Scalable Byzantine Fault-Tolerant Gossip

Matej Pavlovič
LPD, I&C, EPFL

Abstract—**Gossip-based protocols are a promising approach to efficient and robust data dissemination in large-scale distributed systems. However, their robustness is often limited to tolerating simple (non-Byzantine) failures, which is not always sufficient. Byzantine fault tolerance in gossip comes at the cost of sacrificing or limiting other desirable properties of gossip, like scalability, decentralized operation or performance. In this proposal, we identify weaknesses of several approaches to Byzantine fault tolerant gossip. We propose Robocop, a computation framework addressing these weaknesses by combining the principles of gossip and state machine replication.**

Index Terms—**gossip, Byzantine fault tolerance, scalability, dynamic membership**

I. Introduction

GLOBAL services provided over the Internet have become very popular in recent years. They are being used by billions of users distributed all across the planet. Examples of such services include social networks (like Facebook or Twitter), web indexes and search engines (Google, Bing), content sharing and distribution services (YouTube, Dropbox, Netflix, BitTorrent), communication platforms (Skype), or virtual currencies (BitCoin). All these services are implemented using distributed systems. Given the amount of users and their requirements on service availability and performance, the distributed systems must be scalable and robust against internal failures or even malicious attacks.

Proposal submitted to committee: June 13th, 2009; Candidacy exam date: June 20th, 2009; Candidacy exam committee: Exam president, thesis director, co-examiner.

This research plan has been approved:

Date: ____________________

Doctoral candidate: ____________________
(name and signature)

Thesis director: ____________________
(name and signature)

Thesis co-director: ____________________
(if applicable) (name and signature)

Doct. prog. director: ____________________
(B. Falsafi) (signature)

EDIC-ru/05.05.2009

In order to build such distributed systems, a core problem that needs to be solved is efficient data dissemination. For dynamic[1] systems consisting of millions of nodes, a significant fraction of which may be controlled by a malicious adversary, this problem becomes non-trivial.

In [1], Malkhi et al. introduce the *diffusion problem*, where an update (i.e. a message) that is initially known by a small number of source nodes must be disseminated to and accepted by all correct nodes.

In a naive approach to the diffusion problem, source nodes could directly send the update to all other nodes. Obviously, this approach is impractical for large-scale systems, because the load on the source nodes grows linearly with system size.

To distribute the load more evenly, a structured overlay network in form of a tree may be used. A source node, being at the root of the tree, only sends updates to its children in the tree. The children forward updates to their children and so on, until the updates reach the leaf nodes. This approach (usually referred to as *broadcast trees* or *multicast trees*) uses network bandwidth very efficiently, because every node receives each update only once. However, even simple node or link failures may prevent a big fraction of the system from receiving updates.

A promising approach to scalable and robust diffusion of updates in dynamic systems are *gossip algorithms* (also called *epidemic algorithms*). As the name suggests, updates in a gossip-based distributed system spread like rumors in a group of gossiping people (or like infectious diseases in a population).

With gossip, each node periodically exchanges updates with a randomly chosen gossip partner. It can be shown that every update eventually reaches every node with probability 1. Assuming benign faults, the expected number of gossip rounds necessary to disseminate an update is logarithmic in the number of nodes. Thanks to the random communication pattern, gossip algorithms easily cope with crashed nodes and failed links. The absence of a structured overlay also makes it easy to handle churn (nodes joining and leaving at run-time).

Gossip was first used by Demers et al. to disseminate updates in replicated databases [2]. Today, the use of gossip is by no means restricted to simple message dissemination. Gossip-based algorithms can also be employed for constructing and maintaining overlay networks [3], multimedia streaming [4] [5], or distributed peer sampling [6].

Numerous flavors of gossip algorithms exist, differing in the way gossip partners are selected, the choice of updates

[1]Dynamicity here means that the set of nodes a system consists of changes over time.

to propagate, the protocol termination conditions or the assumptions on the underlying system. A general framework for gossip-based algorithms was described by Kermarrec and van Steen [7].

Some approaches combine gossip with other strategies, such as broadcast trees, to achieve efficient steady-state behavior while maintaining gossip's tolerance to churn and node/link failures. One of the first works to combine tree-based multicast with gossip was Bimodal multicast [8]. Bimodal multicast works in two phases: in the first phase, an efficient but unreliable protocol like IP multicast is used. The second phase uses gossip to mask omissions that might have occurred in the first phase. Leitão et al. use gossip to dynamically construct self-repairing broadcast trees [9].

Although gossip protocols are generally regarded as robust, their robustness is questioned by Alvisi et al. In their work [10], they identify several assumptions on which the robustness of many gossip-based protocol relies, and argue that these assumptions might not always be valid in practice.

Another particularly interesting problem in gossip algorithms is tolerance to malicious (Byzantine) node behavior. If malicious nodes inject spurious updates or change the contents of propagated updates, it becomes hard for correct nodes to distinguish between "correct" and "incorrect" updates (and accept only the "correct" ones). This is analogous to real life gossip, where it is also hard for a person to tell whether a rumor is true or false.

