Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

# Boosting Blockchain Consensus with Semantic Gossip

**Daniel Cason[3] | Ricardo Guimaraes[2] | Nenad Milosevic[1] | Patrick Eugster[1] | Zarko Milosevic[3] | Fernando Dotti[2] | Fernando Pedone[1]**

[1]Università della Svizzera italiana, Lugano, Switzerland

[2]School of Technology, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil

[3]Informal Systems, Toronto, Canada

**Correspondence**
<Corresponding author information>

**Present address**
<author present address information>

## Abstract

State Machine Replication has been extensively used to provide highly-available, strongly-consistent systems. Modern systems often require additionally to scale to several nodes. This is the case of several blockchains, that have to run consensus among dozens to hundreds of nodes.

Gossip allows to scale communication to larger sets of nodes and has been recently considered as communication layer to support consensus. By their nature, both consensus protocols and gossip handle message losses and process failures. This double redundancy means that it is not yet clear how efficient is to use gossip as a black building-box for consensus.

In this paper, we investigate these aspects in the context of blockchains. The paper discusses how far redundancy can be eliminated without compromising consensus' safety and liveness. Then it proposes to selectively eliminate redundancy considering the combined use of protocols. To accomplish that, a cross layer mechanism is proposed that, while keeping modularity, uses message semantics from consensus at the gossip level to reduce overhead while forwarding messages. With a prototype implementing the ideas, experiments with 32 and 128 nodes show that the mechanisms proposed respectively enhance throughput 1.24 $\times$ and 3.42$\times$ while slightly reducing latency.

Furthermore, we argue and show results that the proposed redundancy elimination mechanisms do not harm resiliency...

**KEYWORDS**
Consensus, Gossip, BFT, Blockchains

## 1 | INTRODUCTION

Consensus is a fundamental abstraction in many fault-tolerant distributed systems [1, 2]. Although consensus and state machine replication have been extensively studied under various conditions (e.g., synchrony assumptions, failure models), most studies consider consensus deployments with few participants. This is often justified because deployments with more than three or five participants are considered unrealistic. Modern applications, however, challenge this belief by running consensus among many participants. For example, some blockchain systems run consensus among dozens to hundreds of nodes [3, 4, 5].

To accommodate a large number of participants, failures, and geo-distribution, some blockchain consensus protocols rely on gossip communication to handle the interactions of consensus participants [6, 3, 4, 5]. Gossip provides scalability and reliability guarantees by carefully designed rounds of message exchanges, in which processes communicate with subsets of other processes [7, 8]. However, consensus protocols must also handle message losses and process failures due to their fault-tolerant nature. This redundancy suggests that combining consensus and gossip protocols may lead to overhead. In short, it is unclear how efficient it is to use gossip as a black building-box for consensus. Our first research question is thus:

> *Can we identify and eliminate double redundancy situations of gossip and blockchain consensus combined, without sacrificing consensus properties ?*

title footnote

To tackle this problem, we look into consensus and gossip combined. The idea is to reduce the overhead of gossip by exploiting consensus semantics. We identify concrete cases of unneeded redundancy and argue that its elimination neither harms safety nor liveness. Thus, we propose ways to reduce redundancy selectively. The proposed semantic gossip design optimizes gossip with two techniques, *semantic filtering* and *semantic aggregation*. Semantic filtering allows the gossip layer to discard messages that have become redundant or obsolete according to the consensus logic. Semantic aggregation shows that it is possible to aggregate several consensus messages into a single message without losing information. Although we use the Tendermint consensus protocol [9] to discuss ideas and design, the results generalize to Byzantine fault tolerant consensus protocols since the main aspects that enable filtering and aggregation are present in all of them. Having argued for the possibility and soundness of semantic filtering and semantic aggregation, the natural second research question addressed in the paper is:

*What are the performance benefits of augmenting a gossip-based communication substrate with semantic extensions?*

This question is answered experimentally by comparing classical gossip and combinations of filtering and aggregation for a series of workloads and topologies. The proposed design keeps modularity while dealing with cross-layer aspects. In a 128-node network, our experiments show that filtering and aggregation lead to a reduction of 80% in the number of gossip messages exchanged per consensus instance, leading to improved throughput and reduced latency.

The performance advantage of semantic filtering and semantic aggregation is desirable as long as it does not come at the expense of the reliability that classic gossip provides. We consider this aspect in our third research question:

*Do the proposed semantic extensions compromise consensus resilience ?*

To answer this question, we propose two experiments: in the first one we consider that a dishonest node remains silent and does not propose a block in its turn; in the second experiment we consider that a dishonest node omits part of the messages sent. In the first case, we gradually increase the number of dishonest nodes; in the second case we gradually increase message drop. In both cases the effect on baseline and proposed techniques are measured. We found that Semantic Gossip-based Tendermint retains the resilience of gossip, ... *[fd]: ... complete*

The remaining of the paper is organized as follows: Section 2 defines the system model and introduces background information on gossip and Tendermint; Section 3 proposes the design and implementation of Semantic Gossip in the context of blockchains, focussing the Tendermint protocol. The semantic gossip mechanisms for filtering and aggregation are proposed, discussed to preserve consensus properties, and their design/implementation presented; Section 4 describes the evaluation of Tendermint using gossip, and Semantic Gossip communication. The evaluation starts with a topologycal study to generalize observations, being follwed by throughput and latency assessment using different combinations of semantic gossip mechanisms. The last part brings an assessement of the resilience of Tendermint without and with Semantic Gossip, varying its mechanisms, in the presence of increasing number of byzantine nodes; Section 5 surveys related work and Section 6 concludes the paper.

## 2 | BACKGROUND

This section introducess the system model and assumptions, provides the needed background on gossip communication, and presents the Tendermint blockchain consensus protocol.

### 2.1 | System model and assumptions

We consider a distributed system with an unbounded set of client processes $\mathcal{C} = \{c_1, c_2, ...\}$ and a bounded set of server processes $\mathcal{N} = \{n_1, ..., n_n\}$ we call nodes. Processes communicate by exchanging messages and do not have access to a shared memory or a global clock. Processes can be *correct* or *faulty*. A correct or honest process follows its specification whilst a faulty process can present arbitrary (i.e., Byzantine) behavior.

The system is partially synchronous: it is initially asynchronous and eventually becomes synchronous. There are no processing and communication bounds when the system is asynchronous but these bounds exist when the system is synchronous. The time when the system becomes synchronous, the Global Stabilization Time (GST), is unknown to processes. We assume reliable links, i.e. if an honest process sends a message to another honest process, the second eventually receives it.

We use cryptographic techniques for authentication and digest calculation. We assume that adversaries (and Byzantine processes under their control) are computationally bound so that they are unable, with very high probability, to subvert the cryptographic techniques used. Adversaries can coordinate Byzantine processes but cannot delay correct processes.

*[fp]: There are different terms used for processes in the paper, such as peers, validators, nodes, ... This needs to be consistent throughput the paper.*

*[fd]: as above processes are clients and servers - servers are called nodes in the paper.*

## 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 [7], epidemic algorithms have been proven an efficient and resilient approach to implementing multicast and broadcast primitives [8]. The operation of epidemic dissemination consists of periodic message-exchange rounds, in which every noderandomly selects other nodeswith which to interact.

