RAMBLE: Reliable Asynchronous Messaging for Byzantine Linked Entities

Mohammad Imam
noahi@stanford.edu

Sahil Takiar
stakiar@stanford.edu

Jingkui Wang
jingkui@stanford.edu

***Abstract*—The RAMBLE protocol provides a decentralized, censorship resistant, Twitter style, public messaging system. Unlike blockchain based messaging which requires expensive proof of work computations, RAMBLE achieves eventual consensus through explicit reconciliation using conflict-free data types. RAMBLE utilizes established distributed P2P techniques, such as gossip and epidemic dispersion, which provide a precise threat model proving high probability of message delivery. Additionally, this paper explores Byzantine failure mitigation techniques and provides scalability and performance analysis of our implementation of the protocol.**

I. Introduction

An effective decentralized messaging service needs high availability, censorship resistance, and fault tolerance. Distributed Byzantine fault tolerant (BFT) systems [1], have demonstrated that public ledgers can be used to store arbitrary data. In order to provide strict data safety, implementations of such systems, such as blockchain, require multiple confirmations through expensive proof of work (PoW) calculations [2, 3]. Unlike stores of value that validate through ordered transactions, messages are inherently independent and can be effective without requiring strict data safety and ordering. By utilizing speculative confirmation (0-confidence), messages can be externalized faster without guaranteeing consensus. In the event a message is lost, it can be rebroadcast without causing conflicting outcomes.

We discuss an architecture to build a highly available, BFT, low latency, censorship resistant message ledger. This system includes mutable asynchronous digest blocks and a message datastore. The digest block records a count of confirmations for a particular set of committed message digests. The datastore maps digests to their message content. Independent messages, stored as conflict-free replicated data types (CRDT), allow consistent set reconciliation without requiring external coordination [4]. Backlog messages can be stored temporarily and deleted if they are not confirmed within a certain threshold period (described later in this paper). Messages are idempotent and can be rebroadcast without any adverse impact. Message storage relies on the assumption that a large enough subset of participating nodes will altruistically store data. Since message storage is not enforced by the network, failure by clients to store message data violates strict data safety.

Based on pageview and submission stats reported by Reddit in 2015, the ratio of reads to writes is approximately 100:1. In our protocol, each message is limited to 255 characters and is at most 512 bytes with metadata. If we extrapolate the Reddit model, 200 million users would share 700 million messages per year which would require close to 360 GB of [5]. For a single month, only 30 GB of storage would be required. By optimizing data storage using sharding with 1000x replicas, distributing among 200 million users would result in 1.8 MB of storage per user. Assuming only 10% of users are altruistic, it is still less than 20 MB of storage per user.

The popularity of peer-to-peer (P2P) systems, such as BitTorrent, has shown that access to idle compute power and storage capacity between 100 GB to 1000 GB is commonly available, and that end users are willing to donate these resources for a system they are invested in. However, network connectivity and bandwidth are not as abundant and reliable. By utilizing idle compute power and data storage, and optimizing for sparse network connectivity, we demonstrate a censorship resistant decentralized distributed forum, tolerant of Byzantine failures without expensive PoW requirements.

We present Reliable Asynchronous Messaging for Byzantine Linked Entities (RAMBLE), a censorship resistant distributed messaging protocol. Centralized forums, such as Twitter and Reddit, are subject to censorship and can experience availability issues such as 503 errors during peak times. Smaller forum communities, often running on a single server, are subject to DDoS attacks and are not reliable during host disruptions. RAMBLE prioritizes high availability and fault tolerance by replicating message state on each participating client and relaxing data safety. Although safety is crucial for blockchain transactions containing financial information, a messaging system can continue operating having lost a small percentage messages without disrupting most users.

Section II of the paper discusses the design and implementation of RAMBLE, including our anti-entropy algorithm and message dissemination protocol. Section III describes a precise threat model that bounds the probability that messages are successfully delivered to non-malicious clients, as well as how RAMBLE guards against various Byzantine failures. In Section IV, we present our performance evaluation and show that RAMBLE's protocols work in traditional P2P settings. Section V and VI describe future and related work, and we summarize our findings in Section VII.

