Eggscrambler: Anonymous Broadcasting Using Commutative Encryption

6.824 Project Report

Arvid Lunnemark arvid@mit.edu
Arman Talkar atalkar@mit.edu
Wendy Wu wsw23@mit.edu

May 14, 2021

1 Introduction

We present a system to solve the problem of anonymous broadcasting among honest participants and malicious listeners. In anonymous broadcasting a set of users use a protocol to broadcast a message to every other user without revealing that the message they sent comes from them. After the protocol is done and the set of messages is published, it should be impossible to tie user i's message in this list to the identity of user i, even for an adversary that can monitor all network traffic.

2 Design

Our approach has two parts: an anonymous broadcasting protocol and a fault-tolerance layer. The anonymous broadcasting protocol relies on commutative encryption and is described in detail in section 6. As a summary, each peer (the application handling the broadcast on behalf of the user) encrypts a message submitted by the user, and sends it to all other peers participating in the broadcast round. All peers then encrypt the received messages so that all messages have been encrypted by all peers. Then the set of messages is passed to each peer in a ring. Each peer randomly shuffles the order of the messages and encrypts each message again. As long as all peers are honest, this shuffling step prevents all subsequent and prior peers from inferring which user submitted a particular message. After shuffling, the messages are again passed around a ring and decrypted (by virtue of commutative encryption), and then broadcast.

The fault tolerance layer uses an extended version of the Raft implementation developed in the labs. Since the protocol requires coordination between peers and requires that all peers fully participate in a round to broadcast a message, we implement the `AddServer` and `RemoveServer` RPCs specified in the Raft thesis. This allows users to join and leave the protocol at will. The broadcasting protocol cannot proceed if a peer failure occurs (whether it is caused by a network partition or the peer leaving the cluster or crashing). Upon such a failure, the fault-tolerance layer finds a new stable configuration before resuming a new round of the broadcast protocol.

The Raft layer is also used to coordinate the parts of the protocol that pass messages around in a ring by implementing a test-and-set primitive. The results of a peer's scramble or decryption will only be visible to the other peers if the state it operated on is still the latest state, forcing the peers to go one-after-another.

3 Implementation

The system architecture is shown in figure 1. Running on top of each Raft instance is a state machine server that tracks progress through each round of the protocol by consuming commands from the Raft log. Running on top of the state machine server is a state machine client that receives state updates from the server and, if it

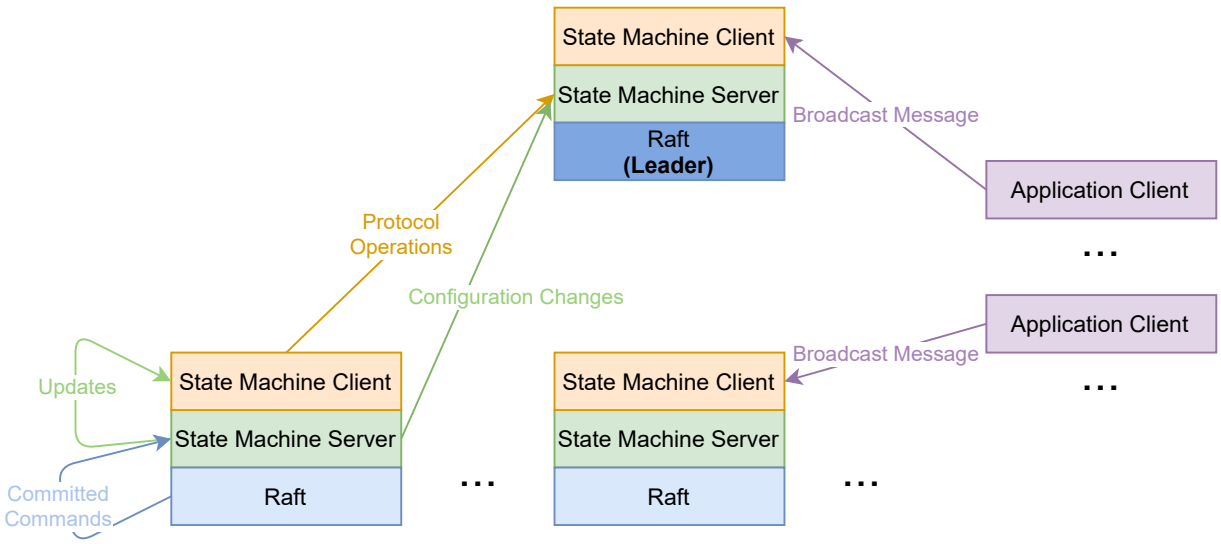

Figure 1: System architecture. Notably, the state machine client reads from its local server, and submits updates to the (possibly remote) leader server.

is active (a participant in the current round), issues requests to the leader server that progress the round. Each round an active client either uses a message provided by an application client, or uses an empty message — in both cases, the message is prefixed by a recognizable header including a random string.

