Gossip Consensus

Daniel Cason
Università della Svizzera italiana (USI)
Lugano, Switzerland

Nenad Milosevic
Università della Svizzera italiana (USI)
Lugano, Switzerland

Zarko Milosevic
Informal Systems
Toronto, Canada

Fernando Pedone
Università della Svizzera italiana (USI)
Lugano, Switzerland

Abstract

Gossip-based consensus protocols have been recently proposed to confront the challenges faced by state machine replication in large geographically distributed systems. It is unclear, however, to which extent consensus and gossip communication fit together. On the one hand, gossip communication has been shown to scale to large settings and efficiently handle participant failures and message losses. On the other hand, gossip may slow down consensus. Moreover, gossip's inherent redundancy may be unnecessary since consensus naturally accounts for participant failures and message losses. This paper investigates the suitability of gossip as a communication building block for consensus. We answer three questions: How much overhead does classic gossip introduce in consensus? Can we design consensus-friendly gossip protocols? Would more efficient gossip protocols still maintain the same reliability properties of classic gossip?

CCS Concepts: • **Computer systems organization → Reliability**; **Availability**; **Redundancy**.

Keywords: Distributed consensus, Fault tolerance, Gossip communication

ACM Reference Format:
Daniel Cason, Nenad Milosevic, Zarko Milosevic, and Fernando Pedone. 2021. Gossip Consensus. In *22nd International Middleware Conference (Middleware '21), December 6–10, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3464298.3493395

1 Introduction

Consensus is a fundamental abstraction, at the core of state machine replication [29, 45]. Although consensus and state machine replication have been extensively studied under a variety of conditions (e.g., synchrony assumptions, failure models, roles assigned to processses), most studies assume that processes can communicate directly with one another. More precisely, the network graph, where vertices represent processes that execute consensus and edges represent the possibility that two processes can communicate directly, is fully connected. A fully connected network graph is a reasonable assumption in systems that run within the same administrative domain (e.g., Google's Spanner [14]).

Middleware '21, December 6–10, 2021, Vitual Event, Canada

ACM ISBN 978-1-4503-8534-3/21/12…$15.00
https://doi.org/10.1145/3464298.3493395

A new breed of decentralized systems, however, notably blockchain systems, is at odds with the assumption of a fully connected network graph. In decentralized systems, no single entity owns the infrastructure; instead, multiple entities from different administrative domains collaborate. In such environments, it is unreasonable to expect that each process of one domain can communicate directly with all processes of another domain. For example, some processes in a domain may hide behind a firewall, instead of connecting directly to processes in other domains.

Reaching consensus in the absence of full connectivity is challenging [2]. Some blockchain consensus protocols face the partially connected-network-graph challenge by relying on gossip communication [4, 10, 11, 46]. With gossip, processes communicate using rounds of message exchanges. In each round, a process can send messages to and receive messages from its neighbors (i.e., processes it is connected to in the graph) [7, 15]. A process communicates indirectly with processes it is not connected to, possibly after several rounds of message exchanges. Since gossip provides high communication reliability and some consensus algorithms can handle message losses (e.g., Paxos [30]), one could naturally lay a consensus protocol on top of a gossip communication protocol (see Figure 1). This gossip-based consensus approach suggests that one does not need to design a consensus protocol from scratch for a partially connected network graph.

At the core of the gossip-based consensus approach lies a fundamental question, and the main driver of the research reported in this paper: *Do consensus protocols and gossip-based communication indeed fit together?* The answer is not obvious: On the one hand, gossip provides consensus with highly reliable communication in a partially connected network. On the other hand, gossip's multiple rounds of message exchanges may slow down consensus. Moreover, gossip's inherent message redundancy may be unnecessary since consensus accounts for process failures and message loss.

This paper investigates gossip-based consensus from a systems perspective. We consider a particular consensus protocol, Paxos [30], and experimentally study its behavior when relying on gossip communication. Our choice of Paxos is justified as follows: (a) Paxos is sufficiently known in the distributed systems community and needs no lengthy explanation; (b) while Paxos is not simple, it is simpler than many other consensus protocols (e.g., [12, 32, 34, 41]); (c) process interactions in Paxos include all communication patterns of interest (i.e., one-to-one, one-to-many, many-to-one, and many-to-many), which renders our study of general interest; and (d) Paxos is a viable option for decentralized systems that tolerate benign failures only (e.g., [4]).

We start by considering the impact of gossip communication on the performance of consensus (i.e., throughput and latency). Unsurprisingly, Paxos atop gossip performs poorly (in terms of

throughput and latency) when compared to a baseline Paxos deployment where processes can communicate directly with the Paxos coordinator. The adoption of gossip causes a latency degradation up to 52% in our experiments, while the maximum achieved throughput can be reduced by up to 74%, depending on the system size. Although this comparison is not fair, as Paxos atop gossip can run in a partially connected network, while baseline Paxos assumes a fully connected network, it provides a reference.

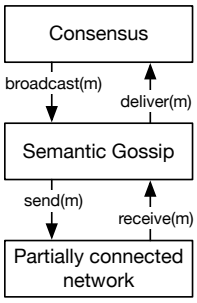

Figure 1. Gossip-based consensus.

We then consider the design and implementation of a "consensus friendly" gossip communication substrate. The idea is to reduce the overhead of gossip by exploiting consensus semantics. We introduce Semantic Gossip, which optimizes classic gossip with two techniques, *semantic filtering* and *semantic aggregation*. Semantic filtering allows the gossip layer to discard messages that have become dispensable, according to the consensus logic. In the case of Paxos, decision messages render voting messages irrelevant. Thus, once a process starts propagating a decision message, it stops the propagation of any voting messages that lead to the decision. Semantic aggregation allows processes to group multiple messages into a single one with equivalent meaning to consensus. In Paxos, multiple voting messages can be grouped into a single multi-process voting message. Notice that while Semantic Gossip uses knowledge about Paxos, it does not require any changes in the Paxos implementation.

Semantic filtering and aggregation substantially reduce the number of messages exchanged by processes to reach consensus. When both techniques are combined, the reduction can be up to 58%, compared to messages exchanged with classic gossip. Moreover, Semantic Gossip boosts the performance of Paxos when compared to implementations based on classic gossip. The adoption of the two semantic techniques improves the latency of gossip-based Paxos by from 7% to 24%, while it is also able to sustain higher workloads than the Paxos implementation based on classic gossip.

The performance improvements brought by Semantic Gossip are welcome as long as they do not come at the expense of the reliability that classic gossip provides. To consider this aspect, we inject failures (i.e., message loss) in the execution of Paxos based on both classic gossip and Semantic Gossip. We found that Semantic Gossip-based Paxos retains the resilience of gossip, up to 20% of injected message loss.

The remainder of the paper is organized as follows. Section 2 defines the system model and introduces background information on gossip and Paxos. Section 3 proposes the design and implementation of Semantic Gossip. Section 4 describes the evaluation of Paxos using point-to-point, classic gossip, and Semantic Gossip communication. Section 5 surveys related work and Section 6 concludes the paper.

2 Background

In this section, we detail our distributed system assumptions, and provide some basic background on gossip communication and the adopted consensus algorithm (Paxos [30]).

2.1 System model

We consider a distributed system composed of a fixed set of processes that communicate through message exchange. We assume, however, that not every pair of processes can communicate directly with each other. Processes that can communicate directly with each other are connected by bi-directional communication channels. The communication between processes that are not directly connected to each other requires the cooperation of intermediate processes to relay messages until they reach their destinations.

The system is partially synchronous [16]: processes and communication channels are asynchronous in general with synchronous periods during which messages are processed and delivered within a bounded but unknown delay. This is a requirement to circumvent the impossibility of fault-tolerant consensus in pure asynchronous systems [20]. We adopt the crash-recovery failure model. Processes can fail by crashing, when they cease to participate in the distributed algorithm without prior notice, and may later recover. Before crashing, and after recovering, processes behave strictly according to the distributed algorithm (i.e., we do not consider Byzantine behavior). Communication channels are unreliable: messages can be dropped, duplicated, reordered, or arbitrarily delayed, but we assume that they cannot be corrupted.

Processes are distributed across data centers in several geographic regions worldwide. Processes within a region are connected via local-area network links, with low latency; processes in different regions communicate via wide-area network links, with higher and more variable latencies. We assume that clients of the consensus protocol know the region that is closest to themselves from a latency perspective and interact with processes located in that region.

