Distributed Algorithms - Final Project
(Anonymous chat)

March 7, 2021

1 Project overview

The goal of the project is to build a system that enables any pair of clients to privately chat in a way that maintains their anonymity and message privacy relative to all other clients and servers.

No party (client or server) should learn anything about the contents of the messages exchanged between 2 non-colluding clients, nor should it learn anything about the mere fact that the 2 clients are talking with each other.

Given certain assumptions, a passive adversary cannot learn anything either.

1.1 Environment, setting, assumptions

Model - We assume a synchronous communication model. For non-omissioned parties, messages are delivered after at most some finite time bound Δ.

Servers - There are n servers, out of which at most $f < \lceil \frac{n}{2} \rceil$ can have omission failures.

Clients - There are m clients, out of which at most $g < \lceil \frac{m}{2} \rceil$ can have omission failures.

Public-Key Infrastructure - We assume a pre-existing PKI, that allows all parties to create encryption keys publicize their public keys to all other parties. Practically speaking, this means our project assumes the public keys of all parties are already known.

Channels - Channels are just logical partitioning. There are d channels.

We also define:
$THRESHOLD = f + 1$

1.2 Goals and requirements

1. **Anonymity** - Given two clients chatting, no other client or server can see (or learn anything) that the two parties are exchanging messages.

2. **Privacy** - Given two clients chatting, no other client or server can see (or learn anything) the content of the messages exchanged.

3. **Liveness** - Given two non-faulty (no omission failures) clients A and B: if A wants to send a private message to B, he is guaranteed (eventual) success.

4. **Safety** - All clients receive/learn of the same messages.

An attempt at more formal definitions, based on those that were given in theoretical exercise set 2:

1. **Validity of Share** - if a non-faulty client calls $Share(s,d)$, then s is a shared value on channel d.

2. **Liveness of Share** - if a non-faulty client callls $Share(s,d)$, then all non-faulty replicas will complete *Share* with success.

3. **Safety of Share** - all non-faulty replicas agree on the outcome of share: success or failure.

4. **Privacy of Share** - if no party calls *Recover*, then the adversary (that is controlling a set of at most f faulty replicas) learns nothing about any of the shared values.

5. **Anonymity of Share** - if client A shares a secret that is a private message for client B, then the adversary (controlling at most f faulty replicas and g faulty clients) learns nothing about the identities of the sending or receiving clients.

6. **Validity of Recover** - any successful *Share* will be included in the output of the recovery of any non-faulty replica

7. **Liveness of Recover** - any attempt to *Recover* will be termianted by all non-faulty replicas

8. **Safety** - all clients that learn of a message (or batch of messages) learn of the same message

2 Solution overview

Solution in brief

This a quick run-down. The aim is to convey the main idea. It skips some details for the sake simplicity and clarity (which are of course given throughout later sections)

Client *A* wants to send a message to client *B*. He encrypts the message with client *B*'s public key, choses a random channel (uniformly) over which to send the message. Client *A* then does secret-sharing with the servers, with the chosen channel's secret being the encrypted message and all other channels having noise/empty messages.

Instead of sending servers their share only, the client encrypts each share with the appropriate server's public key and sends **all** encrypted shares to **all** servers.

Servers replicate the encrypted shares amongst themselves via Raft.

After a minimal amount of messages (from different clients) have been received, the servers go on to recover the messages: each server decrypts its shares, performs secret share addition and sends the resulting shares (1 for each channel) to all other servers.

Once a server has enough shares, it can recover the messages on all channels and push them to the clients.

Upon recieving messages from servers, clients attempt to decrypt each message with their private key. If the message was sent to them, decrypting with their private key will uncover the message sent to them.

At this point it, the use of asymmetric encryption/PKI for two critical operations should have been noted:

1. To ensure only the reciever/recipient client (for a given message) can decipher the message sent.

2. To easily replicate a client's secret sharing amongst all servers, without having to deal with intacacies of server-omissions while maintaining secrecy.