II. Design and Implementation

The RAMBLE protocol utilizes a locally stored database, a node membership system, a gossip-based epidemic protocol, and an anti-entropy mechanism. A traditional anti-entropy implementation applies the following steps: 1) A client periodically pairs with another client, 2) the clients exchange history and reconcile differences, and 3) clients terminate communication at which point they both reflect the same set of data [15]. We chose a hybrid approach involving an optimistic epidemic dissemination protocol and an infrequent periodic anti-entropy system to ensure eventual consistency. Epidemic dissemination is ideal for sharing a limited data set, such as new messages, and provides a bounded worst-case network load. Anti-entropy is ideal for comparing history and reconciling CRDT sets.

Considering that conversations are temporal in nature, the message schema preserves a weak temporal ordering. Although the protocol itself is asynchronous, each message contains a timestamp set by the sender so that clients can better visualize messages and confirm basic sanity checks. A network connected device has access to a time service that is within an hour of accuracy with respect to other nodes on the network. The messaging service verifies that new messages are not backdated past a 24 hour threshold and are not impossibly dated in the future. Additionally, there is a critical invariant check based on timestamps: a message reply cannot have a timestamp less than the timestamp of its parent digest. A parent digest is a reference to the parent message of a reply. If a parent digest is null, the message is considered a top level thread. In the event a message has a later timestamp then the current time, a client is expected to update their clock such that any reply is later than the parent or it should reject the message as invalid.

We define the following nomenclature:

- **Node**: A container running the RAMBLE software stack with an on disk embedded database
- **Client**: A frontend interface allowing user to post messages and view received messages
- **Fingerprint**: A unique client identifier composed of an IP address and public-key shared via a federated exchange
- **Message**: A signed data type of string data, timestamp, parent digest, and fingerprint
- **Fanout**: The number of random peers per message broadcast
- **Filter**: A set of rules which determine message or sender validity
- **Digest**: A hash taken over an unsigned message, the fingerprint of the sender, and the timestamp of the message

The key design choice in RAMBLE involves separating the peer-to-peer synchronization tasks as multiple asynchronous services. Each service provides an asynchronous queue to pipeline inputs and forward outputs to another service. To ensure data integrity, all messages are signed by the author, preventing a client from impersonating another client. This improves performance and reduces Byzantine attack vectors by limiting the scope of any single pairwise exchange.

When a node joins the network, it presents a unique fingerprint, determined by a mapping between its public-key and IP address. Nodes without any prior history utilize a bootstrap bulk-message protocol which allows a peer to forward their digest block history and contents of their local database to the new node. RAMBLE ensures data integrity by requiring that all messages are signed and verified using the sender's public-key. This ensures that messages cannot be modified by Byzantine actors without exposing their fingerprint on the message. RAMBLE utilizes fingerprinting in the various asynchronous services to determine whether a node exhibits Byzantine behavior.

Implementations Details: The RAMBLE implementation is written in Java and is publicly available on Github [12]. Users can interact with a RAMBLE client via a command line interface (CLI), a simple UI based on the Spark Web Framework, or as an embedded Java application. Network objects are efficiently serialized using Google Protobuf. All network communications are done using Netty, an asynchronous, event-driven network library for Java. Netty was chosen instead of a generic remote procedure call framework because it optimizes asynchronous network communication. All messages and metadata are stored locally in H2, an embedded Java SQL database. Apache Gossip is an open-source implementation of a generic gossip based protocol, mainly geared towards failure detectors. While it features integration for sharing CRDTs, the reconciliation implementation is inefficient. Thus, we decided to use it purely for RAMBLE's membership service. The core class in RAMBLE is composed of a set of services, each responsible for a different part of the protocol. These services are discussed below.

A. Bootstrap Protocol

When a new node starts up, it must specify a known-valid peer to connect to. The new node will connect to this peer and download all its messages as well as any metadata necessary for all the other services to run. Once this has complemented, the new node sets its status to 'UP' and joins the membership cluster via the initial peer. At this point, the node is fully able to interact will other peers in the cluster and can start posting its own messages.