The diffusion problem in a Byzantine environment was first studied by Malkhi et al. [1]. They proposed a class of *direct verification*[2] protocols that solve the diffusion problem. They proved lower bounds on the dissemination time of this class of protocols, rendering the protocols impractical at large-scale.

Minsky et al. proposed *path verification* protocols [11] for the Byzantine diffusion problem, generalizing direct verification and circumventing the lower bounds of [1]. However, the dissemination time of path verification protocols is linear in the number of tolerated failures. Scalability is thus still limited if the fraction of Byzantine nodes is constant.

In our research, we target scalability of dynamic gossip-based systems, in which an important fraction of nodes may behave maliciously. Our approach combines gossip with state machine replication, leveraging the Byzantine fault-tolerance of state machine replication and the scalability of (benign fault-tolerant) gossip. The result is a framework for efficient data dissemination across a dynamic large-scale distributed system, tolerating a constant fraction of malicious nodes.

The rest of this document is structured as follows. Section II elaborates on path verification protocols [11] and their performance in disseminating messages. Section III describes FlightPath [4], a system for streaming live content using gossip. Fireflies [3], a gossip-based membership service is described in Section IV. Finally, in Section V, we summarize the weaknesses of the described systems and sketch our approach to overcoming them.

[2]The term "direct verification" was introduced later by Minsky et al. [11]

II. Update Dissemination Using Path Verification

This section describes an approach to dissemination of updates using *path verification* protocols [11]. These protocols rely on tracking the path along which updates are relayed from node to node. Nodes use this path information to decide whether the update can be accepted or not.

A. Model and assumptions

1) System: We assume a system of n nodes, where each node can communicate with any other node. The assumed system is synchronous (i.e. computation and communication delays are bounded), such that the protocols can be described in terms of rounds.

2) Adversary: Correct nodes follow the specified protocol. The system may contain up to t malicious nodes that can behave arbitrarily. They might even collude in order to harm the system as much as possible. Since no cryptographic primitives are used in the protocols, it is not necessary to assume the malicious nodes to be computationally bounded. We assume, however, that messages are not modified by the underlying network and that a correct node can determine the sender of each message (making it impossible for malicious nodes to impersonate other nodes).

3) Initial state: For simplicity of description, we assume that only a single update is being disseminated. At the start of each instance of the protocol, there are $k > t$ correct nodes (*source nodes*) that have accepted the same initial update. The way this is ensured is not part of the protocol.

B. Dissemination protocols

Generally, a node accepts an update if it has at least $t + 1$ copies of that update originating at different source nodes. To obtain these copies, it periodically (once per round) gossips with randomly chosen nodes.

We start by describing direct verification protocols, which are a special case of path verification protocols. Afterwards, we present the general concept of path verification.

1) Direct verification: With direct verification, a node accepts an update when it has gossiped with $t+1$ different nodes that have *already accepted* that update.

The drawback of this approach is that if $n \gg k$, the update propagates slowly in the initial phase of the protocol. This is due to the initially low probability (k/n) of gossiping with a node that already accepted the update. Once a certain fraction of nodes accepted the update, the remaining nodes follow very quickly.

2) Path verification: Path verification addresses the issue of direct verification by allowing updates to spread before they are accepted. To this end, instead of simple updates, so-called *proposals* are gossiped. A proposal consists of an update and a *path*. A path is a sequence of node identifiers. Source nodes are initialized with proposals containing the update and an empty path. A node v that receives a proposal by gossiping with node u appends u to the path of the received proposal.

Since u may be malicious and forge the path of the gossiped proposal, the source node of that proposal is not known to v.

Therefore, v accepts an update if it has received $t+1$ proposals that:

1) contain the same update and
2) have disjoint paths.

This ensures that at least one path contains only correct nodes and thus the update is correct.

If nodes kept all received proposals and transferred all proposals they have each time they gossip, the number of proposals at each node would grow exponentially. Therefore, path verification relies on two sub-protocols:

- a *sampling protocol* that decides which of the received proposals a node keeps, and
- a *selection protocol* that, for each host, chooses a single proposal that is transmitted to its gossip partner.

Now we can see that direct verification is indeed a special case of path verification. It corresponds to keeping only proposals with paths of length one and selecting a proposal with the accepted update and an empty path.[3]

Minsky et al. propose several path verification protocols, differing in the sub-protocols used for sampling and selection:

- *Direct Diffusion*: A direct verification protocol as described above.
- *Youngest Diffusion*: The sampling protocol keeps only a fixed number of most recently received proposals. The selection protocol chooses the "youngest" known proposal, where the "age" of a proposal is the number of rounds since the proposal originated.
- *Promiscuous Youngest Diffusion*: Like Youngest Diffusion, but a node that has accepted an update acts like a source node, always selecting a new proposal with age 0 and the accepted update.
- *Hybrid Diffusion*: Equivalent to running Youngest Diffusion and Direct Diffusion in parallel. It combines the "fast finish" of Direct Diffusion (see II-B1) with the improved start of Youngest Diffusion.