2.2 Gossip communication

The gossip communication approach is derived from epidemic dissemination strategies used to propagate information in a distributed system. Originally proposed for the dissemination of updates in replicated databases [15], epidemic algorithms have proven to be an efficient and resilient approach to implementing multicast and broadcast primitives [7]. The operation of epidemic dissemination consists of periodic message-exchange rounds, in which every process randomly selects other processes with which to interact.

There are three general gossip dissemination strategies. In the *push* strategy, every process that has updates (i.e., new messages) to propagate sends them to the selected peers. In the *pull* strategy, processes request updates to the selected peers, which transmit the updates, if they have any, to the requesting processes. These two strategies can be combined into a *push-pull* strategy, in which processes in a round can both send updates to peers and receive updates from them. The *push*, *pull*, and *push-pull* strategies differ in terms of performance, the number of messages exchanged, and

the number of rounds to contact a given portion of the processes with high probability. The best strategy typically depends on the application behavior, the size and frequency of updates, and on the methods used to control the dissemination [15]. In this work we adopt the *push* strategy, however, our contributions could be extended to other strategies.

An algorithm interacts with the gossip communication layer using a *broadcast* primitive that addresses a message to all processes. It is a non-blocking primitive, as the dissemination is asynchronous and may take several rounds. The *deliver* primitive returns messages broadcast by processes. It is a blocking primitive returning messages locally broadcast and messages received from other processes. There are no guarantees that a message broadcast by a non-faulty process is delivered by all non-faulty processes; due to process or link failures, a message may never reach some destinations. In addition, the random choice of peers to which messages are sent may not provide full connectivity. However, a proper choice of parameters provides very high reliability, specially when the *push* dissemination strategy is adopted [7].

2.3 Paxos

Paxos [30] is a distributed consensus algorithm used to implement state machine replication [45]. In short, Paxos allows a set of processes, some of which can fail, to agree on a totally ordered sequence of values. This is achieved by running multiple independent instances of consensus, identified by positive integers, where each sequence decides on a single value. The output of the algorithm consists of the values decided in subsequent instances of consensus, following the total order established by instance identifiers, with no gaps.

Paxos distinguishes among the roles that processes play in the execution of the algorithm: proposers, acceptors, and learners. We assume that each Paxos process plays all these roles. Thus, a process proposes values, works to ensure that a single value is accepted in each instance of consensus, and learns the decided values. Paxos has been optimized in many ways (e.g., [6, 32, 37, 39, 41, 43]). In this paper we adopt the classic version of the algorithm, described in [30].

Each instance of consensus proceeds in rounds, identified by positive integers. Each round is orchestrated by a process, the coordinator. A coordinator can start the same round in multiple instances of consensus. A round consists of two phases, Phase 1 and Phase 2. In each phase, the coordinator sends a message, either a Phase 1a or Phase 2a message, to all processes (one-to-many communication pattern) and waits for Phase 1b or Phase 2b reply messages from a majority of them (many-to-one communication pattern). Messages are tagged with the identifier of the instance they belong to. A process replies to the coordinator of a round provided that it has not replied to messages from higher-numbered rounds in that instance of consensus. In Phase 1, the coordinator of a round tries to find out if a value may have been chosen in lower-numbered rounds. In Phase 2, the coordinator asks the processes to accept a value, either learned from Phase 1b messages or any value proposed by a client.

When a majority of processes accept a value in the Phase 2 of a round, the value of that instance of consensus is decided. Paxos ensures that no other values can be chosen in higher-numbered rounds of that instance of consensus, as at least one process will report the accepted value in Phase 1. When the coordinator learns, from Phase 2b messages, that a value is decided, it informs all processes using a Decision message (one-to-many communication pattern). This communication step becomes redundant if Phase 2b messages are received by all non-faulty processes, not only by the coordinator.

Paxos is safe in the presence of concurrent coordinators, but for the sake of progress a single process is expected to act as the coordinator at a time. Once elected as the coordinator, a process starts a round in multiple instances of consensus at once. In a restricted set of instances processes may have accepted values, forcing the coordinator to re-propose them in Phase 2. But for most instances no process will report having accepted values in previous rounds; in these cases, the coordinator is free to propose any value in Phase 2. Thus, in regular (fail-free) operation, the decision of a value only requires the execution of Phase 2 of a round [30].

3 Semantic Gossip

In this section, we motivate the need for gossip protocols optimized for consensus, describe the design of a gossip protocol that takes advantage of consensus semantics, and detail its implementation.

3.1 Motivation

Implementing consensus on top of gossip communication is straightforward. Essentially, the original communication layer, which assumes a fully connected network graph and provides (one-to-one) *send* and *receive* primitives, is replaced by a gossip communication layer, which provides (many-to-many) *broadcast* and *deliver* primitives (see Figure 2).

In Paxos, Phase 1a and Phase 2a messages, sent by the coordinator to all processes (one-to-many communication pattern) naturally benefit from gossip communication. Instead of sending Phase 1a and Phase 2a messages to all processes the coordinator is directly connected to, it can broadcast the messages via gossip. Eventually, with reasonably high probability, the messages are delivered to all Paxos participants and their propagation cease. Notice that Paxos tolerates message loss, so probabilistic delivery guarantees are sufficient.

Gossip is not well-suited for propagating Phase 1b messages, from all processes to the coordinator (many-to-one communication pattern), as these messages only concern the coordinator, but will be delivered to all participants. Fortunately, Phase 1b messages are rarely sent during regular, fail-free operation; thus, the overhead of propagating them via gossip should not have relevant impact on the overall performance. In the case of Phase 2b messages, the fact that they will be delivered to all processes, not only to the coordinator, can end up being positive. In fact, processes do not need to wait for a Decision message from the coordinator if they receive identical Phase 2b messages from a majority of processes. As a result, the propagation of Phase 2b messages via gossip may actually speed up decisions.

The mismatch between Paxos, and more generally a fault-tolerant consensus protocol, and gossip communication stems from the fact that Paxos was designed to tolerate process crashes and message losses, while gossip protocols strive to provide probabilistic reliable communication. Both consensus and gossip achieve their guarantees by means of communication redundancy. The result is an unnecessarily high number of message exchanges, which penalizes performance. The degree of redundancy increases when using

gossip communication, as processes are likely to receive the same message multiple times, from different peers.

The use of gossip as an underlying means of communication, however, is beneficial for Paxos since gossip does not require direct communication between every pair of processes. This feature naturally extends to environments in which processes are connected to subsets of processes only, and balances the communication load among processes.

3.2 Design

In this section, we discuss simple techniques to address the mismatch between a fault tolerant consensus protocol, using Paxos as reference, and the underlying gossip communication substrate. The goal is essentially to reduce the message redundancy at the gossip layer, employing the knowledge about the message semantics provided by the consensus protocol. The challenge is to achieve this reduction in message redundancy without sacrificing modularity (i.e., without modifying the original Paxos protocol) and the original resilience guarantees offered by gossip.

Semantic filtering. The first technique provides to the consensus protocol the ability to decide whether a message should be sent to a peer. This means interfering with the operation of the gossip layer, which by default forwards every message to all peers. The consensus protocol can then restrain the propagation of messages that are (potentially) no longer useful to a peer, and therefore to all other processes the peer is connected to. The main goal is to save network and processing resources that would be used to forward messages that peers will probably disregard.

Semantic filtering is implemented through a set of rules to identify messages that, according to the consensus semantics, have become obsolete or redundant. For instance, a message from a given round of consensus typically renders any message from previous (smaller) rounds obsolete. Or, for some round steps, acknowledgements from a majority of processes may render further acknowledgements redundant. The semantic filtering rules are evaluated when a message is ready to be sent to a peer. If the message is filtered out, because it is identified as either obsolete or redundant, the gossip layer discards it; otherwise, it is sent as usual.

The evaluation of the semantic filtering rules can be seen as a lightweight execution of the consensus protocol on behalf of a peer. In fact, to identify messages that can be filtered out it is necessary to store some information about messages that were previously sent to that peer. The more comprehensive the rules are, the more information is stored per peer, and the more costly is to evaluate them. Thus, the choice of a set of semantic filtering rules should balance the cost of evaluating them for every message forwarded, with the benefits that an effective filtering can provide.