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

Gossip communication is analogous to an epidemic, where a virus plays the role of a piece of information (e.g., a message), and infection plays the role of learning about the information. When a node broadcasts a message or receives a message for the first time, it becomes infected. An infected node propagates the message to a number of nodes before it is removed from the propagation cycle. If nodes propagate a message to a large enough number of peers nodes before being removed, then with high probability all nodes should be infected, i.e., they eventually deliver the broadcast message.

The above description refers to the broadcast of a single message and applies to the different epidemic dissemination strategies. The *push*, *pull*, and *push-pull* strategies differ in terms of performance, the number of messages exchanged, and the number of rounds to infect a given portion of the nodes 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 [7]. 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 nodes. It is a non-blocking primitive, as the dissemination is asynchronous and may take several rounds. The *deliver* primitive returns messages broadcast by nodes. It is a blocking primitive returning messages locally broadcast and messages received from other nodes. There are no guarantees that a message broadcast by a non-faulty node is delivered by all non-faulty nodes; due to other nodes' 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 [8].

## 2.3 | Tendermint

Tendermint [9] powers Cosmos, a network of proof-of-stake blockchains. Both Tendermint and Cosmos are mature technologies, used by over a hundred businesses and running on hundreds of computing nodes. Tendermint builds an overlay network: a node communicates directly with a restricted subset of nodes, the node's neighbors or peers. To send a message to nodes that are not its peers, a node relies on gossip communication.

A node is expected to maintain a long-term persistent identity in the form of a public key, from which the node's unique ID is derived. When attempting to connect to a peer node, the first verifies whether the peer node is in possession of the private key corresponding to its ID, thus preventing man-in-the-middle attacks. We assume that each node is connected to at least $f+1$ peer nodes, ensuring that at least one of the connected nodes is honest.

Tendermint runs a sequence of consensus instances. An instance is also called a height and decides one block of transactions. To decide a height, one to multiple attempts, called rounds, can be needed. Nodes play the role of *proposers* in the first round of successive heights of consensus according to a function *proposer(height, round)* known by all nodes. To handle asynchrony

and failures, a new round in the same height can be initiated and led by a different node. Nodes also have voting power to accept (or not) blocks being proposed, acting as *validators*.

Similar to PBFT's execution [10], composed of three steps, a failure-free round in Tendermint has the steps *propose, prevote* and *precommit*, briefly explained below for node *n*.

*[fp]: The following fragment needs editing, as it doesn't naturally follow the text before. It'd be good if Daniel could double check this. [fd]: reviewed this.*

**init** *height=1, round=1*;
**upon** *n == proposer(height,round)*:
broadcast PROPOSAL with a signed block of transactions and identification *id*;
**upon** receiving PROPOSAL from the current height and round's proposer:
if the block can be accepted, sign and broadcast PREVOTE.
Acceptance depends on further consensus rules and validation by the application;
**upon** receiving PROPOSAL and matching PREVOTE messages from a quorum for the block *id*:
broadcast a signed PRECOMMIT message for the block *id*;
**upon** receiving PRECOMMIT for the same block *id* from a quorum:
commit the block and append to the blockchain.
Deliver block's transactions to application and proceed to the next height, 1st round.

Coherent with the proof-of-stake approach [11], nodes may have distinct voting power, that is, the vote of one node may have more weight than another. A quorum is built by a subset of nodes aggregating more than 2/3 of the total voting power.

The straightforward description above is then enriched with additional rules to define whether a proposer must re-propose a block accepted in a previous round, and whether a node should accept the block proposed in a round. The full algorithm can be found in [9].

The set of nodes is defined per consensus height. The initial set is defined at the genesis state. Then, modifications can be carried out with application commands. The new set of nodes is proposed as an instance and is adopted if accepted.

# 3 | SEMANTIC GOSSIP

In this section, we first motivate investigating the interplay between consensus protocols and gossip communication. We then propose semantic gossip mechanisms, and elaborate on design details.

## 3.1 | Motivation

Using gossip as a communication layer for consensus is in general a straigthforward adaptation: the direct point-to-point communication using a set of channels providing full connectivity is replaced by gossip a layer that reaches all processes. One of the reasons it is straightforward is because the consensus layer deals with process failures and communication problems that may arise, which makes it robust to be used with different underlying communication properties.

In the following we discuss two fundamental characteristics of a wide range of consensus protocols that motivate the investigation of the interplay of consensus with a gossip communication layer.

### 3.1.1 | Detecting quorums

One way to design redundant systems is through the concept of quorums. A quourum can be understood as a level of redundancy sufficient to allow the protocol to progress while keeping safety, under the fault model assumption. Considering this, a node could spare to send messages if it already knows a quorum has been achieved by other nodes. While this observation does not help when nodes communicate directly, the use of gossip at the communication level allows to benefit from it. If a node receives a message *m* and computes that it already has matching messages from a quorum of nodes, it assumes the previous messages, sufficient to build a quorum, were forwarded. Thus the redundant message *m* (and further ones) is not needed at the neighbour

nodes and is not forwarded. This redundancy elimination is however only possible if the gossip layer is aware of the consensus protocol semantics.

### 3.1.2 | Phased communication

Consensus protocols are typically organized in communication instances, and proceed in rounds using corresponding message types. This helps to detect if a given message is obsolete for the current progress of the protocol. In such case it could be ignored. A further observation is that, as nodes behave simmetrically, it is common that the same types of messages, stemming from different nodes are communicated concurrently through the network. While using direct communication this maybe a pointless observation, when using gossip we can unfold the examination: messages with the same semantics, but differing in the specific node's answers (e.g. votes), will coexist in intermediate nodes along the gossip forwarding. Moreover, it is typical that such messages are addressed to the same set of nodes (all that participate in the consensus protocol). In such cases, instead of forwarding and processing separately each such message, they can be aggregated at an intermediate node, resulting one message collecting the space of parameters of the equivalent bunch of messages. This leads to sparing both messages through the network as also message processing at destinations.

## 3.2 | Design

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

### 3.2.1 | Semantic filtering

The first technique allows the consensus algorithm to decide whether a message should be sent to other nodes at the gossip layer. The consensus algorithm can then restrain the propagation of messages that are (potentially) no longer useful to other nodes. Semantic filtering is implemented through a set of rules to identify messages that, according to the consensus semantics, have become redundant or obsolete, as discussed in Sections 3.1.1 and 3.1.2. The semantic filtering rules are evaluated when a message is ready to be sent to peer nodes. 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 algorithm on behalf of another node. In fact, to identify messages that can be filtered out it is necessary to store some information about messages that were previously sent to other nodes. The more comprehensive the rules are, the more information is stored per peer node, and the more costly it 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.

#### 3.2.1.1 | *Semantic filtering in Tendermint*

As discussed in Section 2.3, a node progresses to the next step when a quorum of PREVOTE or PRECOMMIT matching messages is received. Additional messages of these types are not needed. Also, duplicated messages are not useful and need not to be forwarded. The filtering rules thus state that a vote message is dropped by a node in the following cases:

- If a vote of the same type (either PREVOTE or PRECOMMIT), with the same originator, height, round, and value has been already computed before by the node;
- If the vote (either PREVOTE or PRECOMMIT) is for a height and round for which the node has already handled a quorum of matching votes.