The selection protocol of all above-mentioned protocols allows a node to obtain only a single proposal per gossip round (the one that was selected by the selection protocol at its gossip partner). A sampling protocol called *Bundle Sampling* maintains a whole set of proposals to be gossiped every round. Using Bundle Sampling, faster update dissemination can be achieved at the cost of increased memory and bandwidth requirements.

C. Evaluation

Figure 1 shows the dissemination times of Direct, Youngest and Hybrid Diffusion in a simulated system of $n = 1000$ nodes. We can clearly see that the more sophisticated path verification protocols outperform simple Direct Diffusion. An interesting point is that Hybrid Diffusion performs significantly better than either of Direct and Youngest Diffusion. This is due to the different causes of Direct and Youngest diffusion being slow. Where one is slow the other is fast and vice-versa, yielding a better overall performance.

[3]Note that the selected proposal need not necessarily be one that was received from another node. Before accepting an update, no proposal is selected.

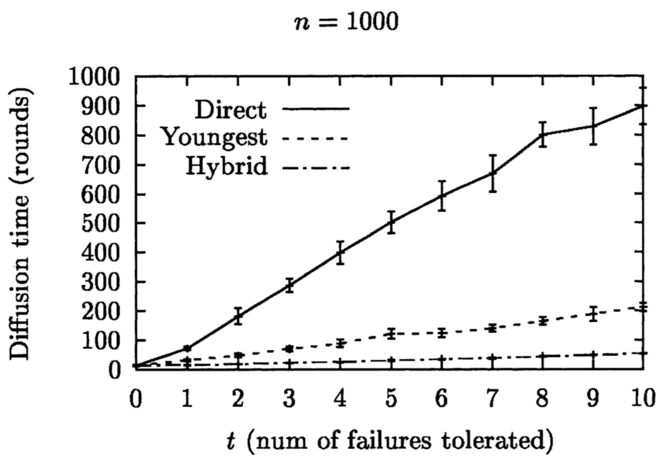

Fig. 1. Comparison of dissemination times for Direct Diffusion, Youngest Diffusion and Hybrid Diffusion, all without Bundle Sampling. Even though the slopes of the curves are different, they all exhibit a linear dependency on the number of tolerated failures.

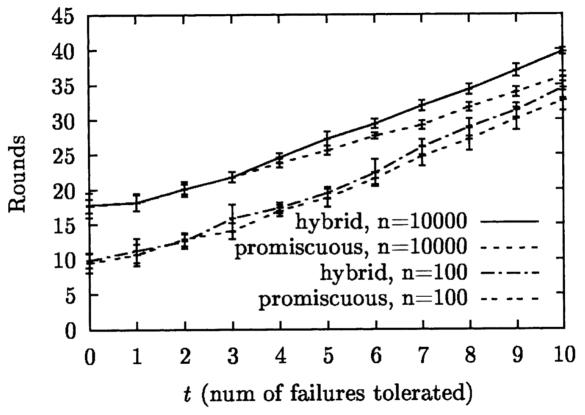

Fig. 2. Comparison of Hybrid Diffusion and Promiscuous Youngest Diffusion with Bundle Sampling. The linear relation between the number of tolerated failures and the diffusion time (vertical axis) is evident.

The comparison of Hybrid Diffusion to Promiscuous Youngest Diffusion, both used with Bundle Sampling, is shown in Figure 2. First, looking at the scale of the vertical axis, we can see the dramatic impact of Bundle Sampling on dissemination time. Second, Promiscuous Youngest Diffusion slightly outperforms Hybrid Diffusion in simulation. However, the authors of [11] state that while the worst-case behavior of Hybrid diffusion can be easily bounded, it is not the case for Promiscuous Youngest Diffusion.

As shown in figures 1 and 2 , the dissemination time is linear with the number of faults tolerated. This is impractical for large-scale systems, since the number of expected faults usually grows linearly with system size.

III. Data Streaming with FlightPath

FlightPath [4] is a gossip-based system for streaming live content. It is bandwidth-efficient, supports dynamic membership with high churn and tolerates a fraction of Byzantine nodes.

We use the BAR (Byzantine-Altruistic-Rational) model [12] to describe the possible behavior of nodes. Altruistic nodes faithfully follow the protocol. Rational nodes deviate only if they expect a benefit (e.g. better stream quality, lower network utilization, etc.) from doing so. Byzantine nodes may behave arbitrarily, even maliciously.

FlightPath uses a game-theoretical model of ε-Nash equilibria to argue that rational nodes have little incentive to deviate from the protocol. In contrast to previous work [5], which uses strict Nash equilibria, FlightPath trades in a little amount of fairness between nodes to increase the overall performance of the system.