In the case of Paxos, the proposed semantic filtering rules affect the propagation of Decision and Phase 2b messages. A Decision message is broadcast by the coordinator when it receives Phase 2b messages from a given round and instance from a majority of processes. A Decision message from a given instance thus renders any Phase 2b message from that same instance obsolete. A (regular) process can also learn the value decided in an instance of consensus by receiving identical Phase 2b messages from a majority of processes. From this point on, any further Phase 2b message from the same instance becomes redundant. In both cases, Phase 2b messages are not forwarded to a peer when they refer to an instance for which the peer is expected to already know the decision, from the messages previously sent to it.

Semantic aggregation. The second technique provides the consensus protocol with the possibility to replace a number of similar or related messages, which will be sent to a peer, with a single message comprising the information carried by the original messages. This technique explores the scenario in which the gossip layer has multiple pending messages to send to a peer, so that some of them are likely, according with the consensus semantics, to be aggregated. It is an opportunistic mechanism that aims to reduce the number of messages exchanged by processes via gossip, especially when they operate under moderate to high load.

Semantic aggregation is also implemented through a set of rules that, from a list of pending messages: (i) identify those that are prone to aggregation, and (ii) define how an aggregated message can be built from the original messages. When messages prone to aggregation are found, the first of them in the list of pending messages is replaced by the aggregated message, built according with the respective rule, while the remaining ones are removed from the list. In other words, an aggregated message both replaces and filters out the original messages that it aggregates. Messages that are not prone to aggregation, or for which aggregation is not deemed advantageous by the consensus protocol, are not affected by this technique. They are kept in the list of pending messages, and are forwarded to the peers as usual.

Semantic aggregation rules can be either reversible or not. When a process receives from a peer a message aggregated using a reversible rule, it reconstructs the original messages and treats them as regular messages. That is, messages that are received for the first time are delivered to the consensus protocol and forwarded to other peers—in this process, in particular, they can be semantically aggregated again. When an aggregated message is built from a non-reversible rule, it is treated as a new message broadcast by the process that aggregated it. In this case, the consensus protocol must be able to handle the semantically aggregated message.

Observe that, despite the similarities, semantic aggregation is not the same as batching [21]. When implemented at network level, batching essentially concatenates messages, treated as raw byte arrays, to optimize the network usage. At application level, some message types are batched until the batch size reaches a threshold or a timeout expires. As a result, batching can have negative effect on performance when the system is subject to low loads, as the sending of messages is postponed. This does not happen with semantic aggregation, which despite being ineffective under low loads, does not delay the sending of any messages. Moreover, the technique is more flexible than batching, as messages are not only concatenated, but can be transformed, merged, in any arbitrary way defined by semantic aggregation rules. So, while the size of a batch of messages is proportional to the number of messages in the batch, an aggregated vote message, for instance, has essentially the same size regardless of the number of single vote messages it has replaced.

As for semantic filtering, the best candidates in Paxos for semantic aggregation are Phase 2b messages. When there are multiple identical Phase 2b messages pending to be sent to a process, they can be easily replaced by a single message. For this, a single semantic aggregation rule was adopted. It considers for aggregation Phase 2b messages referring to the same instance and round of consensus; so they only differ by their senders. The aggregated

message consists of any of the original Phase 2b messages plus a field to store the multiple senders. As reconstructing the original Phase 2b is straightforward, the aggregation rule is reversible and no changes in the Paxos protocol were required.

3.3 Implementation

We implemented a gossip-based communication layer to interconnect processes. At the system setup, each process opens connections to a randomly selected set of k processes, where k is a system parameter. Connections are bi-directional, so that the set of peers of a process includes both the k peers to which it opened connections, and a number of peers from which it received connection requests. In fact, the expected number of peers each process interacts with is $2k$.

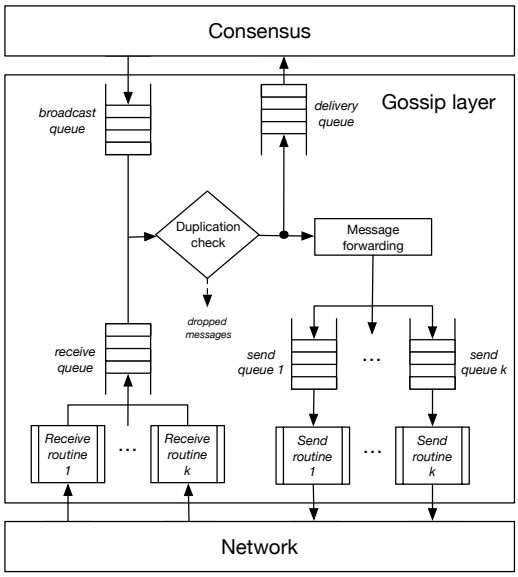

Figure 2. Architecture of the gossip layer at a process.

Classic gossip. Figure 2 illustrates the architecture of the gossip layer. A process interacts with the consensus protocol via two queues. The *broadcast queue* is fed by locally broadcast messages, and the *delivery queue* offers messages to the consensus protocol. A process also maintains, for each peer it is connected to, a Send and a Receive routine. A *send queue* is associated to each Send routine; messages added to a *send queue* are eventually sent to the corresponding peer. There is a single *receive queue* shared by all Receive routines, to which messages received from all active peers are added. A message added to the *broadcast queue* is locally delivered and sent to all peers: it is added to the *delivery queue* and to all active *send queue*s. A message added to the *receive queue* is delivered and forwarded to all peers but the peer the message came from: it is added to the *delivery queue* and to all, but the message's origin, *send queue*s. The selection of peers to which a message is sent is done by the Message forwarding module from the gossip main routine.

Messages are propagated using the *push* disseminating strategy. This means that the same message can be received by a process several times, from distinct peers. We control the flooding of messages using a simple approach based on a cache of *recently seen* messages, maintained by every process. A message is registered to the *recently seen* cache before it is delivered to the consensus protocol and sent to the process' peers. If the same message is received within a short period of time, so that the message's identifier is still on the *recently seen* cache, the message is dropped—i.e., it is not delivered nor forwarded to the peers. This is the role of the Duplication check module represented in Figure 2: it prevents, with some probability, a message from being delivered and forwarded more than once. There is no actual guarantee of a deliver-and-forward once behavior, but the adoption of a reasonable *recently seen* cache size reduces the probability of message duplication. It is worth noting that the *recently seen* cache stores message unique identifiers, that can be defined by the consensus protocol to prevent hash collisions, and not full messages, so its size is constant and relatively small. The same functionality could be obtained adopting other approaches, such as a sliding Bloom filter [42].

Semantic extensions. The gossip layer offers two ways to control its behavior, the techniques presented in Section 3.2: semantic filtering and semantic aggregation. The consensus protocol can adopt one or both techniques by implementing interface methods offered by the gossip layer.

Semantic filtering is provided by allowing the consensus protocol to implement a *validate* method, which receives a message and a destination peer, and returns a boolean:

```
Bool validate(Message, Peer)
```

The *validate* method is invoked by a Send routine when it is ready to send a message to the respective peer. If the method returns false, the message is dropped, as the decision was to filter out the message. Otherwise, the message is sent to the peer, the default behavior when the method is not implemented. Implementations of the *validate* method should be fast and non-blocking, as it is likely to be invoked concurrently by multiple sending routines. The implementation should keep some information about the state of each peer, essentially a summary of relevant messages that were previously processed and not filtered out, and thus sent to that peer. The cost of storing such information versus the benefit in terms of resource saving by filtering out messages that would be sent to a peer should be considered.

Semantic aggregation is provided through the implementation of a pair of methods, *aggregate* and *disaggregate*:

```
Message[] aggregate(Message[], Peer)
Message[] disaggregate(Message)
```

The *aggregate* method receives an array of messages and a destination peer, and returns an array of messages. It is invoked by a Send routine when it has multiple pending messages to be sent to the respective peer. Messages returned by the *aggregate* method, both original and aggregated ones, are sent to the peer, in the order in which they are returned. The *disaggregate* method receives an aggregated message and returns either an array of reconstructed messages, for reversible semantically aggregated messages, or the same message received otherwise. It is invoked by the main gossip routine of a process when a message marked as aggregated is received from a peer. Messages returned by the method are processed as regular messages, in the order in which they are returned: they are checked against the *recently seen* cache and, if not duplicated, delivered and forwarded to peers.

4 Experimental evaluation

In this section, we evaluate the performance of the Paxos consensus protocol under different network assumptions.