B. Membership Service

After the bootstrapping protocol has completed, nodes must join a membership service so they can discover, and be discovered by, other RAMBLE peers. New nodes first share their public-key and IP address with a predetermined set of peers. One of the peers will respond with a set of other nodes available in the network. The membership service maintains an up-to-date list of reachable peers that can participate in the epidemic protocol. The membership list is shared using the Apache Gossip protocol. Apache

Gossip allows a group of nodes to discover and periodically check the liveliness of a cluster. In addition, the service includes metadata containing the public-key of the node. Liveness checks are done by receiving heartbeat updates from each peer. If a peer fails to broadcast a heartbeat within a threshold period, it is excluded from the peer list and must re-initiate membership. The failure detector used in Apache Gossip is based on accrual failure detectors [6]. All messages are sent via UDP.

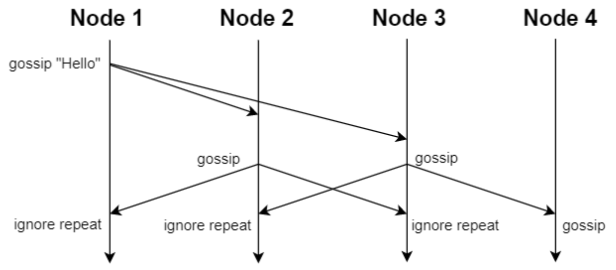

Fig. 1. Epidemic Broadcast Service with Fanout = 2. Node 1 posts the message “Hello” and broadcasts it to nodes 2 and 3. Both nodes 2 and 3 re-broadcast the message since they have not broadcasted the message yet. When node 1 receives the message again it drops it because it has already broadcasted it once before. A fanout = 2 means each node will broadcast a message to two random peers.

C. Epidemic Broadcast Service

Algorithm 1 demonstrates that unlike the periodic round based gossip protocol in Apache Gossip, the epidemic broadcasting service utilizes a message queue to simultaneously broadcast to multiple nodes. The Netty based peer broadcast service uses TCP to ensure that a peer receives a message without data corruption. When a client posts a new message or receives a message not previously broadcasted, it is added to the message queue. The broadcasting service reads from the queue and sends the message to a configurable number of peers. Once a message is successfully disseminated, its digest is added to an ignore list, terminating the epidemic gossip locally for that message.

Algorithm 1: Epidemic Broadcasting Service

```
1 while dequeue(message) do
2     for i = 1 to FANOUT do
3         do
4             Node n = get_random_peer()
5         while !broadcast_message(n)
6     end
7     enqueue_commit(message)
8     ignore_message(message.get_digest())
9 end
```

D. Anti-Entropy Service

Random peer selection for individual messages does not guarantee that every node will receive every message. Instead, the anti-entropy service supplements the epidemic broadcasting service by periodically verifying digest history and reconciling digest blocks. Digest blocks are

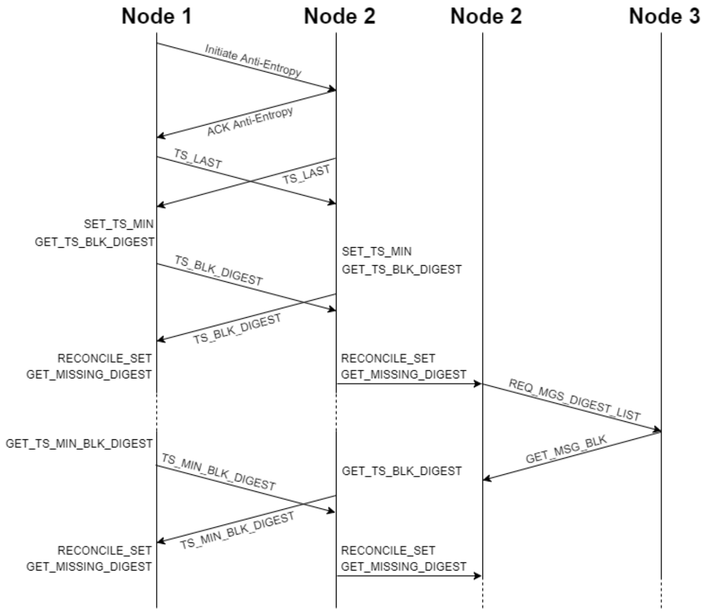

Fig. 2. Anti-Entropy Service: this figure shows a multi-threaded asynchronous anti-entropy implementation. Node 2 performs anti-entropy with Node 1 to determine the set of missing message digests for a particular timestamp range. The dotted lines show the algorithm repeating between the MAX and MIN TS range. The horizontal lines connecting the two ‘Node 2’ threads show message digests being sent to a queue and message data being served by Node 3. Node 2 can contact an arbitrary number of other nodes to reconcile the missing messages.