In this section, we first describe the basic gossip protocol used by FlightPath. Since the performance delivered only by the basic protocol is low, we also elaborate on several optimizations that substantially improve FlightPath's performance. We continue by explaining how FlightPath handles churn and finish by evaluating tolerance to Byzantine faults and identifying weak points of FlightPath.

A. Basic protocol

The data stream originates at a single source node, which is assumed to be correct. The source splits the stream into chunks called *stream updates*. To ensure the authenticity of the updates, the source also computes their digests and signs them.[4]

The protocol progresses in synchronous rounds. Each round, the source sends new updates and their signed digests to a small number of other nodes. For further dissemination to the remaining nodes, FlightPath uses a special gossip protocol. Each update expires after a fixed number of rounds. When an update expires, all nodes that possess it deliver it to the application and stop gossiping about it.

The gossip protocol used for update dissemination relies on three basic ideas:

1) *Verifiable pseudo-random partner selection*: Each round, every node pseudo-randomly selects one gossip partner. As the seed for the random number generator, it uses its own signature of the current round number. It transmits the seed together with the gossip request, such that the gossip partner can verify the selection. If the verification fails, the gossip partner assumes that the request is coming from a rational or Byzantine node and ignores the request.
2) *Fair trading of updates*: Two gossiping nodes first exchange lists (called *histories*) containing IDs of updates they already have and IDs of updates they still need. They then deterministically choose updates to exchange, under the constraint that the number of received updates must equal the number of sent updates. In case one of the nodes has "nothing to offer", the nodes cancel the trade.
3) *Deferred gratification*: After selecting the updates to exchange, the nodes send each other encrypted "briefcase" messages containing the selected updates.
   They also exchange *promise* messages containing signed hashes of the briefcase contents. The purpose of the promises is to prevent malicious nodes from sending garbage instead of updates. If a briefcase contains junk data, the corresponding promise can be used as a *proof of misbehavior* (POM), leading to the eviction of its sender from the system. We assume that a malicious node will not perform steps that lead to its eviction from the system. Only after receiving a briefcase and the corresponding promise, a node reveals the decryption key for the other briefcase to its gossip partner. The key exchange protocol itself is constructed in a way that provides little incentive for a rational node to not send its key.

[4]Unlike the model used in path verification protocols, this requires a computationally bounded adversary.

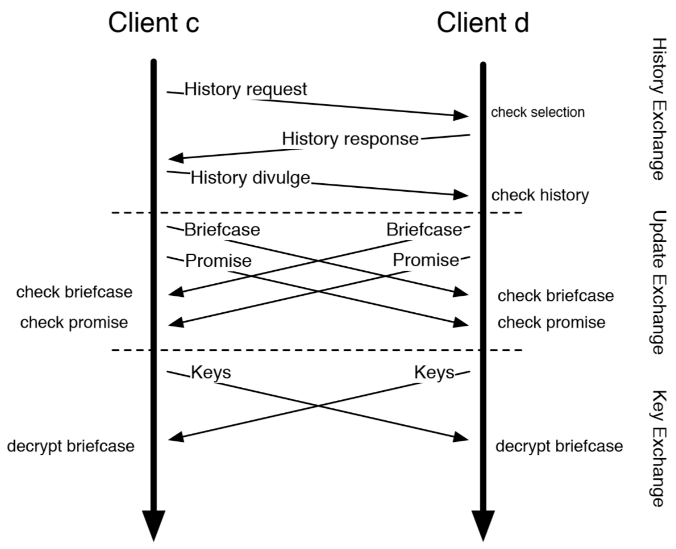

Fig. 3. Basic protocol that is executed by two gossiping nodes in FlightPath.

A graphical representation of the gossip protocol is depicted in Figure 3.

B. Optimizations

Using only the basic protocol, FlightPath has several issues concerning reliability of delivery, bandwidth efficiency, and even load distribution. Therefore, FlightPath includes several optimizations to cope with these issues. Some of the optimizations allow rational nodes to benefit from the system without contributing their fair share to the dissemination. However, the amount of benefit they are able to obtain is always limited. The optimizations are as follows:

1) Reservations: Although each node gossips with two other nodes each round on expectation[5], the actual number of gossip partners may vary. On one hand, this can lead to nodes failing to receive all updates before expiration (if the nodes are not lucky enough to be selected by other nodes and therefore have fewer gossip partners). On the other hand, a node involved in many concurrent exchanges experiences a higher network load.

To overcome these problems, nodes *reserve* trades with each other in advance. A node may send a gossip request for round r already in round $l < r$. The selected gossip partner remembers the request and rejects all subsequent gossip requests for round

[5]Each round, a node sends one gossip request and expects to receive one gossip request.

r. A node whose gossip request has been rejected tries to reserve a trade with another partner.

The basic partner selection mechanism (see III-A1) must be also modified to allow "rejected" nodes to try other gossip partners. In the modified protocol, the verifiable pseudo-random selection maps to several nodes instead of just one. Each of those nodes must accept a gossip request if it has not already reserved a trade for the corresponding round.