4.1 Methodology

We carried out experiments with Paxos using three setups, which differ by the implementation of the communication substrate, while sharing the same Paxos implementation.

The first setup, *Baseline*, provides a reference for the performance of a classic Paxos deployment (with three phases, as presented in Section 2.3) in the experimental environment. In the Baseline setup, the Paxos coordinator communicates directly with every other process, which essentially assumes a fully connected network and violates the multi-administrative environment that we assume in the paper. Nevertheless, comparing to the Baseline setup illustrates the inherent difficulty of designing protocols for multi-administrative environments: performance-wise and under normal conditions, the Baseline setup provides a best case.

In the second setup, *Gossip*, processes can communicate directly only with a small subset of other processes, chosen at random. Communication takes place via gosssip, and messages are disseminated through a randomly generated overlay network. In the third setup, *Semantic Gossip*, processes communicate in the same way as in the Gossip setup, but the gossip layer is augmented with the semantic filtering and aggregation techniques described in Section 3.2.

The evaluation seeks to address three aspects. First, we compare the performance of Paxos (i.e., throughput and latency) in the three considered setups using different system sizes. As we will show, Paxos with Semantic Gossip substantially outperforms Paxos with gossip. We then consider how the two approaches compare with respect to their reliability. Finally, we discuss the results in light of different network topologies.

4.2 System setup

We implemented Paxos, the gossip communication layer, and the Semantic Gossip extensions in Go. We rely on libp2p [1] to establish and maintain communication channels between pairs of processes. Libp2p channels are build atop TCP connections, and provide encryption, multiplexing, flow control, and network-level batching. Although libp2p channels are reliable, our implementation may discard messages when queues connecting different routines are full, as a way to prevent slow processes from blocking the main transport routine. In addition, libp2p connections may be dropped when receivers are much slower than senders; although the dropped connections are reestablished, some messages may be lost. Temporary disconnections between peers, however, do not compromise the network connectivity.

The same Paxos implementation was used for all setups. In the Baseline setup, the elected Paxos coordinator opens libp2p channels to all other processes, which during fail-free operation only interact with the coordinator. In the Gossip and Semantic Gossip setups, each process opens a libp2p channel to a random subset of k processes, so that each process communicates directly with $log_2 n$ other processes on average, where n is the system size. This number of connections per process ensures, with high probability, that the generated network overlay is connected [17]. To provide a fair comparison of results, for each system size n, we enforce the same network overlay in experiments with Gossip and Semantic Gossip setups. In Section 4.6, we then consider multiple randomly generated network overlays and argue that this choice does not affect our conclusions.

We conducted the main experiments in a geographically distributed environment, with processes evenly spread among 13 AWS regions: North Virginia, Canada, Northern California, Oregon, London, Ireland, Frankfurt, São Paulo, Tokyo, Mumbai, Sydney, Seoul, and Singapore. We have placed the Paxos coordinator in North Virginia in all experiments because it is the region that has the lowest latency to and from all other regions. Table 1 lists the WAN latencies between the coordinator's region (North Virginia) and the other twelve regions. Processes were hosted by t2.medium Amazon EC2 instances, with 2 vCPUs and 4GB of RAM.

We conducted an additional set of experiments in a cluster, where we emulated the wide-area latencies between the above mentioned 13 AWS zones. Latencies between cluster nodes were configured using the Linux Traffic Control kernel module [25], that allows postponing the sending of messages to a given destination for a provided delay. The emulated WAN provided an affordable approximation of the AWS environment for experiments requiring hundreds of executions. Those experiments we carried out in a cluster with two groups of machines: (i) Dell PowerEdge 1435 with two Dual-Core AMD Opteron 2GHz and 4GB of RAM, and (ii) HP SE1102 with two Quad-Core Intel Xeon 2.5GHz and 8GB of RAM. By hosting two processes in nodes of group (ii), the performance observed in the emulated environment was comparable with the performance observed in AWS.

Clients generate an experiment workload by proposing values to Paxos. There is one client per region that submits values to a Paxos process hosted in the same region as the client. The communication between clients and Paxos processes is reliable. When a Paxos process receives a value from a client, it forwards the value to the coordinator; the coordinator then proposes the client value in Phase 2 of the next unused Paxos instance. Paxos processes inform all connected clients about Paxos decisions. This happens because clients are state machine replicas and, as so, should execute all commands ordered by the consensus protocol. When a client is notified of the decision of a value it has submitted, it computes the end-to-end latency; throughput is computed as the rate of decisions per time unit. Clients operate in an open-loop model: a client does not wait for the decision of a submitted value before submitting a new one. The rate at which clients submit values to Paxos is an experiment parameter, and all clients submit values at the same rate.

4.3 Overall performance

Figure 3 compares the performance of Paxos in the three setups: Baseline, Gossip, and Semantic Gossip. Experiments were carried out in AWS with distinct numbers of Paxos processes: n = 13, 53, and 105. These system sizes were obtained by placing, respectively, 1, 4, and 8 processes in each of the 13 AWS regions. An additional process, acting as the Paxos coordinator was placed in North Virginia region, so that to achieve n = 53 and 105. In all experiments, the load is generated by 13 clients, one per region, that submit values at a fixed rate. We ran experiments with distinct values sizes, but we only present data for 1KB values, because results with other values sizes presented similar trends. We subjected Paxos to increasing client workloads (submission rates) until we noticed that the protocol is saturated. We highlight the saturation points

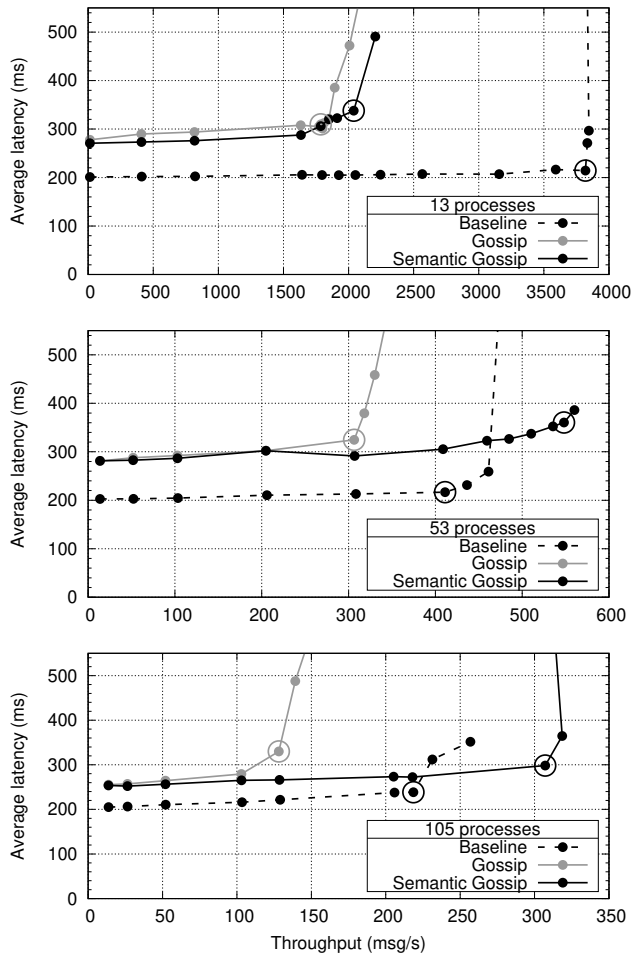

Figure 3. Overall performance of Baseline, Gossip and Semantic Gossip, with varying system sizes and 1KB values.

in the graphs by drawing a circle around them. More precisely, for each setup and system size we highlight the point of the highest ratio between average latency and throughput. From this point on, increasing client workloads results in small throughput increments at the cost of relevant latency increments. Saturation throughputs, normalized for the system size, are summarized in Figure 4.

The first conclusion we can draw from Figure 3 is the relevant overhead derived from the adoption of gossip as a communication means in a partially connected network. In fact, the multiple communication hops required to deliver messages to their destinations via gossip results in a relevant increment in the average latencies to order values, when compared to the Baseline setup, where we artificially assume full connectivity. When considering the lowest workload, the left-most points in the graphs of Figure 3, the average latencies in the Gossip setup are 38%, 39%, and 25% higher than in the Baseline setup for n = 13, 53, and 105. As we increase the workload, the overhead due to the adoption of gossip communication grows, so that in the saturation points of the Gossip setup average latencies are 51%, 52%, and 49% higher than in the Baseline setup, for n = 13, 53, and 105. In addition, we observe that Paxos in the Gossip setup saturates before, i.e., at lower workloads than in the Baseline setup. As a result, throughputs at the saturation points in the Gossip setup are, for n = 13, 53, and 105, respectively, 47%, 74%, and 59% lower than in the Baseline setup.