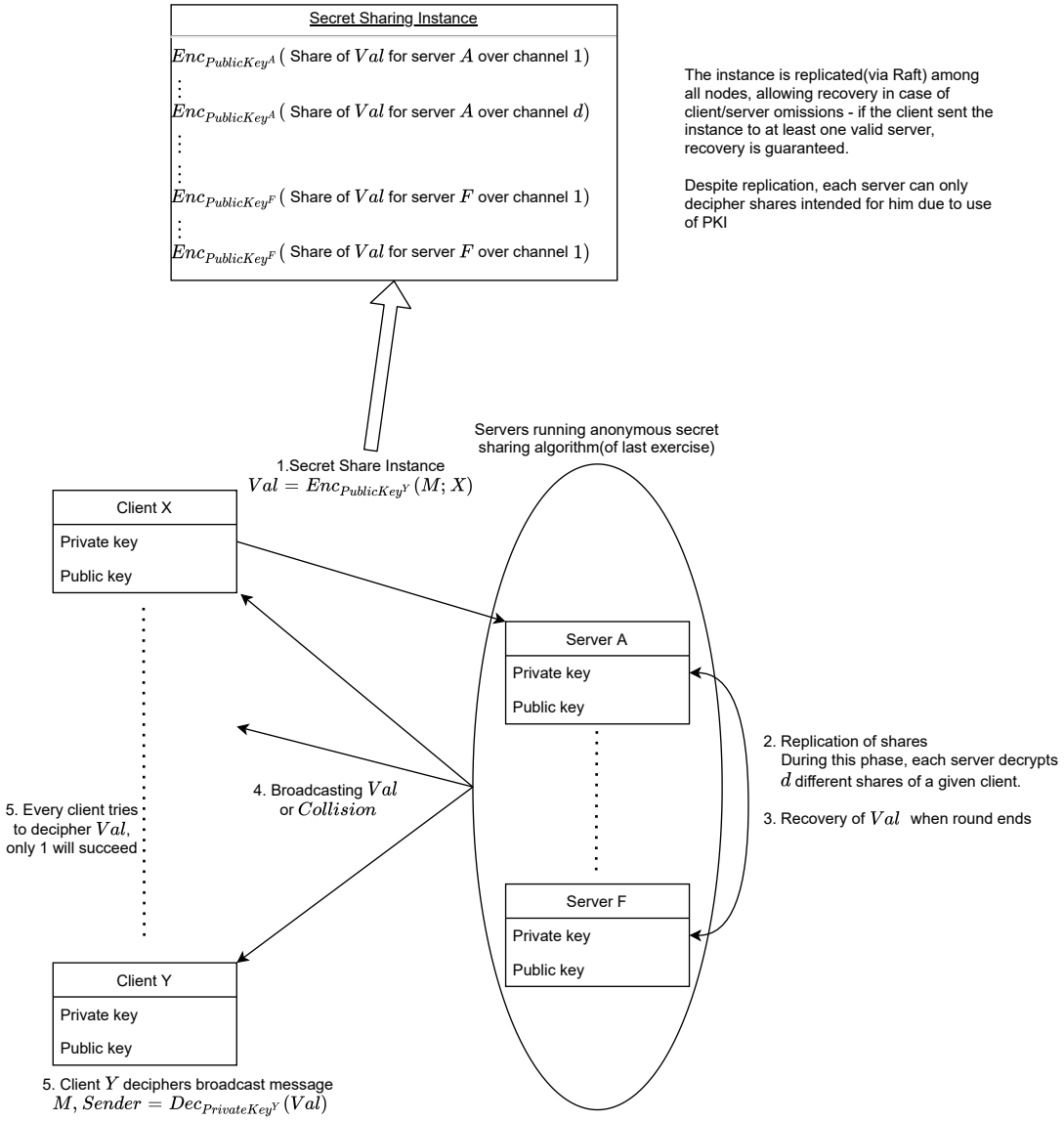
2.1 High-Level protocol

There are way to many implementation details, tentative changes and nuance than can be comfortably covered in this document. Using the word “protocol” is slightly misleading - the following sectionis meant to convey the main points and ideas for our solution in an intuitive manner, not to define the exact 1-to-1 schemes and formatting that the implementation uses.

The following diagram shows depicts the high-level idea, afterwhich we’ll do a few “zoom-in”s:

Anonymous Chat Architecture

Sending a message M from client X to client Y



2.1.1 Sending messages

To send a message, the client choses a channel (uniformly) where the message will be sent. All other channels will have an secret (all zero bits).

The client prepares the message for sharing and encrypts it with the recipients public key. Then, the client creates secret shares for all channels/messages, encrypts each share with the appropriate server's encryption key, and finally sends all servers a copy of all of the encrypted shares.