2) Splitting need: A node involved in several concurrent trades is likely to receive duplicate updates, effectively wasting bandwidth. The "*splitting need*" optimization limits the number of updates exchanged in a single trade. Although this reduces the number of duplicate updates, it also increases the chance that a node will not receive all the updates it needs. This problem is addressed by using erasure codes.

3) Erasure codes: Instead of simply cutting the data stream into k updates, the source encodes each round's stream data into $m > k$ updates, only k of which are needed to reconstruct the stream. After a node has collected k updates for a round, it stops requesting more updates for that round.

4) Tail inversion: In general, it is preferable for nodes to exchange new updates when possible. New updates are generally more valuable because it is more likely that they can be used for trading in the future. However, a node that misses some old updates may have trouble receiving those if its gossip partners have many new updates. As a countermeasure, a node may explicitly request updates from the two oldest rounds before expiration.

5) Imbalance ratio: With this optimization enabled, nodes track the number of sent and received updates for each gossip partner. Individual trades are allowed to be imbalanced (i.e. one node sends more updates than it receives), as long as the overall ratio of sent and received updates does not exceed a certain threshold.

6) Trouble detector: If a node is "in trouble", i.e. it misses too many updates, it may initiate more than one trade per round. Note that this is possible due to the modified partner selection mechanism introduced for reservations (see III-B1). The price to pay is increased network load.

C. Dynamic membership

FlightPath relies on a centralized *tracker* (which is assumed to be always correct and available) that maintains an up-to-date *membership list*. It monitors the nodes using a ping protocol and removes crashed ones from the list. New nodes have to contact the tracker before joining the system. The tracker periodically updates the *membership list* to reflect the membership changes.

Nodes have their clocks synchronized with the tracker. Time is sliced into *epochs* and each epoch has a corresponding membership list. The tracker periodically sends new membership lists to the source node, which disseminates them the same way as it does the data stream. The source node starts disseminating new membership list early enough, such that a node always has received the membership list at the start of the corresponding epoch.

With the approach just described, a new node cannot start participating in the stream immediately. It has to wait until it is included in the current epoch's membership list. To avoid this problem, the authors of FlightPath developed an extension to the protocol that allows nodes to verify partner selections even without global knowledge of the system.

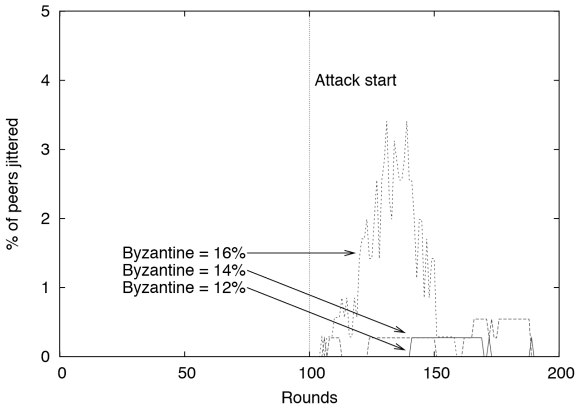

Fig. 4. Stream quality during a Byzantine attack of 12, 14 and 16 percent of nodes. The vertical axis shows the percentage of jittered nodes in each round. A node is considered jittered in a round if it does not receive all updates for that round.

D. Evaluation

FlightPath was evaluated using a 200 Kbps data stream in a system consisting of several hundreds of machines. In a scenario with no Byzantine failures, it was able to deliver the complete stream to all nodes, using 250 Kbps of bandwidth on average.

During a simulated attack of Byzantine nodes, the average network bandwidth grew by up to 100 Kbps. The stream quality during the attack is depicted in Figure 4.

The results demonstrate that FlightPath is suitable for streaming live content with reasonable overhead, even in presence of over 10% Byzantine nodes.

A drawback of FlightPath is its reliance on a centralized tracker for membership management. The tracker may become a scalability bottleneck for large systems. This might especially be the case if it is replicated for reliability using standard state machine replication techniques like [13] or [14] (as suggested by the authors of FlightPath).

Another scalability limit of FlightPath may be the dissemination of complete membership lists to all nodes, since the length of those lists grows linearly with system size.

A fully distributed and Byzantine-tolerant membership service is described in the next section.

IV. Membership Management with Fireflies

Fireflies [3] is a protocol for maintaining Byzantine-tolerant network overlays. It provides each member node with the complete view of all other members, and can thus also be seen as a distributed membership service. It provides a pseudo-random mesh for efficient gossip-based communication, which it also uses to disseminate protocol-internal messages.

This section gives an overview of the membership protocol of Fireflies, describes how gossip is used to support the membership protocol, evaluates Fireflies' performance and identifies its weaknesses.