An explanation for the performance degradation of Paxos in the Gossip setup is the inherent redundancy of gossip communication. We then compared the number of messages received by the Paxos coordinator in Baseline and Gossip setups. In the Baseline setup the coordinator is the only process that communicates directly with all processes, thus the most overloaded process. In the Gossip setup, in terms of communication, the coordinator is a process like any other. With n = 105 processes, the number of messages received by a regular process in the Gossip setup is around 8 times the number of messages received by the coordinator in the Baseline setup. In fact, the gossip layer discards around 87% (about 7/8) of received messages because they are duplicated. For smaller system sizes the redundancy factor observed in the Gossip setup is smaller but still relevant. For n = 53, the redundancy factor is about 5 times and around 80% of received messages are duplicated. For n = 13, the redundancy factor is about 2 times and around 49% of received messages are duplicated. This difference is due to the average number of processes to which each process is connected, of the order of $log_2 n$ ($log_2 105 \approx 6.7$, $log_2 53 \approx 5.7$, and $log_2 13 \approx 3.7$).

A second observation from Figure 3 is the performance improvement obtained with the adoption of the semantic filtering and aggregation techniques. For the smallest system size, n = 13, and workloads below the saturation of Paxos, we observe a discrete but consistent reduction in average latencies in the Semantic Gossip setup when compared with the Gossip setup: from 6% to 7%. Then, around the saturation workload of the Gossip setup, the behavior in the Gossip and Semantic Gossip setups become quite similar, although the saturation throughput in the Semantic Gossip setup is 14% higher than in the Gossip setup. With n = 53, despite some fluctuation in results, we note an overall performance improvement derived from the adoption of the semantic techniques. At the Gossip setup's saturation workload, in particular, the average latency is 11% lower in the Semantic Gossip setup, which also reaches a saturation throughput 79% higher than in the Gossip setup. The improvement is more noticeable for n = 105, where the corresponding reduction in average latency reaches 24% while the increase in the saturation throughput is of 2.4× with Semantic Gossip.

The advantage of Semantic Gossip can be explained when we compare the number of messages exchanged by processes via gossip. With n = 105, considering the saturation point of the Gossip, the number of messages received by a process in the Semantic Gossip setup is 58% lower than in the Gossip setup. For this reduction, of about 2.1×, contribute both the messages discarded through semantic filtering and multiple messages replaced by a single message through semantic aggregation. If we consider the messages delivered to Paxos (when received for the first time and possibly disaggregated), the number is 16% lower in the Semantic Gossip setup, as a direct result of semantic filtering. The portion of messages discarded because they are duplicated is 82% in the Semantic Gossip setup, a small reduction from the 87% observed in the Gossip setup. The inherent redundancy of gossip communication is thus preserved, just as Paxos still operates with a reasonably safe level of redundancy.

Regions	Canada	N.California	Oregon	London	Ireland	Frankfurt	S.Paulo	Tokyo	Mumbai	Sydney	Seoul	Singapore
Latency (ms)	7	30	39	38	33	44	58	73	93	98	87	105

Table 1. WAN latencies between the coordinator's region (North Virginia) and the other twelve regions.

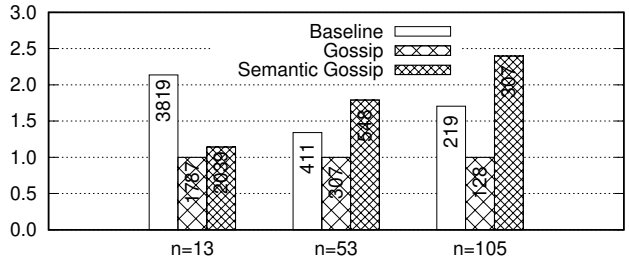

Figure 4. Normalized throughput at saturation point in the three setups. Absolute throughput (messages per second) presented in the bars.

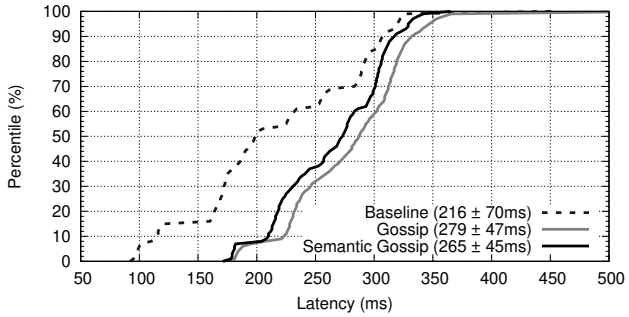

Figure 5. Latency distribution in all setups with $n = 105$. Legend with average latency and standard deviation.

4.4 Latency distributions

Figure 5 presents the cumulative distribution function (CDF) of latencies measured by clients in a given configuration for the three analyzed setups. The data presented refers to experiments with $n = 105$ and the same client workload (104 submissions/s), the bigger workload under which the protocol is not yet saturated in the three setups. Due to space limitations, we don't present data for other system sizes, which under corresponding workloads present a behavior similar to the one observed in Figure 5.

As observed in Figure 5, latency distributions show considerable dispersion and present several noticeable steps. This occurs because latencies are measured by 13 clients, one client per region, that submit a value to a Paxos process located at the same region, then wait until the corresponding decision is informed by the same process. The client located at the same region as the coordinator has the advantage of having its values delivered to the coordinator with low delays. Latencies measured by this client, about 7.7% (1/13) of all, are the lowest, noticeable in the bottom left part of the CDFs. Values submitted by clients located in other regions are forwarded to the coordinator, an operation subjected to WAN latencies. The cost of this operation is more noticeable in curves for the Baseline setup, as Paxos processes are, exceptionally, allowed to send values directly to the coordinator. From the second region (Canada) to the coordinator's region (North Virginia) the latency is relatively small: 7ms. The second step in Baseline's CDF is thus around 15.4% (2/13). Then, up to the seventh region (Frankfurt), WAN latencies are larger but still below 50ms. The step around 53.8% (7/13) represents this interval, corresponding approximately to the median of the latency distributions in the Baseline setup.

Latencies observed by a client are less affected by its geographic location in the Gossip and Semantic Gossip setups than in Baseline setup. As a result, the standard deviation of latencies is lower in the Gossip and Semantic Gossip setups than in the Baseline setup. The least latency variability in gossip-based setups is associated to the adoption of a randomly generated overlay network. While processes located in close geographical regions are not necessarily connected, which increases latency between them, processes farther from the coordinator are not so significantly penalized. In fact, from the 70th-percentile, corresponding to latencies measured by the 4 clients more distant from the coordinator, the overhead imposed by the adoption of gossip communication is much less noticeable (less than 20ms or 6%).

When comparing the Gossip and Semantic Gossip setups, we observe an almost constant distance between the CDFs. Except for the latencies measured by clients connected to the Paxos coordinator, from the 7th to the 97th percentiles latencies measured in the Semantic Gossip setup are from 13ms to 20ms (5.0% to 5.6%) lower than in the Gossip setup. The average latency in the Semantic Gossip setup is 5.4% lower than in the Gossip setup. The improvement in average latencies reaches 24% in the saturation point of the Gossip setup, but we choose to compare the setups under a workload at which none of them is saturated. A less noticeable aspect in Figure 5 is the tail of the latency distributions. The 99.9th latency percentile in the Semantic Gossip setup is 140ms (28%) lower than in the Gossip setup; which, in its turn, is 54ms lower than in the Baseline setup. In addition to the lowest latency standard deviation, this reaffirms the less variable latencies observed in the Semantic Gossip setup.

4.5 Reliability

A major feature of gossip-based communication is its reliability, which allows masking link and process failures. This capability stems from the inherent redundancy of gossip communication, attested by the data collected in our previous experiments. Since a message is transported through multiple distinct paths in the overlay network, a communication disruption between two processes is less likely to prevent the message from being received by all destinations. In this section, we assess the degree of reliability provided by the Gossip and Semantic Gossip setups.