#### 3.2.1.2 | *Semantic filtering keeps consensus properties*

Discarding messages at the gossip level is perceived as message loss (absence of) at the consensus level. Thus, filtering could at most prevent progress but not safety. To ensure progress, a filtering mechanism cannot hinder nodes from receiving messages needed to build a quorum. Notice that with the first filtering rule the message dropped has been handled before and with the second filtering rule, a node discards messages after having handled a quorum of matching messages. In both cases, since the node handled the previous messages, they were already forwarded to other nodes.

*[fp]: The text here is unclear, as the message already could be lost.*

*[fp]: What does it mean for "a quoru to be identified at a node"?*

### 3.2.2 | Semantic aggregation

The second technique provides the consensus algorithm with the possibility to replace a number of similar or related messages, which will be forwarded to other nodes, with a single message comprising the information carried by the original replaced messages. This technique explores the scenario in which the gossip layer has multiple pending messages to send to a peer node, 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 nodes 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 to 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 algorithm, are not affected by this technique. They are kept in the list of pending messages, and are forwarded to peer nodes as usual.

Semantic aggregation rules can be either reversible or not. When a node receives from a peer node a message aggregated using a reversible rule, it reconstructs the original messages and treats them as regular messages. That is, messages received for the first time are delivered to the consensus algorithm 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 algorithm must be able to handle the semantically aggregated message.

Observe that, despite the similarities, semantic aggregation is not the same as batching [12]. 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.

#### 3.2.2.1 | *Semantic aggregation in Tendermint*

Regarding Tendermint, semantic aggregation is also applied to PREVOTE and PRECOMMIT voting messages. Given a subset of messages to be sent to a peer node, messages of the same type, for the same height, round and value (block id) can be aggregated. The aggregated message carries a list of pairs $\langle origin, signature \rangle$ identifying originators of the messages aggregated, and only once the identical contents.

#### 3.2.2.2 | *Semantic aggregation keeps consensus properties*

The basic requirement for a sound aggregation strategy is that the information conveyed by an aggregated message is equivalent to the original non-aggregated ones, enabling the same decisions to be taken at the destination. Notice that the restriction to aggregation is strong: aggregated messages should be identical, only stemming from different originators. The receiver of the aggregated message, be it PREVOTE or PRECOMMIT, will compute the exact same votes as for the non-aggregated messages.

### 3.2.3 | Semantic Gossip and BFT

A typical gossip layer is agnostic to the contents carried. Since semantic gossip introduces forwarding decisions based on message contents related to the application layer, in a byzantine environment care has to be taken that aggregation and filtering are not exploited by dishonest nodes. Filtering and aggregation decisions have to be taken with the same security concerns as used by the application level when handling messages.

The security measure taken at consensus layer is to ensure that any received message has its signature verified before processing it. We adopt the same measure for semantic gossip. From an engineering perspective, since signature verification is a costly procedure, instead of performing verification additionally at the gossip layer we propose an interface from consensus to the gossip layer to return a copy of delivered and verified messages. After this verification, gossip proceeds to the semantic decisions. This aspect is shown in Figure 1 with the loop-back arch from signature verification at consensus to the filtering input queue at the semantic gossip layer.

*[fp]: I don't get this part. [fd]: reviewed*

## 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 processes. The resulting overlay has to be connected and each node is required to have at least $k = f + 1$ neighbours. Section 4.2 discusses how nodes can create such an overlay using a distributed procedure.

### 3.3.1 | Classic gossip

Figure 1 (left) illustrates the architecture of a gossip layer. A process interacts with the gossip layer through primitives *broadcast* and *deliver*. Broadcast messages are handled just as gossip messages received from other peers. Messages are delivered once they pass the duplication check. The *delivery queue* offers messages to the upper layer.

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. All Receive routines share the same queue with the local broadcast. A message added to the *broadcast and delivered queues* is locally delivered and sent 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 received messages, maintained by every process. A message is registered to the recently received cache before it is delivered to the consensus algorithm 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 received 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 1: 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 cache size reduces the probability of message duplication. It is worth noting that the cache stores message identifiers (hashes), not full messages, and so, it is relatively small.

### 3.3.2 | Semantic extensions

The gossip layer offers two ways to control its behavior: semantic filtering and semantic aggregation, as presented in Section 3.2. The consensus algorithm can adopt one or both techniques by implementing interface methods offered by the gossip layer. Figure 1 (right) depicts how the classic gossip architecture is extended for filtering and aggregation, and identifies which functions are needed from the consensus layer, i.e., encapsulating semantics of the specific consensus protocol.

#### 3.3.2.1 | *Semantic filtering*

Semantic filterin is provided by allowing the consensus algorithm to implement a *validate* function, which receives a message and a destination peer, and returns a boolean:

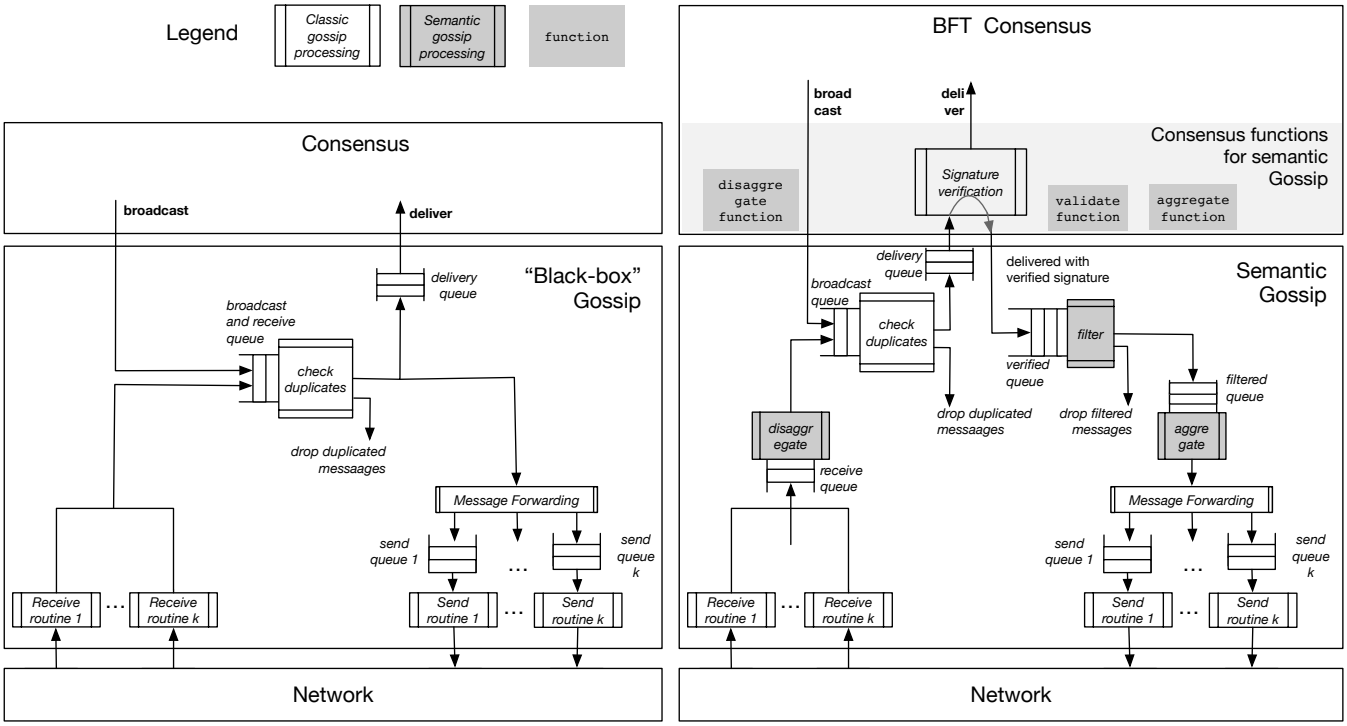



**FIGURE 1** Architectures: (left) - Gossip; (right) Semantic Gossip

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

The *validate* function is invoked by the *filter* module at the gossip layer when a message has not been dropped due to duplicates or signature violation, and is considered for forwarding. If the function 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 function is not implemented. Implementations of the *validate* function 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 resources saving by filtering out messages that would be sent to a peer should be considered.