determined by querying a set of message digests between a predetermined range of timestamps. Each node keeps a count of the number of times a particular digest block has been verified. When this number passes a set threshold, messages mapped to the digests contained within the block are considered committed. Although the digest block is permanently mutable due to the nature of CRDTs, the client will not actively attempt to reconcile a block after it has been committed.

Algorithm 2: Anti-Entropy Service

```
1  Function anti_entropy(Node n, verified_ts_n)
2      ts_end = get_max_ts_span(verified_ts_n)
3      current_ts = get_current_ts()
4      while current_ts > ts_end do
5          set digests = get_digest_block(current_ts);
6          cache.put(digests);
7          send_block(n, digests)
8          current_ts = current_ts.get_next()
9      end
10
11 Function on_recv_block(block_ts, digests)
12     set digests = compute_complement(digests,
         cache.get(block_ts))
13     enqueue_message_needed(digests)
```

E. Client Filtering

Client-side filtering is done to protect against spam posted by malicious users (more details are discussed in the next section). The client filter blacklists a particular node if it has sent an excessive number of messages within a configurable timeframe. Nodes are identified by their fingerprint, which is a combination of its IP address and public key.

III. Threat Model

Byzantine fault tolerance introduces various attack vectors uncommon in centralized systems. In this section, we will explore various vulnerabilities we have identified and solutions to mitigate their impact. The RAMBLE design parameters are chosen based on a probabilistic approach that minimizes average expected failure such that 99.5% of messages disperse to at least a single non-faulty node when less than 50% of all nodes exhibit Byzantine behavior. Since the anti-entropy algorithm runs in perpetuity, as long as a single non-faulty node receives a message, it will continue to propagate to other non-faulty nodes. For N nodes, F Byzantine actors, and a fanout parameter C, we characterize the average expected value that a valid message will propagate to at least one non-faulty node.

$$\begin{aligned} prob(m) &= 1 - \frac{F}{N-1}\frac{F-1}{N-2}\cdots\frac{F-C+1}{N-C} \\ &= 1 - \prod_{i=1}^{C}\frac{F-C+1}{N-C} \\ F &= \frac{N}{2} - 1, \text{ for large N, } \approx \frac{1}{2}^{C} \\ \text{Choose C } &s.t.\ prob(m) > 0.995, \\ prob(m) &= 1 - \frac{1}{2}^{8} \\ &= 0.99609375,\ C = 8 \end{aligned}$$

Public forums do not have the same incentives as other BFT systems such as cryptocurrency. For most applications, there is no financial gain from attacking a messaging system. Although a coordinated attack could prevent a particular node from communicating messages with the rest of the network, it must be sustained in perpetuity otherwise the node can rebroadcast the messages and overcome censorship. Moreover, like any centralized forum, RAMBLE is susceptible to a coordinated set of messages that aim to portray a particular view or influence user opinions.

A. Protocol Dynamic Denial of Service

The APIs used to request messages, sync history, and perform anti-entropy tasks expose a dynamic denial of service (DDoS) attack vector. A coordinated set of Byzantine nodes can concentrate requests to a set of non-faulty nodes, starving the network stack of the affected nodes from from performing epidemic gossip and anti-entropy. To mitigate this attack vector, nodes enforce per fingerprint and global API rate limits. Nodes that are rejected by their pairwise member are encouraged to find a new random node to ensure progress.