We implemented a fault-injection mechanism that randomly discards messages received by a process. In addition, the timeout-trigged procedures that enable Paxos to react to message loss events were disabled. As a result, Paxos processes may fail to learn the decision for some consensus instances. The impact for the clients is more relevant: a single unsuccessful instance of consensus renders all subsequent instances in the same execution also unsuccessful, as values are delivered in total order, with no gaps. As clients operate

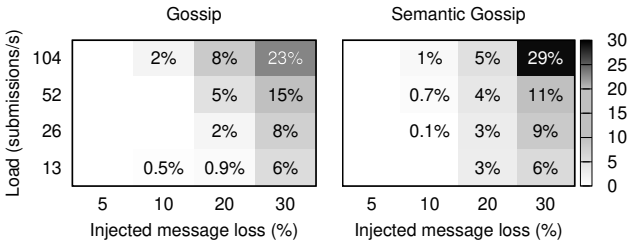

Figure 6. Impact of message loss in the reliability of Paxos in the Gossip and Semantic Gossip setups under injected message loss, as the portion of failed instances of consensus.

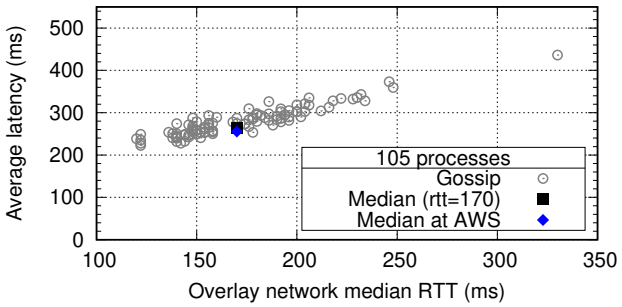

Figure 7. Latency of Paxos in the Gossip setup under low workload in 100 distinct overlay networks. The overlay network adopted in the core experiments is highlighted.

in open-loop, they continue submitting values at a given rate even after failing to order a value. We can then compute the number of values that were submitted by clients but not ordered by Paxos, due to the injected message loss.

Figure 6 summarizes the impact of message loss in the operation of Paxos with $n = 105$ processes in the Gossip and Semantic Gossip setups. We subjected Paxos to increasing workloads, the number of values submitted per second by the 13 clients (y axis), and increasing injected message loss rates (x axis). We ran 10 experiments for each client workload and message loss rate, to minimize the effect particularly favorable or unfavorable executions (as messages are discarded at random). Due to the large number of executions, the experiments were carried out in the emulated AWS environment. Figure 6 depicts the aggregate portion of values submitted but *not ordered* in each configuration. The white cells of the graph represent configurations in which all submitted values were successfully ordered in the 10 executions, despite the injected message loss (i.e., we omit the 0% values).

We can draw two major conclusions from Figure 6. First, the Gossip setup is indeed resilient to message loss: with injected message loss rates below 10% every submitted value is ordered. This means that: (i) every submitted value was received by the coordinator; (ii) at least a majority of processes received the Phase 2a message from the coordinator and accepted the submitted value; and (iii) all 13 processes handling clients received both the Phase 2a message and Phase 2b messages from a majority of processes. With 10% of injected message loss, only 2.5% of the submitted values were not ordered, due to the violation of any of the three conditions above mentioned. Notice that potentially less than 2.5% of the consensus instances have actually failed, but the client did not deliver any values ordered after the first unsuccessful instance of consensus. With 20% of message loss, up to 8% of instances of consensus are affected, while when 30% are discarded up to 23% of the submitted values are not ordered—in both cases under more than 100 submissions/s workload. Second, the benefits on performance obtained with the adoption of the semantic extensions do not come at the cost of lower reliability. In fact, the Semantic Gossip setup has proved to be, in the overall, as reliable as the Gossip setup under message loss rates up to 20%. With 30% of message loss and under higher workloads, however, the portion of values that Paxos failed to order reaches 29% in the Semantic Gossip setup. This indicates that under such (extreme) circumstances the semantic extensions may impact the inherent reliability of gossip communication.

4.6 Network overlays

The overlay network interconnecting the processes, and in particular the latencies between the coordinator and the remaining processes, affects the performance of Paxos. In fact, since the decision of a value requires a round-trip from the coordinator to a majority of processes, the median of RTTs from the coordinator to other processes ultimately dictates the latency of a Paxos instance. Distinct random overlay networks are likely to present different median of RTTs from the coordinator to other processes, and so will have different baseline latencies for deciding values. This is the reason for enforcing the same network overlay in all experiments in the Gossip and Semantic Gossip setups with the same system size, as mentioned in Section 4.2.

Figure 7 illustrates the method to select the overlay network enforced in experiments with the same system size, $n = 105$ in the case. We randomly generated 100 network overlays and submitted them to a minimal client workload in the Gossip setup. For each network overlay we compute the median of RTTs from the coordinator to all processes (x axis), and associate it with the obtained latency (y axis). Notice that multiple overlay networks can have the same median RTT but sport distinct latencies, as the RTT is not the only element to determine latencies. These two parameters allows totally ordering the multiple overlay networks, from which we select the median one. Due to the high number of executions required, we carried out these experiments in the emulated AWS environment. Once the overlay network is selected, we enforce it in AWS and verify whether the performance in the real environment is similar. Figure 7 highlights the selected overlay network, and presents the latency achieved in this overlay network in both emulated and actual AWS EC2 environments.

The adoption of a single network overlay for all core experiments with Gossip and Semantic Gossip setups raises another research question: Are the performance improvements observed with the adoption of the semantic techniques associated to the choice of a particular overlay network? To answer this question we selected a client workload at which the Gossip setup becomes saturated, and adopted this workload to assess the Paxos performance in the Gossip and Semantic Gossip setups in 100 distinct overlay networks. Due to the high number of executions involved, experiments were carried out in the emulated AWS environment.

Figure 8 presents results with $n = 105$ processes in the Gossip and Semantic Gossip setups, adopting the same 100 overlay networks

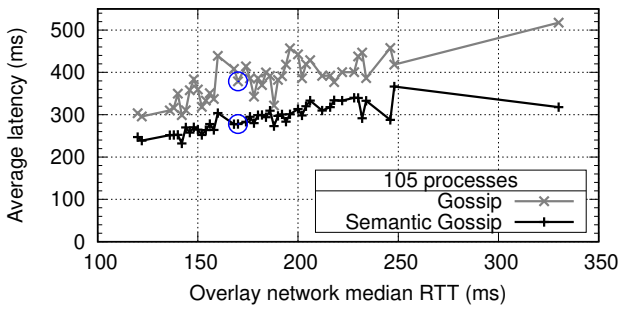

Figure 8. Latency of Paxos in the Gossip and Semantic Gossip setups in 100 distinct overlay networks. The overlay network adopted in the core experiments is highlighted.

illustrated in Figure 7. We aggregate data of overlay networks with the same median RTT (x axis), presenting the average latency (y axis) among multiple experiments, for the sake of readability; Figure 8 therefore present 44 data points for each setup. The workload applied to Paxos in these experiments is enough to evidence the performance improvements derived from the adoption of the semantic techniques. In fact, for all network overlays considered in Figure 8, Semantic Gossip improves latency from 11% to 39%, 23% on average, when compared to the Gossip setup. As a reference, the improvement observed in the network overlay adopted in the core experiment is of 24% from the Semantic Gossip to the Gossip setups.

4.7 Discussion

In this section, we recall the main conclusions from our study and comment on the generality of the proposed approach.

The adoption of gossip communication has a negative impact on the performance of Paxos. While this is an expected result, our study quantifies this overhead: Gossip slows down the decision of values, by increasing by up to 52% the average latencies when compared with the Baseline setup.

Augmenting the gossip-based communication substrate with semantic extensions improves the performance of Paxos in all configurations evaluated: Semantic filtering and aggregation reduce the number of messages exchanged by processes via gossip by up to 58%. As a result, Semantic Gossip provides average latencies from 7% to 24% lower than in the Gossip setup, while sustaining up to 2.4× higher workloads and providing stable and less variable latencies. Moreover, the proposed semantic extensions do not compromise the reliability of gossip communication. For example, without any timeout-trigged retransmission mechanisms, Paxos was able to operate correctly despite up to 10% of message loss, both in the Gossip and Semantic Gossip setups.