#### 3.3.2.2 | *Semantic aggregation*

Semantic aggregation is provided through the implementation of a pair of functions, *aggregate* and *disaggregate*, by the consensus layer:

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

The *aggregate* function receives an array of messages and a destination peer, and returns an array of messages. It is invoked by the gossip *aggregate* module when there are multiple pending messages to be sent to the respective peer. Messages returned by the *aggregate* method, either original untouched or aggregated, are sent to the peer, in the order in which they are returned.

Any message received from peers can be aggregated. The *disaggregate* module at the gossip layer invokes the the *disaggregate* function provided by the consensus layer when a message marked as aggregated is received. The *disaggregate* function is the inverse of *aggregate*: it receives an aggregated message and returns an array of reconstructed messages, for reversible semantically aggregated messages. Messages returned by the method are processed as regular messages, in the order in which they are returned. These messages are checked against the recently received cache and, if not duplicated, delivered and forwarded to peers.

# 4 | EXPERIMENTAL EVALUATION

Having proposed and argued for the semantic gossip mechanisms, we addressed the first research question defined in Section 1. Now, we turn our attention to the sencond and third research questions. To address those we build a prototype of our ideas and conduct a series of experiments.

## 4.1 | Methodology

Gossip networks are typically built having nodes connecting to a number of randomly chosen peers. A basic concern is how different topologies could affect gossip and the proposed mechanisms and thus impact the consensus protocol using gossip. To address this concern, our study starts with a characterization of possible topologies obtained from a gossip arrangement of nodes, and their properties, in Section 4.2. We then present system configurations and deployments to conduct experiments in Section 4.3. The first set of experiments aims to answer the second research question, being reported in Sections 4.4 to 4.7. We used increasing workload aiming to assess the relation of throughput and latency. The workload is increased by stepwise enabling Tendermint to handle more instances concurrently. Regarding the third research question, resilience evaluation is reported Section 4.8. We fixed a workload and assessed resiliency by stepwise increasing the number of byzantine nodes in the network from 0% to 30%, 5 by 5%. For each case, we observe again the result of throughput and latency. In Section 4.9 we evaluate throughput and latency under increasing message loss rate.

## 4.2 | Choice and Distributed Generation or Network Overlays

Due to the random generation of network overlays by the gossip protocol, we have to ensure that the overlays used in the experiments are representative, not leading to distortion in the measures obtained. Moreover, we have to provide a distributed algorithm to generate these overlays. In this section we discuss how the overlays used in the experiments can be obtained and their characteristics.

### 4.2.1 | Network overlay definition

A network overlay is a graph $G(V,E)$, where $V$ represents the set of processes running the experiment (validator nodes) and $E : V \times V$ represents the set of bi-directional network connections between processes. $G$ is simple (a process does not connect to itself and there is only one edge among any two nodes) and non-directed (an edge means communication in both directions is possible). The following definitions are useful in next sections:

- $peers_G(p \in V) := \{q \in V : (p,q) \in E\}$ is the set of processes directly connected to $p$ in the overlay $G$.
- $degree_G(p \in V) := |peers_G(p)|$ is the number of connections process $p$ has in the network overlay $G$.
- $outPeers_G(p \in V)$ and $inPeers_G(p)$: the set of outbound and inbound peers of $p$, i.e., processes $q \in peers_G(p)$ to which $p$ is expected to respectively dial and establish a connection, or accept a connection request.

#### 4.2.1.1 | *Soundness*

A network overlay is valid, said sound, if the following requirements are fulfilled:

- connected: $G$ is connected, i.e., there is a path built out of edges in $E$ among any two processes in $G$.
- minimum neighbourhood: for fault-tolerance, each node in the overlay is required to have at least $f+1$ neighbours, where $f$ is the allowed number of faulty processes, ensuring that any node is always connected to at least one honest node. In our case $f < n/3$, $n$ being the total number of nodes.

### 4.2.2 | Distributed generation

Given a network overlay $G$ to be used in an experiment (or instantiation of the protocol), each process $p \in V$ should be able to compute the set $peers_G(p)$ with which they should be connected in the adopted network overlay. Each process $p$ has to be able to identify its *in* and $outPeers_G(p)$. For its generation, we characterize a random network overlay using three parameters:

- $n$: the number of processes in the network, i.e., $n = |V|$.
- $k$: the connectivity parameter, which can be seen as the target value for $degree_G(p)$ for every $v \in V$.
- $s$: the seed employed to randomly select the connections between processes in the network overlay.

Notice that $n$ and $k$ are parameters for the generation of a class of random network overlays, while $s$ allows to uniquely identify each individual random network overlay belonging to that class. Using the same seed and random number generator, a deterministic distributed algorithm can be used to compute the same network at each distributed process. In a distributed algorithm, if each of the $n$ nodes chooses $x$ neighbours to connect, the expected final number of neighbours of a node[†] is $E_{nbr} = 2x - x^2/n$. To generate networks in which $E_{nbr}$ approaches the desired connectivity $k$, in the distributed algorithm we use $x$ obtained from the above relation. For instance, in a network with $n = 128$, if we want nodes to have in average $k = 43$ neighbours, $x$ should be $23,69$. So we use $x = 24$: each node chooses randomly $24$ neighbours.

Our deterministic algorithm for distributed overlay generation has the following main parts: assuming input $\langle k, n, seed \rangle$ and using the same random generator: (a) from $k, n$ find $x$ as above; (b) in the same order of nodes in $n$, for each node randomly choose $x$ neighbours; (c) if the network is not connected or there exists any node with less than $k = f + 1$ neighbours, repeat from (b), otherwise adopt the network. To ease the process of finding a network with the needed minimum connectivity we can slightly increase $x$.