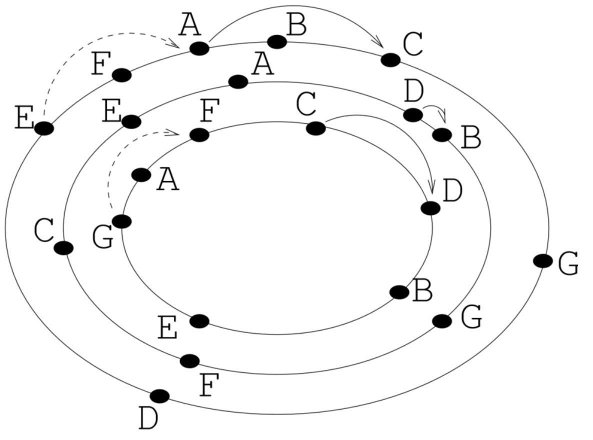

Fig. 5. An example system with 7 nodes and 3 rings. Solid arrows represent valid accusations, dashed arrows represent invalid ones.

A. Membership protocol

The membership protocol provides each member node with a weakly consistent view of the whole system. It relies on a broadcast channel through which nodes broadcast two types of messages:

- *Notes*, used to announce a node's presence, making other nodes insert that node in their view. Notes are also used for canceling false accusations (see below).
- *Accusations*, used to remove faulty nodes from the view of correct nodes.

All broadcast messages are signed and contain a public key certificate issued by a trusted authority. The broadcast channel itself is implemented using gossip (see Section IV-B).

The basic idea of the membership protocol is that nodes monitor each other via pings and broadcast accusations of failed nodes. An accused node is not directly removed from the views of other nodes. To countermand false accusations, the accused node is given a chance to broadcast a note proving its presence and canceling the accusation. A correct node only removes an accused node from its view if the accused node does not announce its presence within a certain time after being accused.

To limit the number of accusations and notes in the system, the nodes are organized in $2t+1$ rings. Each ring corresponds to a pseudo-random permutation of nodes. t is a system parameter expressing a measure of Byzantine fault tolerance. The system can tolerate t Byzantine nodes deterministically. More than t Byzantine nodes are tolerated with high probability as long as the overall fraction of Byzantine nodes remains small.

Node a can accuse node v only if, on some ring,

- a is the immediate predecessor of v, or
- all v's predecessors between v and a are accused.

Accusations not satisfying these conditions are discarded by correct nodes. Figure 5 depicts accusations in an example system with 7 nodes organized in 3 rings. Note that A's accusation of C on the outer ring is valid, because B is accused by D on the middle ring.

A correct node receiving a valid accusation of another node starts a timer. A node receiving an accusation of itself broadcasts a note to inform the other members about its presence, making them stop the timer and cancel the accusation.

In order to prevent a ping-pong of repeated accusation – note broadcasts due to Byzantine nodes, the note contains a bitmap of $2t+1$ bits, one for each ring. A node may *disable* a ring by broadcasting a note with the corresponding bit cleared. Accusations on rings disabled by the accused node are ignored. To prevent Byzantine nodes from becoming "immortal" by disabling too many rings, a node is allowed to disable at most t rings at a time.

B. Gossip

The broadcast channel used by nodes to disseminate notes and accusations is based on gossip over a pseudo-random mesh. Gossiped messages are signed and since the nodes have a global view of the system, they can verify the authenticity of each message. The impact of Byzantine nodes on broadcast is thus limited to decreasing performance. They can do so by not relaying gossiped messages and by initiating as many gossips as possible, wasting the bandwidth of correct nodes.

In order to reduce the impact of Byzantine nodes on the gossip protocol, Fireflies limits the choice of possible gossip partners for each node. The basic principle is similar to the one used in FlightPath [4]: using verifiable pseudo-random selection. A node only accepts connections from those nodes that are allowed to contact it.

Since a pseudo-random structure used for reasoning about valid accusations is already present, Fireflies reuses this structure for gossip. Node v can only gossip with node u if u is the successor of v on some ring.

Due to churn, it might happen that v believes u to be its successor, but u does not see v as its predecessor any more. In this case, if v tries to gossip with u, u informs v about v's proper gossip partner.

C. Evaluation

Fireflies was evaluated in simulation and on the PlanetLab platform.

Simulation was used to measure the message dissemination time needed by the broadcast channel in presence of various fractions of Byzantine nodes (Figure 6). We can see that even for large systems (10000 nodes) and large fractions of Byzantine nodes (25%), the dissemination time stays in the order of tens of rounds. Fireflies thus proves to be suitable for efficient data dissemination in a Byzantine environment.

The experiment on PlanetLab evaluates the behavior of Fireflies with 270 nodes when an important fraction of nodes (80) suddenly leave the system. The results (not shown here) suggest that Fireflies is robust against sudden node departures and can handle them with reasonable bandwidth overhead.