3.1 Testing and Usage

The interface exposed by Eggscrambler and how it used by an application is illustrated in fig. 2. This interface abstracts away all of the internal protocol details that are necessary for anonymity (and are described in section 6).

To test our implementation, we used a modified version of the lab testing code. Our test suite consists of 8 tests that test the system in reliable and unreliable networks, with configuration changes, and in cases when the protocol is forced to abort. The test suite passes consistently with no races.

In addition to tests, we created a small application that interfaces with the anonymous broadcasting system to provide a fully anonymous version of *MIT Confessions*. Currently, the Confessions application runs on a single machine only, but in the coming week, we aim to extend it to support communication over IP using the real RPC library.

3.2 Configuration Changes

When a state machine client is initialized, it checks to see whether or not its underlying state machine server is part of the raft configuration or not. If it isn't, it first asks the server to join the configuration as a *provisional* server. In this status, the server receives all log entries from the leader but is not a voting member, allowing a server to catch-up its log before it is added. The leader applies provisional status by committing a log entry so that the status persists through leader change.

Once a leader receives a successful `AppendEntries` response from a provisional server, it can accept an `AddServer` RPC for it. Our implementation matches the description given in the Raft thesis. Once the state machine client determines its server is part of the configuration it can participate in the protocol.

When a state machine client wants to become inactive, it first checks if it is a participant in the current round and, if so, aborts the round. It then requests that its server make a `RemoveServer` RPC, again implemented as in the thesis. Once it determines it is no longer part of the configuration it may safely go offline.

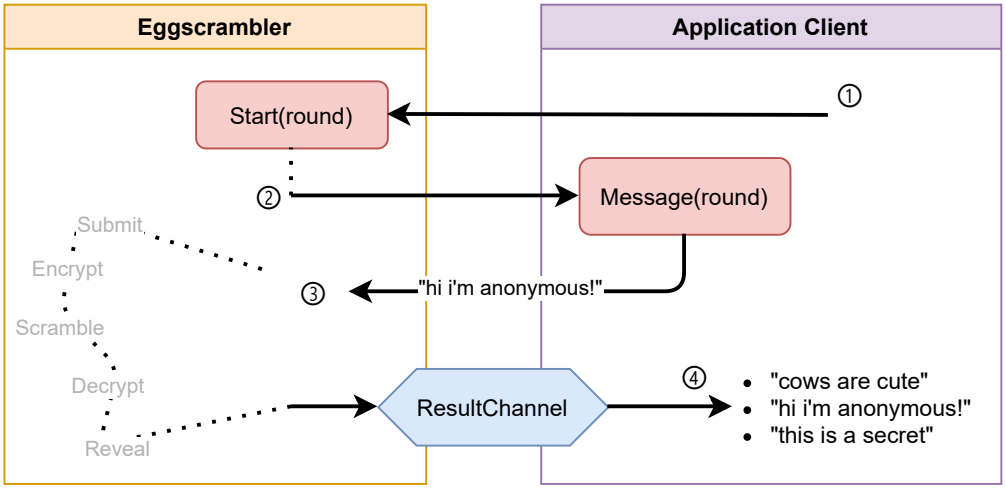

Figure 2: Interface between the anonymous broadcasting system and an application using it. One application client calls `Start(round)` (1), after which the system calls *all* application clients' `Message(round)` (2). The system proceeds through the phases of the protocol (3) and finally outputs the anonymized messages over the `ResultChannel` (4).

The Raft authors found a bug in the original presentation of `AddServer` and `RemoveServer` caused by leader failures during concurrent attempted changes. The fix is for a leader to commit a log entry in its own term before accepting an configuration change request. To ensure this happens, we have the leader attempt to commit a no-op command immediately upon election.

3.3 Commutative Encryption

Anonymity in our protocol requires a commutative encryption system; otherwise, the order of decryption would necessarily be the reverse of the order of encryption, which would reveal the original message sender's identity. Additionally, the commutative encryption must also remain secure even if multiple messages are encrypted using the same key, because in every round the message from each user is separately encrypted by all users. Thus, despite believing so in the beginning, it turned out that we were unable to use ElGamal commutative encryption, in which a user's private key can be calculated from the message prior to encryption with that user's key and after encryption by the user.