#### 4.2.2.1 | *Classification*

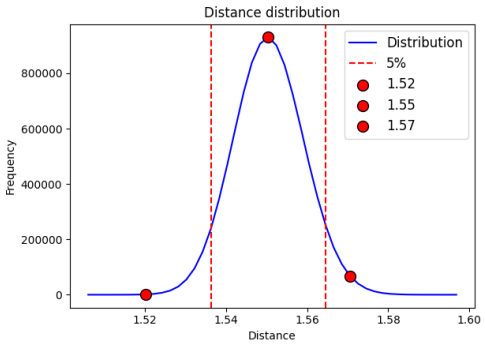



**FIGURE 2** Distance distribution of the random generated overlays for n=32,k=13.8.

Once we have a distributed generation algorithm, we have to evaluate if radom generated topologies are free from any bias that may impact the results. Since the distance among nodes is a fundamental aspect of gossip performance, we classify overlays using their average shortest path length among any pair of nodes. We call this metric the distance of an overlay and in Figures 2 and 3 we present the distance distribution for overlays with $n = 32, x = 8$ leading to $k = 13.87$; and $n = 128, x = 29$ leading to $k = 51.43$. We select sound instances from this population that are in different extremes of the distance distribution (below 5% and above 95%), as well as samples in the center of the distribution. Experiments are run with each selected overlay instance, showing in Figures 4 and 5 that they have comparable impact on all semantic gossip techniques, meaning that the techniques

[†] $E_{nbr} = A + B - C$ where: $A$ is the number of neighbours chosen by this node, $A = x$; $B$ is the number of nodes that choose this one has neighbour, as $n$ nodes choose this one with probability $x/n$, $B = n \times x/n$; and $C$ is the number of mutual choices that may take place, as two nodes choose to be neighbours with probability $(x/n)^2$, we have that for $n$ nodes $C = (x/n)^2 \times n$. With this, $E_{nbr} = x + n \times x/n - (x/n)^2 \times n = 2x - x^2/n$

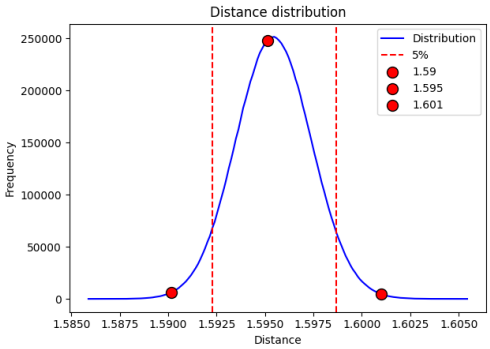



**FIGURE 3** Distance distribution of the random generated overlays for n=128,k=51.3

are not favored by the choice of different topologies generated with the same arguments. Once this is observed, we follow the studies with the average distance topology (center of distribution).

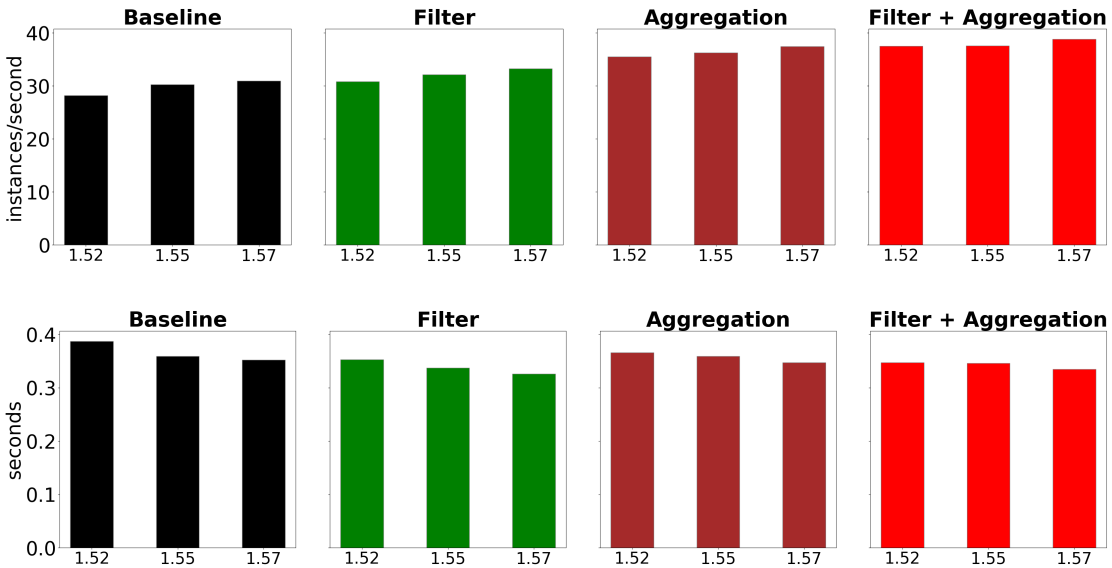



**FIGURE 4** Throughput (above) and latency (below) of the best throughput/latency relation points, for different setups and different average node distances (x axis for each setup) in a network with n=32 and k=13,8.

## 4.3 | System configuration and deployment

We implemented Tendermint, the gossip communication layer, and the Semantic Gossip extensions in Go. We rely on libp2p [13] 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.

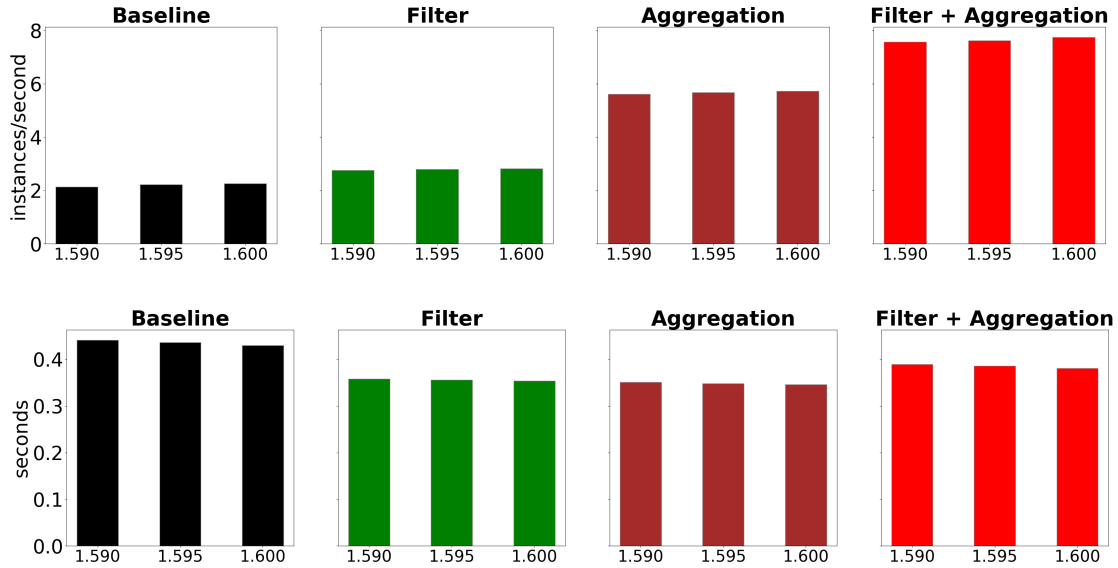



**FIGURE 5** Throughput (above) and latency (below) of the best throughput/latency relation points, for different setups and different average node distances (x axis for each setup) in a network with n=128 and k=51,43.

The same Tendermint implementation is used to build different setups. In the first, *Baseline*, Gossip is used. The other setups use Semantic Gossip, being *Filtering* only, *Aggregation* only, and *Filtering and Aggregation* combined. Either using Gossip or Semantic Gossip, each process opens a libp2p channel to a random subset of processes as discussed in section 4.2.2.

The above setups are experienced with networks of 32 and a 128 nodes. Load is generated at each node, upon its turn to propose a block for validation. I.e. whenever a node is enabled to propose a block, it does. To experience higher workloads, we implemented a varying window of concurrent consensus instances allowed. This means that while a block proposed by a node is being validated, further block(s) can be proposed by the next nodes. The number of blocks that may be under validation concurrently, in the network, is called window. The population of messages in intermediate nodes increases with the window sizes, allowing to better understand the effect of the mechanisms proposed.

