from the majority. However, in Blockplane-paxos there is additional overhead caused by intra-datacenter communication to locally commit records for commit and communication operations. This overhead varies across datacenters from almost no change in latency (for the case of Ireland) to 33% overhead in the case of California. The Blockplane-paxos overhead in Virginia and Oregon is within 10–13%. The magnitude of the change is affected by the original majority latency. If the majority latency is small, then the effect of intra-datacenter latency would be more significant. Otherwise, the overhead of intra-datacenter communication would be masked by the large wide-area communication latency.

We also compare with PBFT [4]. In this scenario, there are four PBFT node, one at each datacenter. This means that—like Blockplane-paxos in this set of experiments—the number of tolerated byzantine failures is 1. PBFT requires three rounds of communication across three out of the four nodes to reach agreement. However, this does not mean that the latency would simply be three times the RTT to three out of the four nodes. This is because each node broadcasts its messages to all nodes, making the end-to-end latency depend not only on the RTTs from the leader, but also the RTTs between the replicas. In the figure, we show the latency to reach agreement for each datacenter. The PBFT latency varies from 102 ms (for California) to 157 ms (for Ireland). Compared to Blockplane-paxos, PBFT's latency is 16–78% higher than the latency of Blockplane-paxos with the same level of fault-tolerance. The main reason of the improvement in latency that is achieved by Blockplane-paxos is that there is only a single round of communication across datacenters, and the rest of communication is localized within datacenters to mask the effect of byzantine failures. Although Blockplane-paxos achieves a better latency, it requires a larger number of nodes.

The idea of using hierarchy and local-aware computation can be used without the overhead of Blockplane API separation and communication. We quantify the overhead of Blockplane in comparison to a PBFT deployment that uses the same communication patterns of Blockplane-paxos, which is using PBFT locally in a datacenter and using the SMR logs of PBFT to communicate events to commit globally using paxos. We call this *hierarchical PBFT*. Hierarchical PBFT have the similar communication patterns of Blockplane-paxos but without the overhead of API separation and communication. Also, Hierarchical PBFT have the same wide-area communication patterns of paxos but with added local communication. Therefore, we expect the latency of hierarchical PBFT to be between the latencies of paxos and Blockplane-paxos. The figure shows that this is generally the case in our experiments.

Blockplane is intended to be a general framework to byzantize global-scale systems while providing efficient latency characteristics through locality. This set of experiments using the use-case of paxos (and global-scale consensus) shows that using Blockplane may lead to comparable performance to specialized byzantine protocols in global-scale environments. Also, the evaluation showed that the overhead of using Blockplane to byzantize paxos can be modest. However, increasing the level of fault-tolerance may lead to more significant overhead.

E. Reacting to Failure

We evaluate the performance characteristics of how Blockplane reacts to node failures. Figure 8 shows the results of two failure cases of participants. In this set of experiments we

set both f_i and f_g to 1. The first failure case is of a backup participant (Figure 8(a).) In this scenario, the participant in datacenter California is the primary and the other datacenters are backups (each datacenter also has four nodes for local commitment using PBFT.) In the first failure case, the participant in Virginia is committing 100 batches. Up until Batch 45, the backup in Oregon is active, and since it is the closest datacenter to California, it enables committing the batch by one RTT from California to Oregon, which is around 20–40ms. However, we emulate the case of a failure of the backup in Oregon by shutting down the servers. In this case, the primary has to wait to hear from the next closest datacenter (Virginia) which increases the latency to be within 60–80ms. The second case emulates the failure of the primary (California in this case), which is shown in Figure 8(b). In this case, the primary fails after committing 70 batches. This triggers one of the backups to take over and become the new primary. In this case, Virginia becomes the primary and commits batches 71 to 160. The transition leads to increasing the latency since Virginia needs more time to replicate to another backup compared to California. Also, the primary transition can cause some batches to experience higher latency than usual, such as the case in two batches between batches 71 and 80 that incur a latency around 250ms.

IX. Related Work

Byzantine Agreement Byzantine fault-tolerant protocols date back to the early 1980s [11], [17]. A notable milestone is the development of Practical Byzantine Fault-Tolerant (PBFT) [4] that is used by Blockplane to commit requests in the Local Log. There has been a resurgence of byzantine fault-tolerance protocols in the decade following the publication of PBFT [1], [5], [8], [9], [12], [18]. These protocols offer various performance trade-offs in terms of metrics such as the number of communication rounds needed to commit a command and the number of nodes needed to tolerate f failures. For example, Q/U [1] requires the use of $5f + 1$ nodes rather than $3f + 1$ nodes required by PBFT and other protocols [4], [5], [8].