The drawbacks of Fireflies lie in its limited scalability. Every node having a full view of the whole system causes memory requirements to grow linearly with system size. Even if memory was not the problem (nowadays, memory is cheap), the rate of notes and accusations can be expected to grow linearly with system size. This is a more serious problem, since

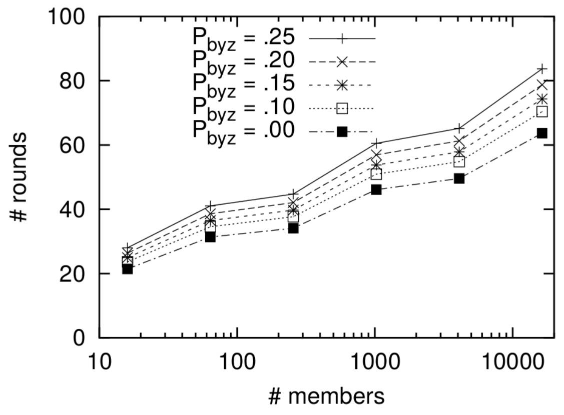

Fig. 6. Number of gossip rounds it takes to disseminate a message using Fireflies' gossip-based broadcast channel, for various fractions of Byzantine nodes.

every node has to receive and process notes and accusations from all other nodes.

By relying on a globally trusted certificate authority, Fireflies is also not fully decentralized. This issue could be circumvented by each node storing the public key of every other node as part of the node's identity. However, the price to pay would be increased bandwidth and memory requirements for saving and disseminating the public keys.

V. Research Proposal

In this section we summarize the strong and weak points of gossip-based systems in the context of the work just presented. We also give a sketch of *Robocop*, a system to address the identified weak points.

A. Summary of current issues

Gossip is inherently robust to simple crash failures of both nodes and links. Gossiped messages are automatically routed around failures thanks to gossip's randomized communication pattern. Gossip algorithms are fully decentralized, fast and efficient, scale well and easily cope with churn.

However, keeping all those properties in presence of Byzantine faults is not trivial. The presented gossip-based systems all achieve Byzantine fault tolerance, but none of them is able to maintain all the other desirable properties of gossip.

Path verification protocols remain fully decentralized and (although not explicitly stated by their authors) seem to be easily extensible to handle churn. They do not rely on cryptographic primitives and scale well if the number of Byzantine nodes remains low.

In practice, however, the number of Byzantine failures is usually proportional to the system size. The performance of path verification protocols degrades linearly with the absolute number of tolerated Byzantine failures. Thus, in practice, if a constant fraction of Byzantine nodes is assumed, the performance also degrades linearly with the size of the system.

FlightPath is fast, efficient, and seamlessly handles churn, but it sacrifices decentralized operation and scalability. The centralized tracker imposes a possible scalability bottleneck as well as a single point of failure (although the latter can be eliminated by replicating the tracker). Further scalability issues may arise from disseminating the full membership list to all nodes of the system.

Finally, *Fireflies* achieves high churn tolerance and fast, efficient and reliable data dissemination. By saving full membership view at each node and broadcasting notes and accusations, it sacrifices scalability. It is also not fully decentralized, since it relies on a trusted certification authority.

B. Robocop: ROBust grOup COmPutation

Robocop is a framework for gossip-based group communication and membership. It is targeting the issues identified above. Its aim is to provide Byzantine fault tolerance in gossip, while minimizing impact on the desirable properties of gossip.

1) Basic concept: The basic idea behind *Robocop* is to combine gossip with state machine replication. We partition nodes of the system into small clusters, each acting as a state machine replication group. The nodes in a cluster execute a highly robust protocol like PBFT [13]. A cluster can then be seen as a single reliable node of an abstract system. The abstract system executes a gossip protocol on top of the reliable nodes, not having to care about Byzantine failures, since those are masked by the clusters.

2) Keeping clusters "healthy": In order for the state machine replication groups (i.e., clusters) to function correctly, we must bound the fraction of faulty nodes in each cluster. A cluster corrupted by a majority of Byzantine nodes may compromise the whole system.

We take a probabilistic approach here. If the nodes constituting a cluster are picked uniformly at random, the fraction of faulty nodes is likely to be close to the overall fraction of faulty nodes in the whole system. However, as Rodrigues et al. point out [15], with many small clusters in the system, it is still likely that at least one of them will be corrupted.

The probability of cluster corruption can be decreased by increasing the cluster size (in terms of number of nodes per cluster). But given that state machine replication is generally resource-hungry and poorly scalable, it is necessary to keep the clusters as small as possible. Fortunately, the result of Guerraoui et al. [16] proves that logarithmic[6] cluster sizes are sufficient to keep all clusters uncorrupted with high probability.

Ensuring that the nodes in a cluster are sampled uniformly from the whole system is a non-trivial task in presence of churn. Byzantine nodes could develop strategies of repeatedly joining and leaving the system until they succeed to corrupt a cluster. Therefore, after every join or leave event, Robocop reshuffles the nodes of the corresponding cluster. It does so by exchanging the cluster's nodes for nodes randomly picked from other clusters.