B. False Data Flooding

Flooding spam, indistinguishable from a valid message, can ‘drown’ authentic messages from being seen. Spam is not unique to decentralized systems, large centralized sites such as Reddit and Twitter experience a high volume of automated accounts that create content based on trigger words. RAMBLE enforces client side filtering with a set of rules to prevent any particular node from spamming messages. The fingerprint restriction, which pairs an IP address with a public-key, has a significant deflationary impact on spam. Additional detection strategies, though not implemented, may including bayesian filters on keywords, machine learning on spam data sets, and clients manually reporting violating messages. Another example of a fake data attack can involve a faulty node withholding valid messages. Using the same probabilistic argument for fanout, we show that as long as a node can eventually engage in a pairwise exchange with a non-faulty node, they will eventually see all valid messages.

C. Sybil Attack

Sybil attacks rely on forging multiple identities possibly sharing the same IP address in a peer-to-peer network [7]. RAMBLE relies on the notion that IPV4 addresses are scarce compared to the number of non-faulty nodes participating in the system. The fingerprint system prevents mapping an arbitrary number of public-keys per IP address, thus allowing clients to mitigate the attack by banning bad actors. Additionally, since RAMBLE does not rely on voting, pseudonyms have limited influence on the network.

IV. Evaluation

A. Test Environment

The test setup includes three virtual machines across the U.S. Central, U.S. East and U.S West regions on the Google Compute Engine (GCE). Each VM is equipped with 24 virtual CPUs, 156 GB of memory and 10 GB of persistent storage. The VMs are configured with Ubuntu 16.04 and Docker CE to help simulate running multiple RAMBLE instances. A Docker Swarm, running a containerized RAMBLE instance, simulates a network of clients connected by an overlay network. During all tests, the ping latency between the Docker instances on the same VM is less than 1ms, while the latency between machines is 36ms. The containerized RAMBLE instance is based on Ubuntu and OpenJDK 8. The networking stack requires *iproute2*, *inotify-tools*, and *net-tools.* The Docker containers map to a shared volume where all messages, sent and received, are logged. We monitor individual client behavior by using *docker container attach* to interact with a particular client instance and to send out test messages. Each test includes a total of 96 RAMBLE instances which run a set of scripts that generates messages and collects logs from each VMs.

B. Test Results

Figure 3 shows a network delay test where we use *tc* to artificially add delay to network traffic. The test measures

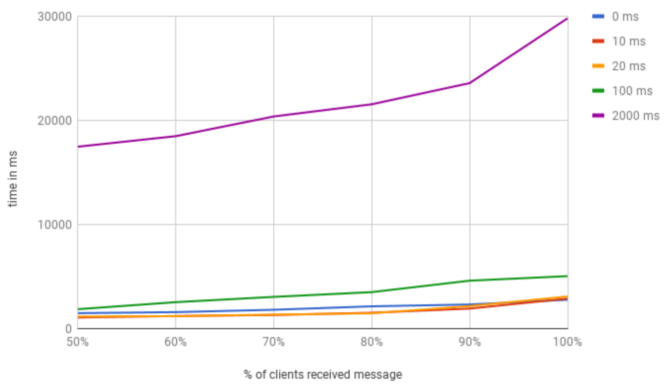

Fig. 3. A network delay test where network packet delivery between RAMBLE clients is delayed a variable amount

the time it takes for a message from a single client to propagate to all other clients. The horizontal axis marks the percentage of clients that have received the message and the vertical axis shows the duration in milliseconds. The data points on the curve show insignificant performance impact from a 10ms to 20ms network delay. The latency begins to grow when we scale the network delay to 100ms. Finally, when we set a really aggressive 2 second delay, we encountered a latency of around 30 seconds. The goal of this experiment is to analyze the performance of our epidemic dissemination protocol in the face of extended network delays, which can be common when working with asynchronous clients in P2P systems.

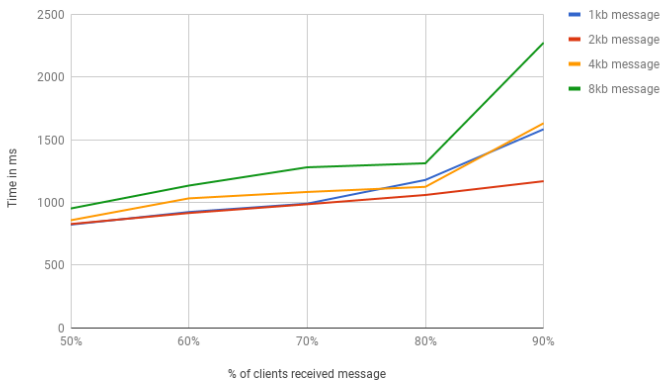

Fig. 4. A message size test where RAMBLE clients post message streams of variable sizes

Figure 4 shows the delivery latency for different message stream sizes. A message stream is a group of messages that are serialized and disseminated together. The results are similar to the network delay test. When the message stream is small (1k, 2k, 4k), we don't see much impact on the latency. When it increases to a total size of 8KB, the latency has a visible increase.