A single instance of the experiment is characterized as a combination of: a given a topology (triple $\langle k, n, seed \rangle$); one of the four setups; a concurrency window size; the time-span and the size of proposed value. For each combination of topology and setup, we start with window 1 and increment until saturation. For each window size, we let the nodes free to propose and decide values as fast as possible, during a times-pan of 4 minutes. Each node has a module where the load is generated and instances decided are delivered. This allows to build an output log with all the deliveries and their consensus latencies. Also, during the experiment, a node periodically records working variables such as the size of the gossip queues presented with Figure 1, number of filtered messages and number of aggregated messages.

As deployment environment we use CloudLab's[14] bare metal machines. We use machines type M400 from CloudLab's Utah cluster with configuration eight 64-bit ARMv8 (Atlas/A57) cores at 2.4 GHz (APM X-GENE) and 64GB ECC Memory (8x 8 GB DDR3-1600 SO-DIMMs).

## 4.4 | Overall performance

Figure 6 shows the performance of Tendermint in the four setups: Baseline, Filtering, Aggregation, Filtering+Aggregation, for 32 nodes and 128 nodes. Each bar represents the throughput at the sample with the best throughput/latency relation for each case. This also is depicted with Table 1: for each Semantic Gossip setup, the absolute and relative to the baseline numerical values ('relative' line) are shown.

With 32 nodes, the Baseline saturation point is reached with window 10, with several nodes showing average load[‡] above the number of cores (8 in our experiment) revealing contention for CPU. We also observed that message handling is a main factor

[‡] Linux measure meaning the number of processes which are either currently being executed by the CPU or are waiting for execution.

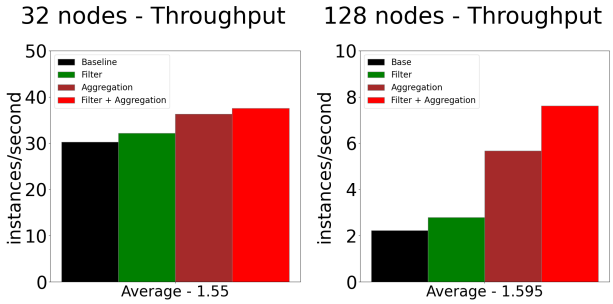



**FIGURE 6** Comparison of throughput of the best throughput/latency relation points for Baseline, Filter, Aggregation and Filter+Aggregation for 32 (left) and 128 (right) nodes, for the average distance topology.

to CPU utilization. As the mechanisms proposed spare messages, the demand for CPU is alleviated. Thus, setups Filtering and Aggregation, in isolation, allow to further scale. The combination of both, Filtering+Aggregation, leads to better performance than the isolated cases, reaching throughput 1.24 × faster than the Baseline while slightly reducing latency.

In experiments with 128 nodes the same is observed. Due to the high number of nodes and consequent message handling demand, the Baseline setup shows saturation starting from window 1, again with average CPU load above the number of cores for several nodes. Filtering and Aggregation alone again show positive results, shrinking latency and enhancing throughput. Their combination allowed to reach 3.42 × the Baseline throughput showing latency compatible with 32 nodes on non-saturated setups.

| Size in Nodes | Baseline | Filtering | Aggregation | Filtering+ Aggregation |
|---|---|---|---|---|
| 32 | 30.261 | 32.145 | 36.271 | 37.559 |
| relative | 1 | 1.06 | 1.20 | 1.24 |
| 128 | 2.223 | 2.791 | 5.672 | 7.618 |
| relative | 1 | 1.25 | 2.55 | 3.42 |

**TABLE 1** Tendermint throughput for different network sizes and gossip setups, at the best throughput/latency point.

## 4.5 | Mechanisms Working

To better understand the effect of the mechanisms, we take the points with best throughput/latency relation and quantify their impact on message handling.

Regarding Filtering, the results for setups Filtering and Filtering+Aggregation are depicted in Table 2. With the 32 nodes network it shows that around 23% of the messages were filtered out in both setups, while with 128 nodes around 30% of the messages were filtered out.

| Size in Nodes | Filtering | Filtering+ Aggregation |
|---|---|---|
| 32 | 23.07% | 23.17% |
| 128 | 29.87% | 28.81% |

**TABLE 2** Percentage of messages filtered out by the semantic filter mechanism at the best throughput/latency point.

With Aggregation, we can observe in Table 3 that the share of messages aggregated with others importantly increases with the number of nodes, revealing that indeed the phased and symmetric structure of the protocol lends itself for the usage of the Aggregation technique.

| Size in Nodes | Aggregation | Filtering+ Aggregation |
|---|---|---|
| 32 | 16.77% | 13.01% |
| 128 | 70.92% | 71.76% |

**TABLE 3** Percentage of messages aggregated at the best throughput/latency point.

## 4.6 | Gossip messages per consensus instance

To further evaluate the impact, we compute the average number of gossip messages received per node to complete a consensus instance.

In an analytical approximation, for a network of $n$ nodes, each of which having $k$ neighbors, considering that a gossiped message reaches every node and each node forwards it once to its neighbors, we have that in average every node should receive $n * k/n = k$ copies of a gossiped message. Now, considering that after the *Propose* by the coordinator, each of the $n$ nodes gossips the *Prevote* and then gossips the *Precommit*, our estimation is that for every instance of consensus a node should receive $2nk$ gossiped messages. For instance, with $n = 32$ and $k = 13.87$, a consensus instance would lead each node to receive $\sim 887$ messages, while with $n = 128$ and $k = 51.43$, would be $\sim 13.166$ messages. However, as a node doesn't need every message to progress, and, depending on messages received, may skip phases, this upper bound should not be meet in practice.

In Table 4 we depict the observed number of gossip messages handled per node, per consensus instance, for the several setups. The baseline values are 7,9% and 2,8% under the estimated upper bound. We notice that Filtering showed reduction of 26% and 31% of messages respectively for 32 and 128 nodes. As discussed, Aggregation shows high impact as the number of nodes increase, here with 17% and 75% less messages for 32 and 128 nodes respectively. Again, the mechanisms combined show still better effect than in isolation for both network sizes.

| Size in Nodes | Estimated upper bound | Base-line | Filte-ring | Aggre-gation | Filter.+ Aggreg. |
|---|---|---|---|---|---|
| 32 | 887 | 822,2 | 612,6 | 690,6 | 537,9 |
| relative | 1,079 | 1 | 0,74 | 0,83 | 0,65 |
| 128 | 13.166 | 12.805,4 | 8.876,3 | 3.613,1 | 2471 |
| relative | 1,028 | 1 | 0,69 | 0,25 | 0,19 |

**TABLE 4** Average of gossip received messages per node, per decided Tendermint instance, at the best throughput/latency point.

## 4.7 | Latencies

Window 1 represents the situation where an instance of consensus is proposed only after the previous has been finished, therefore with minimum contention. For 32 nodes and window 1, we observe in Figure 7(center) that all setups have the same latency distribution (lines are collapsed). This means that the semantic mechanisms neither help nor impacted latency in this light scenario.