3) Interconnecting clusters: Clusters are interconnected through an overlay network that has the form of an H-graph [17]. An H-graph is a sparse multigraph consisting of a union of Hamiltonian cycles. It is an expander with high probability and is thus suitable to be used for two purposes:

[6]Logarithmic in terms of the total number of nodes in the system

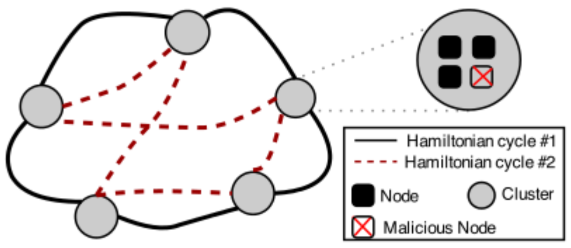

Fig. 7. A schematic representation of the Robocop system. Clusters of nodes are interconnected by an overlay network that has the form of an H-graph.

- Gossip among clusters, and
- Sampling of clusters using random walks (needed for the above-mentioned reshuffling).

Moreover, the overlay is scalable, because global knowledge of the whole system is not required. Nodes only need to be aware of their neighbors in the overlay. Handling churn is fully distributed and requires no broadcast or any other actions involving the whole system. A schematic representation of Robocop is depicted in Figure 7.

4) Contribution and future work: Robocop is the first system to combine state machine replication with gossip while achieving all three of:

- Byzantine fault tolerance,
- scalability, and
- tolerance to churn.

However, in its current state, Robocop relies on several strong assumptions, such as synchronized clocks on all nodes or absence of failures during system initialization. In the future, we want to modify Robocop such that we can relax those assumptions and make Robocop practically deployable in a wide range of environments.

References

[1] D. Malkhi, Y. Mansour, and M. K. Reiter, “On diffusing updates in a byzantine environment,” in *SRDS*, 1999, pp. 134–143.

[2] A. J. Demers, D. H. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. E. Sturgis, D. C. Swinehart, and D. B. Terry, “Epidemic algorithms for replicated database maintenance,” *Operating Systems Review*, vol. 22, no. 1, 1988.

[3] H. D. Johansen, A. Allavena, and R. van Renesse, “Fireflies: scalable support for intrusion-tolerant network overlays,” in *EuroSys*, Y. Berbers and W. Zwaenepoel, Eds. ACM, 2006, pp. 3–13.

[4] H. Li, A. Clement, M. Marchetti, and M. Kapritsos, “FlightPath: Obedience vs. Choice in Cooperative Services.” in *OSDI*, 2008.

[5] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin, “Bar gossip,” in *OSDI*, B. N. Bershad and J. C. Mogul, Eds. USENIX Association, 2006, pp. 191–204.

[6] S. Voulgaris, D. Gavidia, and M. van Steen, “Cyclon: Inexpensive membership management for unstructured p2p overlays,” *J. Network Syst. Manage.*, vol. 13, no. 2, 2005.

[7] A.-M. Kermarrec and M. van Steen, “Gossiping in distributed systems,” *Operating Systems Review*, vol. 41, no. 5, 2007.

[8] K. P. Birman, M. Hayden, Ö. Özkasap, Z. Xiao, M. Budiu, and Y. Minsky, “Bimodal multicast,” *ACM Trans. Comput. Syst.*, vol. 17, no. 2, 1999.

[9] J. Leitão, J. Pereira, and L. Rodrigues, “Epidemic broadcast trees,” in *SRDS*. IEEE Computer Society, 2007.

[10] L. Alvisi, J. Doumen, R. Guerraoui, B. Koldehofe, H. C. Li, R. van Renesse, and G. Trédan, “How robust are gossip-based communication protocols?” *Operating Systems Review*, vol. 41, no. 5, 2007.

[11] Y. M. Minsky and F. B. Schneider, “Tolerating malicious gossip,” *Distributed Computing*, vol. 16, no. 1, 2003.

[12] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth, “Bar fault tolerance for cooperative services,” in *SOSP*, A. Herbert and K. P. Birman, Eds. ACM, 2005, pp. 45–58.

[13] M. Castro and B. Liskov, “Practical byzantine fault tolerance and proactive recovery,” *ACM Transactions on Computer Systems*, vol. 20, no. 4, Nov. 2002.

[14] L. Lamport, “The part-time parliament,” *ACM TOCS*, vol. 16, no. 2, 1998.

[15] R. Rodrigues, P. Kouznetsov, and B. Bhattacharjee, “Large-scale byzantine fault tolerance: Safe but not always live,” in *Proceedings of the 3rd Workshop on on Hot Topics in System Dependability*, ser. HotDep’07. Berkeley, CA, USA: USENIX Association, 2007.

[16] R. Guerraoui, F. Huc, and A.-M. Kermarrec, “Highly dynamic distributed computing with byzantine failures,” in *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, ser. PODC ’13. New York, NY, USA: ACM, 2013, pp. 176–183.

[17] C. Law and K.-Y. Siu, “Distributed construction of random expander networks,” in *INFOCOM*, 2003.