In the interest of maximal flexibility, we wrote a custom implementation of the Massey-Omura cryptosystem instead. Each user keeps private encryption and decryption keys, and these multiply to $1 \pmod{p-1}$, where p is a large prime that determines the field that encryption and decryption happens on. Massey-Omura is commutative as exponentiation is commutative. Additionally, Massey-Omura only requires that the discrete log problem is hard in the underlying group, which is the case for the Galois field of a large prime. Finally, owing to the exponentiation, it is sufficiently difficult to calculate a user's encryption and decryption keys even knowing the message both pre- and post- encryption by that user.

4 Safety and Fault Tolerance Properties

Under the assumption that the underlying commutative encryption scheme is secure, the system provides the following safety guarantee: **if a round result is sent on the `ResultChannel`, and the result indicates success, then the application can be certain that no one is or was able to link any identities to any of the broadcast messages**. If all participants are honest, the round result always indicates success, which means that this guarantee implies that the system preserves anonymity in the setting of honest participants and potentially malicious observers. The guarantee is stronger than that, however: even if there are some Byzantine participants (who could, for example, be the Raft leader), if they are able to break the anonymity,

everyone will know that something bad happened. Section 6.8 describes in detail how the system ensures this guarantee.

The fault-tolerance guarantees are the same as the Raft fault-tolerance guarantees - as long as a majority of the raft servers are online, the system can continue to function. The protocol cannot make progress amidst server failures, but with a connected majority `RemoveServer` RPCs can exclude failed servers and then resume the protocol.

5 Discussion

Neither our design nor our implementation focuses on performance. This means that the throughput of our system is on the order of a few rounds per second. Thus, this protocol would not be useful to anonymize all communications. While we settled on this protocol design using commutative encryption because it is a novel idea, it would be interesting to explore how our solution contrasts to, for example, DC-nets, and whether our system has any benefits over them.

The anonymity guarantee may seem relatively weak, but it should be noted that there is an interesting social component to it. If the system has been running for many rounds without any malicious behavior, one could relatively confidently assume that the next round will not have any malicious behavior. This crucially relies on the fact that we can detect malicious behavior. Nevertheless, it would be interesting to investigate whether the protocol can be modified to guarantee even stronger anonymity. Ideally, the protocol would always guarantee anonymity, even in the face of any number of Byzantine participants, with no caveats.

6 Appendix: Detailed Protocol

The replicated state machine keeps the following state:

- `Round`, a number specifying the current round of messages.
- `Rounds[r]`, an array of data about every round.

The round-specific data contains:

- `Phase`, the phase that the round is in, where the phase is one of `{PREPARE, SUBMIT, ENCRYPT, SCRAMBLE, DECRYPT, REVEAL, DONE, FAILED}`.
- `Crypto`, the parameters used for the commutative encryption scheme used in this round.
- `Participants[p]`, a list of UUIDs for every participant of this round.
- `Messages[p]`, the current values of the messages in this round.
- `Encrypted[p]`, whether or not a participant has encrypted.
- `Scrambled[p]`, whether or not a participant has scrambled.
- `Decrypted[p]`, whether or not a participant has decrypted.
- `RevealedKeys[p]`, the revealed key for each participant.
- `RevealKeyHashes[p]`, the hash of the revealed key for each participant.

Every round starts in the `PREPARE` phase, and ends in either the `DONE` or the `FAILED` phase. In any phase except the `PREPARE` phase, if the client has not received a state machine update for more than 10 seconds, it submits an `Abort` RPC which transitions the round to the `FAILED` phase and creates a new round in the `PREPARE` phase.

6.1 Prepare Phase

Each peer:

1. Generates a local `encryptKey`.
2. Generates a local `scrambleKey`.
3. Generates a local `revealKey`.
4. Generates a UUID and modifies the state machine by appending it to `Participants`.

The state machine transitions to the `SUBMIT` phase when at least one participant calls a special `Submit(round)` RPC. The peer that submits also generates a big prime and modifies the state machine by setting `Crypto` to the big prime.

6.2 Submit Phase

Each peer:

1. Gets a message `msg` from the user application, or if the user application has no message, uses a default message. All messages have the same length and a header, where the header consists of 8 bytes with a constant application-defined value, followed by 8 completely random bytes.
2. Encrypts `msg` once using the `encryptKey`.
3. Modifies the state machine by setting `Messages[me]` to the encrypted message.
4. Modifies the state machine by setting `RevealKeyHashes[me]` to a SHA256 hash of the `revealKey`.