Blockplane differs from earlier byzantine agreement approaches in terms of its *abstraction*, *architecture*, and *algorithms*. In terms of abstraction, Blockplane acts as a middleware with both a commitment and communication interface, unlike other byzantine services that rely on the SMR abstraction of commitment only. Blockplane's abstraction enables more flexibility to the programmer to express—through the communication interface—distributed protocols that rely on coordination. Also, having the communication interface distinct from the commit interface allows Blockplane to optimize the communications performed to minimize wide-area communication in ways that are unattainable if all requests (both commit and communication) are treated similarly. In terms of architecture, Blockplane is different from many byzantine protocols in that it has a hierarchical architecture that groups neighboring nodes together. This allows Blockplane's algorithms to be designed in a way that limits wide-area communication when possible. Steward [2] is a byzantine protocol that enables hierarchical consensus in a similar way to Blockplane. Blockplane differs in that its algorithms proposes a two-dimensional data structure for coordination—rather than the one-dimensional structure in SMR—to leverage this hierarchy.

Database Application Fault-tolerance Middleware. There has been a number of works in developing frameworks and middleware to make database applications fault-tolerance. This includes ones that target tolerating benign failures such as Phoenix Project [3] that uses redo logging to persist the state of applications. More closely to us are frameworks that target tolerating byzantine failures such as Mitra [13] and Fireplug [16]. Unlike Mitra, Blockplane targets wide-area replication scenarios. Unlike Blockplane, Fireplug only considers graph databases.

X. Conclusion

In this paper, we propose Blockplane. Blockplane is a hierarchical middleware solution that is intended to transform systems to make them tolerate byzantine failures. In addition to transforming systems to tolerate byzantine failures, Blockplane aims to also reduce the wide-area communication incurred by the transformed systems in global-scale multi-organization coordination applications. It does so by a hierarchical, locality-aware approach where communication is localized as much as possible. Specifically, each participant is augmented with a number of local Blockplane nodes that perform the durability and communication tasks on behalf of the application. The local nodes run local byzantine fault-tolerance protocols to mask the failures locally. Then, inter-datacenter communication is only performed when necessary for communication or to tolerate datacenter-scale failures.

XI. Acknowledgements

This research is supported in part by the NSF under grant CNS-1815212.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.

[2] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing*, 7(1):80–93, 2010.

[3] R. S. Barga and D. B. Lomet. Phoenix project: Fault-tolerant applications. *SIGMOD Record*, 31(2):94–100, 2002.

[4] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[5] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190. USENIX Association, 2006.

[6] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 1–16. ACM, 2016.

[7] S. Gupta and M. Sadoghi. *Blockchain Transaction Processing*, pages 1–11. Springer International Publishing, Cham, 2018.

[8] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[9] R. Kotla and M. Dahlin. High throughput byzantine fault tolerance. In *Dependable Systems and Networks, 2004 International Conference on*, pages 575–584. IEEE, 2004.

[10] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[11] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[12] J. Li and D. Maziéres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *NSDI*, 2007.

[13] A. F. Luiz, L. C. Lung, and M. Correia. Mitra: Byzantine fault-tolerant middleware for transaction processing on replicated databases. *ACM SIGMOD Record*, 43(1):32–38, 2014.

[14] A. A. Mamun, T. Li, M. Sadoghi, and D. Zhao. In-memory blockchain: Toward efficient and trustworthy data provenance for HPC systems. In *IEEE International Conference on Big Data, Big Data 2018, Seattle, WA, USA, December 10-13, 2018*, pages 3808–3813, 2018.

[15] F. Nawab, D. Agrawal, and A. El Abbadi. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *SIGMOD*, 2018.

[16] R. Neiheiser, D. Presser, L. Rech, M. Bravo, L. Rodrigues, and M. Correia. Fireplug: Flexible and robust n-version geo-replication of graph databases. In *2018 International Conference on Information Networking (ICOIN)*, pages 110–115. IEEE, 2018.

[17] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[18] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. *ACM SIGOPS Operating Systems Review*, 37(5):253–267, 2003.