Interestingly, from Figure 7 we observe that with 128 nodes, window 1 (right), the setups with Aggregation show latency distribution very similar to the reported with 32 nodes, window 1 (center), while the Filtering and Baseline show higher latency, in this order. This emphasizes the importance of Aggregation as the number of nodes grows. As Tendermint uses communication steps with the same types of messages addressed to all participants, the population of messages prone to Aggregation in intermediate nodes, and thus the positive impact of Aggregation, is proportional to the number of nodes.

To detail the behavior with 32 nodes we take from the less performing setup (Baseline) the window with best throughput/latency point (window 13). With this configuration all setups show similar throughput. The latency distributions of the four setups is depicted in Figure 7 (left). Latency distributions are slightly better using Semantic Gossip. Aggregation shows better impact than Filtering, and their combination further reduces latency.

An important general observation is that the mechanisms are not delaying instance's decision, even if messages are dropped due to filtering. On the contrary, latency distributions are consistently better at all percentiles with the mechanisms proposed, showing that sparing redundancy is indeed possible and a good measure.

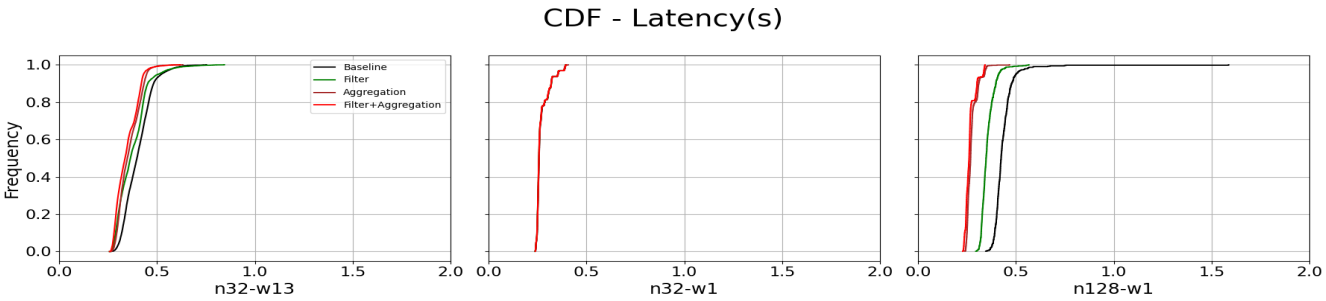



**FIGURE 7** Latency CDFs for Baseline, Filter, Aggregation and Filter+Aggregation, using window 13 for 32 nodes (left), window 1 for 32(center) and window 1 for 128 nodes(right)

## 4.8 | Resilience

*[fd]: SHOULD WE KEEP THIS OR DO THE MESSAGE LOSS STUDY?*

As redundancy is needed to cope with byzantine behavior and since Semantic Gossip is designed to safely eliminate redundancy, we designed experiments to evaluate if and how consensus is affected by byzantine behavior using Semantic Gossip mechanisms in comparison to the Baseline.

Under the assumption that signatures are reliable and since gossip works with signature verification, any attempt to forge or corrupt messages would be detected. Therefore, we conclude the worst byzantine behavior to harm consensus would be that dishonest nodes remain silent, i.e. they do not collaborate with the consensus protocol.

For 32 and 128 nodes, we incrementally convert honest to byzantine nodes, 5 by 5%, starting with 0 and going up to 30%. Regardless of performance, consensus should show progress in all cases since it supports up to 1/3 dishonest nodes. Tables 5 and 6 show the resulting throughput of the experiment respectively for 32 and 128 nodes. The first general observation is that all configurations keep progress. The second is that for 128 nodes all Semantic Gossip configurations performed better than or equivalent to the Baseline. Thirdly, for 32 nodes, in all configurations the effect of having Filtering+Aggregation slightly enhanced or has equivalent throughput. For Filtering and Aggregation separated, in most cases performance was equivalent or better than the Baseline. For 30% dishonest nodes these setups show performance below the Baseline, indicating that message drop by the mechanisms, added by 30% of silent nodes, have slightly delayed consensus to be reached.

| Byzantine Nodes | Baseline | Filtering | Aggregation | Filtering+ Aggregation |
|---|---|---|---|---|
| 0 % | 1 | 1 | 1 | 1 |
| 5 % | 0.044 | 0.041 | 0.040 | 0.037 |
| 10 % | 0.022 | 0.020 | 0.021 | 0.019 |
| 15 % | 0.022 | 0.020 | 0.020 | 0.018 |
| 20 % | 0.021 | 0.017 | 0.019 | 0.018 |
| 25 % | 0.017 | 0.019 | 0.023 | 0.023 |
| 30 % | 0.017 | 0.010 | 0.009 | 0.016 |

**TABLE 5** Tendermint throughput in instances/sec with 32 nodes. Each setup with window of best throughput/latency point with 0 dishonest nodes. Then converting nodes 5 by 5% to dishonest.

## 4.9 | Message loss

In addition to Byzantine behaviors, a consistent consensus mechanism must also be tolerant of imperfect links that allow message loss. We implemented this feature by dropping messages, when they are received, with a predefined probability, so they are not computed by the consensus neither forwarded to other peers. We measure latency and throughput of the different techniques with loss probabilities increasing 5 by 5 from 0 to 60%, in a network of 32 nodes. With window 1, we observe

| Byzantine Nodes | Baseline | Filtering | Aggregation | Filtering+ Aggregation |
|---|---|---|---|---|
| 0 % | 1 | 1 | 1 | 1 |
| 5 % | 0.329 | 0.249 | 0.173 | 0.131 |
| 10 % | 0.200 | 0.153 | 0.094 | 0.065 |
| 15 % | 0.157 | 0.115 | 0.075 | 0.051 |
| 20 % | 0.116 | 0.085 | 0.051 | 0.035 |
| 25 % | 0.100 | 0.073 | 0.042 | 0.039 |
| 30 % | 0.083 | 0.068 | 0.062 | 0.052 |

**TABLE 6** Tendermint throughput in instances/sec with 128 nodes. Each setup with window of best throughput/latency point with 0 dishonest nodes. Then converting nodes 5 by 5% to dishonest.

a gradual degradation of the performance, both for throughput and latency, until we arrive at 50 to 60% message loss. With window 13, Figures 8 and 9 show that we have a gradual degradation of the performance, until we arrive at 30% message loss, where the number of decided instances drops abruptly. The baseline has this threshold point with 35% message loss. This shows that, with Semantic Gossip filtering and aggregation, the system operates with performance better than the Baseline even under loss rates up to 30%.

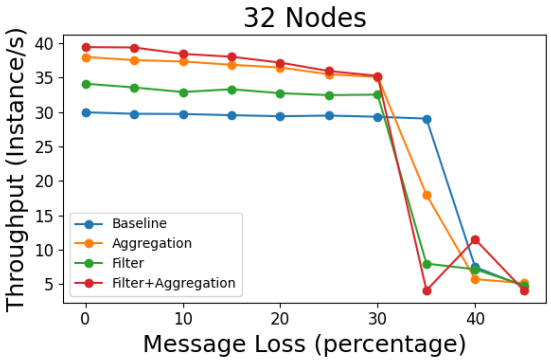



**FIGURE 8** Tendermint throughput in instances/s with 32 nodes. Setups with window size equals 13. Message loss percentage incremented 5 by 5.