We now draw some general considerations from our study. These considerations suggest that other agreement protocols could also benefit from a gossip-based communication substrate with semantic extensions. First, semantic filtering is motivated by the fact that some messages in Paxos supersede previous messages. Whenever this happens, superseded messages can be dropped without negative consequences for the protocol. This is an aspect that is not particular to Paxos, but present in other agreement protocols (e.g., based on rounds). Second, semantic aggregation is inspired by a common pattern in agreement protocols where a protocol step depends on votes cast by processes in previous steps. Instead of sending all votes to all processes, votes can be aggregated and propagated as one message. In Byzantine-fault tolerant agreement protocols, however, semantic aggregation requires special care to avoid new attack vectors. In such cases, an aggregated message should contain signed votes so that a process can ascertain that the votes are valid (although more efficient schemes could also be used to reduce the message size [5]). Finally, since gossip communication offers probabilistic delivery guarantees, agreement protocols that can cope with message loss would be more appropriate to the proposed techniques.

5 Related work

The work presented in this paper is at the intersection of two areas: consensus protocols and gossip protocols. In this section, we briefly review both areas.

5.1 Consensus

Paxos is probably the most known consensus protocol, widely adopted in both academic and industrial setups [6, 13, 28, 32, 38, 39]. While optimal in terms of communication steps and number of tolerated failures [33], Paxos is not easy to understand [9, 43] and implement [13, 28]. Moreover, the distinguished role of the Paxos coordinator makes it the protocol's bottleneck [6, 32, 37], limiting its performance.

Raft [43] is a protocol inspired by Paxos, designed to be easier to understand and implement than Paxos. Raft focuses on replicating a totally-ordered sequence of values, rather than solving single instances of consensus. This lead to important improvements in the leader-replacement mechanism, used in case of (suspicion of) failures. In the absence of failures, however, the operation of Raft and Paxos are identical [24]: the leader broadcasts values, that must be acknowledged by a majority of processes. This makes the semantic extensions proposed for the regular operation of Paxos easily applicable to a gossip-based Raft deployment.

Several Paxos variants address the performance bottlenecks of Paxos. In Mencius [37] processes take turns as the coordinators of successive instances of consensus. While this strategy allows distributing the coordinator load among multiple processes, it does not necessarily improve performance, as it will be ultimately dictated by the slowest coordinator. In S-Paxos [6], the dissemination and ordering of values are detached. Processes disseminate values without the intervention of the coordinator, which proposes value ids in Paxos instances, thus alleviating the coordinator's load. S-Paxos is a good candidate for a gossip-based implementation, where values are inherently disseminated to all processes, while the proposed semantic techniques can be adopted to improve the ordering layer.

Fast Paxos [32] enables any process to propose values directly to all processes, thus bypassing the coordinator. This allows reaching consensus in two communication steps (while Paxos requires three) in instances in which conflicting proposals do not collide. Collisions occur when values are received in distinct orders by processes, which tends to be common with the latency variability of WAN setups, and can be worsened when communication takes place via gossip. Generalized consensus [31] allows processes to deliver some values, considered independent by the application, in distinct

order: the total ordering is relaxed to a partial ordering. The best-known implementation of this approach is EPaxos [41], which allows values to be ordered in two communication steps when no dependent values are concurrently proposed. However, when dependent values are concurrently proposed, EPaxos requires a complex collision-solving procedure, with a communication pattern that is not efficiently implemented atop gossip.

5.2 Gossip

Gossip algorithms were first introduced by Demers et al. [15] to manage replica consistency in the Xerox Clearinghouse Service [44]. The proposed algorithms were specific for the dissemination of database updates, assumed to not be very frequent (a few per second, at most). The adoption of gossip mechanisms as a building block for the dissemination of arbitrary application messages derives from Bimodal Multicast [7]. The algorithm consists of two phases. In the first phase, messages are disseminated in a best-effort fashion through multicast trees, using IP-multicast when available. In the second phase, processes periodically send to a random-selected peer a list of recently received messages, so that to retransmit, on demand, messages that have not yet been received by the peer. Since then, multiple approaches have been proposed to improve throughput and robustness of gossip dissemination [8, 18, 23, 27, 35, 36, 40, 49].

Research in gossip-based broadcast algorithms has focused essentially on two issues. First, the efficient dissemination of messages in large-scale systems through the adoption of overlay networks. Proposed approaches consider building pseudo-random network overlays, by selecting links based on geographic proximity and available bandwidth [27, 40], or topological and connectivity properties [35, 36, 50]. A second research direction addresses the cost/effectiveness of epidemic mechanisms which enable processes to request messages that they failed to receive. The efficiency of these anti-entropy [7, 15] or gossip repair [8, 18, 23, 49] mechanisms is crucial to improve the reliability of gossip dissemination. Efforts have also been made to develop gossip-based services to support large-scale broadcast and multicast algorithms, such as failure detection [48], group membership [22, 26], monitoring and management systems [47].

Semantic Gossip differs from existing approaches because it is designed to support distributed applications that, by themselves, include layers of redundancy. This is the case of Paxos, which includes both typical broadcast steps (to propose values) and the exchange of control messages to ensure agreement, which is a strong form of reliability.

Probabilistic Atomic Broadcast [19] is the algorithm whose behavior most resembles the operation of Paxos atop gossip. The algorithm proceeds in rounds, in each round a process can broadcast a message and should vote for a message, either broadcast or received during the round. Processes periodically exchange the list of messages and associated votes with a random subset of peers. When the number of votes reaches a threshold, all messages in the list are delivered, and the process proceeds to the next round. As in our Paxos deployment, processes send and forward values (broadcast messages) and votes to peers via gossip. Unlike Paxos, the algorithm of [19] only provides probabilistic safety guarantees: two processes may deliver messages in distinct orders, which is equivalent in Paxos to deciding different values in the same consensus instance.

Even though most work on gossip has considered benign failures (e.g., process crashes), recent Byzantine fault tolerant consensus protocols for large-scale environments (e.g., blockchain) have considered the use of gossip as underlying communication substrate. Tendermint is a blockchain middleware based on a BFT consensus algorithm [10] designed for gossip communication. Tendermint has its own gossip layer implementation, that is application-specific and tightly coupled with the consensus implementation. Casper [11], the BFT consensus algorithm proposed to replace the proof-of-work core of the Ethereum blockchain is also designed for a gossip-based environment. HotStuff [51], the BFT consensus protocol at the core of the Libra Blockchain [3], although not designed for gossip-based communication, considers its adoption as the number of processes participating on consensus (validator nodes) grows [46]. The key architectural aspect that distinguishes these proposals from Semantic Gossip is that gossip in blockchain systems is intertwined with consensus logic. Semantic Gossip exploits application (i.e., consensus) semantics without giving up modularity.

6 Conclusions

This paper investigates the deployment of consensus protocols in partially connected networks that rely on gossip communication. We introduce Semantic Gossip, a gossip-based communication substrate that takes consensus semantics into account to optimize performance. Semantic Gossip relies on two techniques, semantic filtering and semantic aggregation. With semantic filtering, the gossip protocol can stop propagating messages that have become redundant from the perspective of the consensus protocol. With semantic aggregation, the gossip protocol can replace multiple consensus protocol messages by a single message of equivalent meaning. Both techniques reduce the number of messages that are propagated by gossip without penalizing the resilience of gossip communication. We have demonstrated the usefulness of Semantic Gossip using Paxos, a well-known consensus protocol. We intend to consider other forms of consensus in the future, in particular consensus protocols that can tolerate Byzantine failures.

Acknowledgments

We wish to thank Patrick Eugster, Pascal Felber, and the anonymous reviewers for the constructive feedback. This work was partially supported by the Swiss National Science Foundation (project number 175717), the Interchain Foundation, and the Hasler Foundation.

References

[1] [n. d.]. Libp2p. https://libp2p.io. [Accessed 2020-05-17].

[2] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. 1999. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science* 220, 1 (June 1999), 3–30. https://doi.org/10.1016/s0304-3975(98)00235-7

[3] Zachary Amsden, Ramnik Arora, Shehar Bano, Mathieu Baudet, et al. 2020. *The Libra Blockchain.* White paper. The Libra Association. https://developers.libra.org/docs/the-libra-blockchain-paper [Accessed 2020-06-01].

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference.* Article 30, 15 pages. https://doi.org/10.1145/3190508.3190538

[5] Paulo S. L. M. Barreto, Hae Y. Kim, Ben Lynn, and Michael Scott. 2002. Efficient Algorithms for Pairing-Based Cryptosystems. In *CRYPTO*, Moti Yung (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 354–369.