In the fanout test, we set the fanout parameter for the epidemic dissemination protocol to measure how long it takes for a certain percent of clients to receive all messages. As the results show, the latency of the two setups are

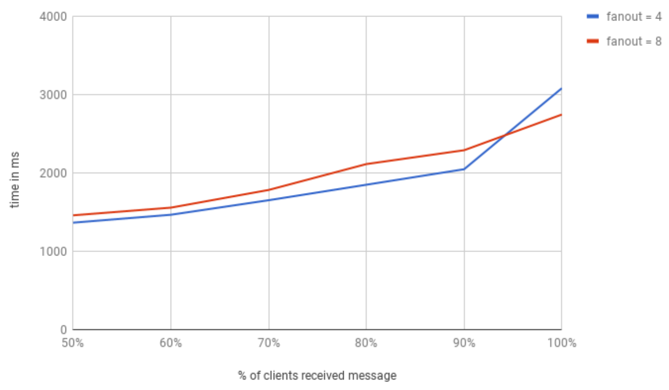

Fig. 5. Testing message propagation latency for different fanout values with no faulty nodes.

not significantly different. In fact, Fanout 4 shows faster propagation latency than Fanout 8 for a large portion of the message propagation lifetime. We believe this is due to network variance being greater than the confidence from using a limited number of RAMBLE clients. Since latency within the same machine is significantly smaller than machines in the other regions, the random selection also introduces significant variability in our measurements.

C. Analysis and Limitations

We evaluate RAMBLE based on its ability to scale, provide high availability, and deliver messages in a timely manner when operating in a P2P environment. These tests show that RAMBLE can operate over a WAN where nodes are geographically distributed across different data centers. Furthermore, the tests demonstrate that RAMBLE's message propagation is resistant to artificial network delays and large message payloads. Although we expect these results to scale with nodes count, the tests were only conducted on 96 RAMBLE instances.

The Google Cloud Engine limited the number of VMs we could use, which prevented our instances count from being orders of magnitude greater than our fanout size, which is needed to overcome variance introduced by the random epidemic dissemination protocol. We intended to spin up thousands of RAMBLE instances, but were limited by JVM and Docker resource consumption. To prevent overloading the vCPU, we limited our instance count to 96 which is insufficient for testing the impact of the fanout choice.

Since nodes sharing an individual VM share the same Docker volume, we noticed decreased write performance caused by multiple clients writing to H2 at the same time, creating disk resource contention. We expect that with a better storage solution, our high message size test should yield a significantly better performance matrix.

The timings reported in each test should not be taken as an accurate representation of real-world message delivery time. Our setup requires running an extensive number of Docker images inside a single virtual machine, which is not a realistic representation of how actual clients would use RAMBLE. However, we feel the tests are still useful to comparatively show the difference in message delivery latency when run on diverse networks, with variable message sizes, and with different fanout values.

V. Improvements

Network Routing Random peer selection introduces increased network latency. An overlapping set of localized clients would result in a more efficient dissemination protocol. A distributed DNS server similar to Coral DNS [8] would reduce round trip latency and improve overall system availability due to its load balancing features.

Storage RAMBLE stores all messages on a single machine instead of sharding across multiple nodes, limiting the number of messages that can be retained in the system. Storing data in a Distributed Hash Table (DHT) would work well in our current model. Nodes can only store the newest messages in their database and can selectively purge older messages that they are less interested in. This requires a routing implementation that allows clients to perform message lookups either by digest or by timestamp range. In addition, other storage optimizations such as archiving older messages in GZIP files should be explored.