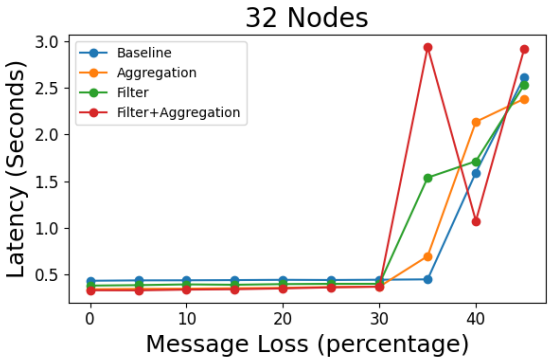



**FIGURE 9** Tendermint latency in seconds with 32 nodes. Setups with window size equals 13. Message loss percentage incremented 5 by 5.

## 5 | RELATED WORK

Gossip algorithms were first introduced by Demers et al. [7] to manage replica consistency in the Xerox Clearinghouse Service [15]. 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 [8]. 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 [16, 17, 18, 19, 20, 21, 22, 23].

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 [19, 22], or topological and connectivity properties [20, 21, 24]. 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, 8] or gossip repair [16, 17, 18, 23] 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 [25], group membership [26, 27], monitoring and management systems [28].

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 [29] 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 [29] 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 [3] designed for gossip communication. Tendermint has its own gossip layer implementation, that is application-specific and tightly coupled with the consensus implementation. Casper [4], 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 [30], the BFT consensus protocol at the core of the Libra Blockchain [31], although not designed for gossip-based communication, considers its adoption as the number of processes participating on consensus (validator nodes) grows [5]. 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

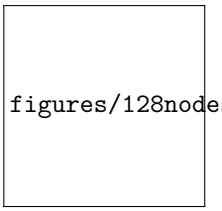
This paper introduces Semantic Gossip, a communication substrate that takes application semantics into account to optimize performance. Semantic Gossip relies on two techniques, semantic filtering and semantic aggregation. With semantic filtering, the gossip protocol can stop propagating messages that have become redundant from the perspective of the application. With semantic aggregation, the gossip protocol can replace multiple 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 the consensus protocol.

We have demonstrated the usefulness of Semantic Gossip using Tendermint, a well-known consensus protocol. It enables to scale ...

## REFERENCES


1. L. Lamport, "Time, clocks, and the ordering of events in a distributed system," vol. 21, no. 7, pp. 558–565, Jul. 1978.
2. F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.
3. E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *arXiv e-prints*, p. arXiv:1807.04938, Jul. 2018.
4. V. Buterin and V. Griffith, "Casper the Friendly Finality Gadget," *arXiv e-prints*, p. arXiv:1710.09437, Oct. 2017.
5. L. E. Team, "Libra: The path forward," Online, Jun. 2018. [Online]. Available: https://libra.org/en-US/blog/the-path-forward/
6. E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *EuroSys*, 2018.
7. A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing - PODC'87*. ACM Press, 1987.
8. K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal multicast," *ACM Transactions on Computer Systems (TOCS)*, vol. 17, no. 2, pp. 41–88, May 1999.
9. E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on bft consensus," 2019. [Online]. Available: https://arxiv.org/abs/1807.04938
10. M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, Feb. 1999.
11. A. Poelstra, "Distributed consensus from proof of stake is impossible," 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:53311056
12. R. Friedman and R. van Renesse, "Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols," Cornell University, Dept. of Computer Science, Tech. Rep. 94-1527, Jul. 1995, submitted to IEEE Transactions on Networking.
13. "Libp2p," https://libp2p.io, [Accessed 2020-05-17].
14. D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14. [Online]. Available: https://www.flux.utah.edu/paper/duplyakin-atc19
15. D. C. Oppen and Y. K. Dalal, "The clearinghouse: a decentralized agent for locating named objects in a distributed environment," *ACM Transactions on Information Systems (TOIS)*, vol. 1, no. 3, pp. 230–253, Jul. 1983.
16. K. P. Birman, R. van Renesse, and W. Vogels, "Spinglass: secure and scalable communication tools for mission-critical computing," in *Proceedings DARPA Information Survivability Conference and Exposition II*, ser. DISCEX'01, vol. 2. IEEE Comput. Soc, 2001, pp. 85–99.
17. P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec, "Lightweight probabilistic broadcast," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 4, pp. 341–374, Nov. 2003.
18. I. Gupta, K. P. Birman, and R. van Renesse, "Fighting fire with fire: using randomized gossip to combat stochastic scalability limits," *Quality and Reliability Engineering International*, vol. 18, no. 3, pp. 165–184, 2002.
19. D. Kempe, J. Kleinberg, and A. Demers, "Spatial gossip and resource location protocols," *Journal of the ACM (JACM)*, vol. 51, no. 6, pp. 943–967, nov 2004.
20. M.-J. Lin and K. Marzullo, "Directional gossip: Gossip in a wide area network," in *Proceedings of Third European Dependable Computing Conference*, ser. EDCC-3. Springer Berlin Heidelberg, Sep. 1999, pp. 364–379.
21. J. Leitao, J. Pereira, and L. Rodrigues, "Epidemic broadcast trees," in *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE, oct 2007.
22. R. Melamed and I. Keidar, "Araneola: a scalable reliable multicast system for dynamic environments," in *Proceedings of the Third IEEE International Symposium on Network Computing and Applications*, ser. NCA 2004. IEEE, 2004.
23. W. Vogels, R. van Renesse, and K. Birman, "The power of epidemics," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 131–135, Jan. 2003.
24. S. Voulgaris and M. van Steen, "Vicinity: A pinch of randomness brings out the structure," in *Middleware 2013*. Springer Berlin Heidelberg, 2013, pp. 21–40.
25. R. van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service," in *Middleware'98*. Springer London, 1998, pp. 55–70.
26. A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié, "Peer-to-peer membership management for gossip-based protocols," *IEEE Transactions on Computers*, vol. 52, no. 2, pp. 139–149, Feb. 2003.
27. H. Johansen, A. Allavena, and R. van Renesse, "Fireflies: scalable support for intrusion-tolerant network overlays," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 3–13, Apr. 2006.
28. R. van Renesse, K. Birman, D. Dumitriu, and W. Vogels, "Scalable management and data mining using astrolabe*," in *Peer-to-Peer Systems*. Springer Berlin Heidelberg, 2002, pp. 280–294.
29. P. Felber and F. Pedone, "Probabilistic atomic broadcast," in *Proceeedings of 21st IEEE Symposium on Reliable Distributed Systems, 2002*, ser. SRDS '02. IEEE Computer Society, pp. 170–179.
30. M. Yin, D. Malkhi, M. K. Reiter, G. Golan Gueta, and I. Abraham, "HotStuff: BFT Consensus in the Lens of Blockchain," *arXiv e-prints*, p. arXiv:1803.05069, Mar. 2018.
31. Z. Amsden, R. Arora, S. Bano, M. Baudet *et al.*, "The libra blockchain," The Libra Association, Tech. Rep., May 2020, [Accessed 2020-06-01]. [Online]. Available: https://developers.libra.org/docs/the-libra-blockchain-paper

# 7 | AUTHOR BIOGRAPHY




**Author Name.** Please check with the journal's author guidelines whether author biographies are required. They are usually only included for review-type articles, and typically require photos and brief biographies for each author.