[6] Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. 2012. S-Paxos: Offloading the Leader for High Throughput State Machine Replication. In *2012 IEEE 31st Symposium on Reliable Distributed Systems (SRDS'12)*. 111–120. https://doi.org/10.1109/srds.2012.66

[7] Kenneth P. Birman, Mark Hayden, Ozny Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. 1999. Bimodal Multicast. *ACM Transactions on Computer Systems (TOCS)* 17, 2 (May 1999), 41–88. https://doi.org/10.1145/312203.312207

[8] Kenneth P. Birman, Robbert van Renesse, and Werner Vogels. 2001. Spinglass: secure and scalable communication tools for mission-critical computing. In *Proceedings DARPA Information Survivability Conference and Exposition II (DISCEX'01, Vol. 2)*. 85–99. https://doi.org/10.1109/discex.2001.932161

[9] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. 2003. Deconstructing paxos. *ACM SIGACT News* 34, 1 (March 2003), 47–67. https://doi.org/10.1145/637437.637447

[10] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. arXiv:1807.04938 [cs.DC] https://arxiv.org/abs/1807.04938

[11] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. (Oct. 2017). arXiv:1710.09437 [cs.CR] https://arxiv.org/abs/1710.09437

[12] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Margo I. Seltzer and Paul J. Leach (Eds.). USENIX Association, 173–186.

[13] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing (PODC'07)*. ACM Press, 398–407. https://doi.org/10.1145/1281100.1281103

[14] James C. Corbett, Jeffrey Dean, and Michael et al Epstein. 2012. Spanner: Google's globally distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, USA, 251–264.

[15] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, and John Larson. 1987. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing (PODC'87)*. ACM Press, 1–12. https://doi.org/10.1145/41840.41841

[16] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (April 1988), 288–323. https://doi.org/10.1145/42282.42283

[17] P. Erdös and J.W. Kennedy. 1987. k-Connectivity in random Graphs. *European Journal of Combinatorics* 8, 3 (July 1987), 281–286. https://doi.org/10.1016/s0195-6698(87)80032-x

[18] Patrick Th Eugster, Rachid Guerraoui, Sidath B. Handurukande, Petr Kouznetsov, and Anne-Marie Kermarrec. 2003. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS)* 21, 4 (Nov. 2003), 341–374. https://doi.org/10.1145/945506.945507

[19] Pascal Felber and Fernando Pedone. [n. d.]. Probabilistic atomic broadcast. In *Proceeedings of 21st IEEE Symposium on Reliable Distributed Systems, 2002 (SRDS '02)*. 170–179. https://doi.org/10.1109/reldis.2002.1180186

[20] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382. https://doi.org/10.1145/3149.214121

[21] Roy Friedman and R. van Renesse. 1995. *Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols.* Technical Report 94-1527. Cornell University, Dept. of Computer Science. Submitted to IEEE Transactions on Networking.

[22] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. 2003. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Comput.* 52, 2 (Feb. 2003), 139–149. https://doi.org/10.1109/tc.2003.1176982

[23] Indranil Gupta, Kenneth P. Birman, and Robbert van Renesse. 2002. Fighting fire with fire: using randomized gossip to combat stochastic scalability limits. *Quality and Reliability Engineering International* 18, 3 (May 2002), 165–184. https://doi.org/10.1002/qre.473

[24] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. 2015. Raft Refloated. *ACM SIGOPS Operating Systems Review* 49, 1 (Jan. 2015), 12–21. https://doi.org/10.1145/2723872.2723876

[25] Bert Hubert, Gregory Maxwell, Martijn van Oosterhout, Remco van Mook, Paul B. Schroeder, et al. 2002. Linux Advanced Routing & Traffic Control HOWTO. https://lartc.org/lartc.html. [Accessed 2020-05-17].

[26] Håvard Johansen, André Allavena, and Robbert van Renesse. 2006. Fireflies: scalable support for intrusion-tolerant network overlays. *ACM SIGOPS Operating Systems Review* 40, 4 (April 2006), 3–13. https://doi.org/10.1145/1218063.1217937

[27] David Kempe, Jon Kleinberg, and Alan Demers. 2004. Spatial gossip and resource location protocols. *Journal of the ACM (JACM)* 51, 6 (Nov. 2004), 943–967. https://doi.org/10.1145/1039488.1039491

[28] Jonathan Kirsch and Yair Amir. 2008. Paxos for System Builders. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS'08)*. ACM Press, Article 3, 6 pages. https://doi.org/10.1145/1529974.1529979

[29] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[30] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[31] Leslie Lamport. 2005. *Generalized Consensus and Paxos.* Technical Report MSR-TR-2005-33. Microsoft Research. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-33.pdf

[32] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19, 2 (June 2006), 79–103. https://doi.org/10.1007/s00446-006-0005-x

[33] Leslie Lamport. 2006. Lower bounds for asynchronous consensus. *Distributed Computing* 19, 2 (jun 2006), 104–125. https://doi.org/10.1007/s00446-006-0155-x

[34] Butler W. Lampson. 2001. The ABCD's of Paxos. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing (PODC'01)*. 13 pages. https://doi.org/10.1145/383962.383969

[35] Joao Leitao, Jose Pereira, and Luis Rodrigues. 2007. Epidemic Broadcast Trees. In *26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. 301–310. https://doi.org/10.1109/srds.2007.27

[36] Meng-Jang Lin and Keith Marzullo. 1999. Directional Gossip: Gossip in a Wide Area Network. In *Proceedings of Third European Dependable Computing Conference (EDCC-3)*. Springer Berlin Heidelberg, 364–379. https://doi.org/10.1007/3-540-48254-7_25

[37] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, 369–384.

[38] Parisa Jalili Marandi, Samuel Benz, Fernando Pedonea, and Kenneth P. Birman. 2014. The Performance of Paxos in the Cloud. In *IEEE 33rd International Symposium on Reliable Distributed Systems*. 41–50. https://doi.org/10.1109/srds.2014.15

[39] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. 2010. Ring Paxos: A high-throughput atomic broadcast protocol. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '10)*. 527–536. https://doi.org/10.1109/DSN.2010.5544272

[40] Roie Melamed and Idit Keidar. 2004. Araneola: a scalable reliable multicast system for dynamic environments. In *Third IEEE International Symposium on Network Computing and Applications (NCA 2004)*. https://doi.org/10.1109/nca.2004.1347755

[41] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. 358–372. https://doi.org/10.1145/2517349.2517350

[42] Moni Naor and Eylon Yogev. 2013. Sliding Bloom Filters. In *Algorithms and Computation*. Springer Berlin Heidelberg, 513–523. https://doi.org/10.1007/978-3-642-45030-3_48

[43] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, 305–319.

[44] Derek C. Oppen and Yogen K. Dalal. 1983. The clearinghouse: a decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Information Systems (TOIS)* 1, 3 (July 1983), 230–253. https://doi.org/10.1145/357436.357439

[45] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *Comput. Surveys* 22, 4 (Dec. 1990), 299–319. https://doi.org/10.1145/98163.98167

[46] Libra Engineering Team. 2018. Libra: The path forward. Online. https://libra.org/en-US/blog/the-path-forward/ [Accessed 2020-06-01].

[47] Robbert van Renesse, Kenneth Birman, Dan Dumitriu, and Werner Vogels. 2002. Scalable Management and Data Mining Using Astrolabe*. In *Peer-to-Peer Systems*. Springer Berlin Heidelberg, 280–294. https://doi.org/10.1007/3-540-45748-8_27

[48] Robbert van Renesse, Yaron Minsky, and Mark Hayden. 1998. A Gossip-Style Failure Detection Service. In *Middleware'98*. Springer London, 55–70. https://doi.org/10.1007/978-1-4471-1283-9_4

[49] Werner Vogels, Robbert van Renesse, and Ken Birman. 2003. The power of epidemics. *ACM SIGCOMM Computer Communication Review* 33, 1 (Jan. 2003), 131–135. https://doi.org/10.1145/774763.774784

[50] Spyros Voulgaris and Maarten van Steen. 2013. Vicinity: A Pinch of Randomness Brings out the Structure. In *Middleware 2013*. Springer Berlin Heidelberg, 21–40. https://doi.org/10.1007/978-3-642-45065-5_2

[51] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2018. HotStuff: BFT Consensus in the Lens of Blockchain. arXiv:1803.05069 [cs.DC] https://arxiv.org/abs/1803.05069v6