Performance We hope to do further performance studies on RAMBLE using real-world workloads, such as Twitter's Streaming API [9]. Our performance numbers were limited by cost issues and resource limitations on the GCE infrastructure. We expect a more representative set of performance graphs as the number of clients in a cluster increase. Furthermore, we hope running with a higher number of instances will allow us to stress test other parts of the implementation, such as the efficacy of our anti-entropy algorithm and the scalability of Apache Gossip. One dimension we have not tested yet is RAMBLE's ability to withstand Byzantine clients and coordinated attacks.

Reputation System A reputation system could be built into RAMBLE to help distinguish between malicious and normal users. Reputation and trust management in P2P networks is an active area of research [13, 14], and many of the published algorithms can be extended to RAMBLE. Reputations can be assigned based the unique fingerprint that identifies each client. The system can help clients identify bots in the network that are attempting to flood spam to other nodes.

VI. Related Work

There have been a number of attempts to use blockchain technology to implement a decentralized messaging platform [3]. While these systems have stronger safety guarantees, they sacrifice latency due to excessive time spent on PoW. Most users of a messaging system expect near real-time (bounded latency) message propagation. Message commit delay of minutes, waiting for block confirmation, is not acceptable for a public forum.

Gossip is a widely studied protocol with a variety of use cases. We leverage work done on gossip algorithms for failure detectors [6, 10]. A closely related area of gossip protocols, is epidemic dissemination protocols that are used to quickly spread information from a given node [11]. Our broadcast service closely mirrors this style of protocol.

CRDTs are data containers that can reconcile differences and are guaranteed to provide eventual consensus given commutative concurrent updates [5]. We utilize CRDTs to share and reconcile message digest history sets. We define messages as independent and idempotent transactions to the overall state of the system, making them ideal for CRDTs.

VII. Conclusion

This paper has described the architecture and implementation of RAMBLE, a censorship resistant distributed public messaging system, utilizing a unique combination of gossip, epidemic dissemination, and anti-entropy protocols. We propose a formal threat model which defines the probability that a message will fail to propagate to any non-faulty node. Additionally, we provide analysis on various common Byzantine threats and comment on the level of safety in the system.

RAMBLE provides configurable and predictable bounded network throughput which scales with the number of nodes present in the system. We make the case for utilizing idle compute and altruistic message storage based on the wide usage of P2P networks such as BitTorrent. Our performance evaluation demonstrates that RAMBLE is suitable for operating across wide-area networks where nodes are geographically distributed and sparsely connected. Moreover, the results show that message dispersion latency scales effectively with node fanout indicating high availability.

References

[1] Miguel Castro et al. "Practical Byzantine Fault Tolerance". *OSDI*. 1999

[2] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". 2008

[3] Jonathan Warren. "Bitmessage: A Peer to Peer Message Authentication and Delivery System". 2012

[4] Reddit in 2015 https://redditblog.com/2015/12/31/reddit-in-2015/

[5] Marc Shapiro et al."Conflict-Free Replicated Data Types". *Stabilization, Safety, and Security of Distributed Systems*, 2011

[6] Naohiro Hayashibara et al. "The ϕ Accrual Failure Detector". *Japan Advanced Institute of Science and Technology.* 2004

[7] John R. Douceur. "The Sybil Attack". *IPTPS.* 2002

[8] Michael J. Freedman et al. "Democratizing content publication with coral". *NSDI.* 2004

[9] The Twitter API https://developer.twitter.com/

[10] Robbert van Renesse et al. "A Gossip-Style Failure Detection Service". *Proceedings of Middleware.* 1996

[11] Anne-Marie Kermarrec et al. "Reliable probabilistic communication in large-scale information dissemination systems". *Technical Report.* Microsoft Research. 2000

[12] Reference Implementation https://github.com/noah-/ramble

[13] Sepandar D. Kamvar et al. "The EigenTrust Algorithm for Reputation Management in P2P Networks." *ACM.* 2003

[14] Yao Wang et al. "Trust and reputation model in peer-to-peer networks." *IEEE.* 2003.

[15] Douglas B. Terry et al. "Managing update conflicts in Bayou, a weakly connected replicated storage system." *SIGOPS.* 1995.