The shared state machine transitions to the `ENCRYPT` phase when `Messages[p]` contains a value for each peer.

6.3 Encrypt Phase

This phase proceeds in a test-and-set manner.

Each peer:

1. Gets a copy of the state machine's `Messages` list.
2. Encrypts each message, except its own, once with `encryptKey`.
3. Modifies the state machine's `Messages` list to the newly encrypted list, with a test-and-set on the number of participants that have encrypted before this participant.
4. Sets `Encrypted[me]` to true.

The shared state machine transitions to the `SCRAMBLE` phase when `Encrypted[p]` is true for each peer.

6.4 Scramble Phase

The scramble phase does not use test-and-set, which is important for the anonymity guarantee.

Each peer:

1. Waits until `Scrambled[p]` is true for all `p < me`.

2. Gets a copy of the state machine's `Messages` list.
3. Scrambles it (i.e., picks a random reordering of it).
4. Encrypts each message once with `scrambleKey` and once with `revealKey`.
5. Modifies the state machine's `Messages` list to this scrambled and encrypted list, atomically.
6. Sets `Scrambled[me]` to true.

The shared state machine transitions to the `DECRYPT` phase when `Scrambled[p]` is true for each peer.

6.5 Decrypt Phase

This phase uses test-and-set, similarly to the encrypt phase.

Each peer:

1. Gets a copy of the state machine's `Messages` list.
2. Decrypts each message once with `scrambleKey` and once with `encryptKey`.
3. Modifies the state machine's `Messages` list to the newly decrypted list, with a test-and-set on the number of participants that have encrypted before this participant.
4. Sets `Decrypted[me]` to true.

The shared state machine transitions to the `REVEAL` phase when `Decrypted[p]` is true for each peer.

6.6 Reveal Phase

Each peer:

1. Modifies the state machine by setting `RevealedKeys[me] = revealKey`.

The shared state machine transitions to the `DONE` phase when it has received a revealed key from all peers. When that happens, the `Round` is also incremented, and a new round in initialized to the `PREPARE` is added to the `Rounds` list.

6.7 Done Phase

When a round is in this phase, in the ideal case, `Messages` contains messages that, if decrypted once with each key in `RevealedKeys`, is the plaintext of a message that was sent by some user.

Before decrypting and sending the messages to the user application, the peer needs to verify that everything has happened correctly. In particular, each peer:

1. Verifies that the `Messages` list here is the same length as it was before the `SCRAMBLE` phase.
2. Verifies that the first 8 bytes of the final value of each message in the `Messages` list contains the application-specified constant value.
3. Verifies that the second 8 bytes of the final value of each message in the `Messages` list are not the same as those 8 bytes in another message in the `Messages` list.
4. Verifies that the `RevealKeyHashes` submitted before the reveal phase are exactly the SHA256 hashes of the `RevealedKeys`.

If these verification steps all pass, the peer decrypts the messages and forwards them to the user application on a result channel. Otherwise, the peer tells the user application that malicious behavior was detected, and that anonymity may have been compromised.

6.8 Safety Guarantee

The main idea is that after a peer has scrambled the messages, it is impossible for anyone to connect a message in the scrambled list to the corresponding message in the pre-scrambled list. This seems intuitively to be true, given that a participant secretly chooses a random reordering, and encrypts each message using a secret key which makes it hard for anyone else to determine the order.

However, it turns out that it is not entirely true that no one else can determine any relation between the scrambled and the pre-scrambled list. In particular, an attacker could insert message m and message $E(m)$ in the pre-scrambled list, where $E(m)$ is the message m encrypted with a secret key, and then after the victim has scrambled, the attacker can iterate over the scrambled list, trying $D(m')$ on each message m', eventually yielding a duplicate value. This tells the attacker which message m corresponds to in the scrambled list, which means that we do not get complete anonymity.

Yet, with the verifications in the `DONE` phase, we can detect such attacks. That is, we want to convince ourselves that if all verification steps pass, then anonymity is guaranteed. The key piece to that anonymity is that, if there are n participants in a round, a peer encrypts exactly n messages using its `revealKey`. This means that if an attacker chooses to inject a fake "tracker" message as above, it will be impossible to end up at a state where, before the `REVEAL` phase, all original messages are encrypted exactly once by each reveal key. The unique header verifications only pass if that is the case, though, which means that we detect such attacks! The reveal key hash commit is necessary because the attacker may otherwise be able to modify its own reveal key by combining it with its attacker key and the attacked user's reveal key.