Message prep

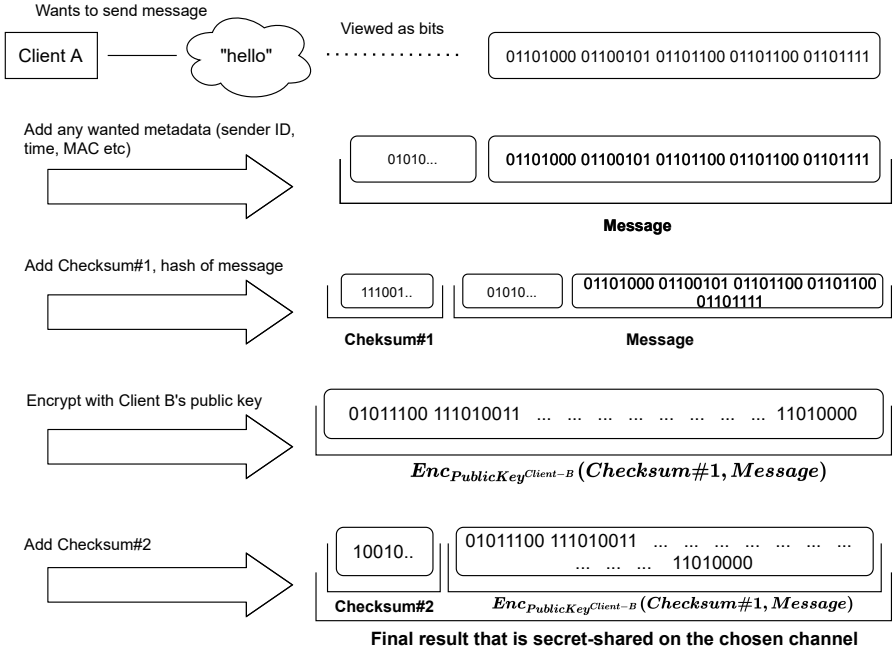
Suppose client *A* wants to send a message, "Hello", to client *B*. To acheive privacy, the message has to be prepared and then encrypted with the recipient's public key.

After adding any wanted metadata such as the sender's ID (so the recipient knows where to send a response), timestamp (just nice, also could be used to prevent replay attacks) - the first step is to pad(or trim) the message to reach a pre-defined size. (This size can be dervied by working backwards from the final prepared message size, to guarantee that final prepared message's size won't exceed the pre-defined, arbitrary, maximum size for a secret).

Then, a *Checksum*#1 of the padded message is concatenated. (Later, this Checksum will be used by client *B* to ensure it has decrypted a valid message).

The result is encrypted with client *B*'s public key, and *Checksum*#2 of the encrypted result is concatenated. (Later, this Checksum will be used by the servers to detect collisions - messages sent over the same channel).

This final $Checksum\#2 + EncryptedMessage$ is the secret that the client shares with the servers. (this is actually done by secret-sharing smaller chunks of the secret).



Encrypted secret shares

To share a secret with the servers, the client creates shares

$$P(0), P(1), ..., P(n)$$

with a threshold of $f+1$ using Shamir's Secret Sharing algorithm.

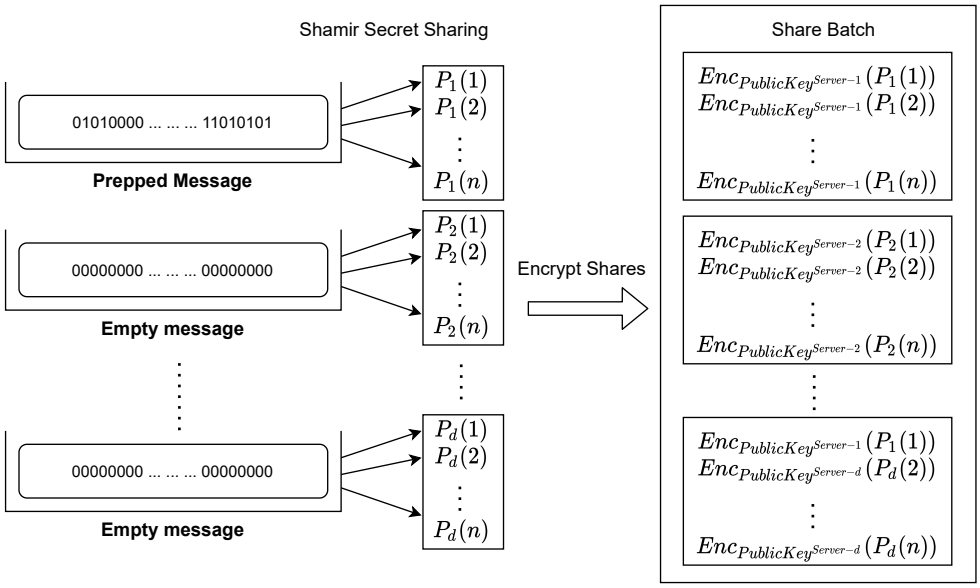
Instead of sending each share to only the appropriate server - the client encrypts every share with the corresponding server's public key and sends **all** encrypted shares to **all** servers.

Denote $Encrypted(j) = Enc_{PublicKey^{Server-i}}(P(j))$. For a single share, the client would send

$$[Encrypted(0), Encrypted(1), ..., Encrypted(n)]$$

to all servers.

Since we have d channels the client will be sending all d of these encrypted share "pack"s to all servers.



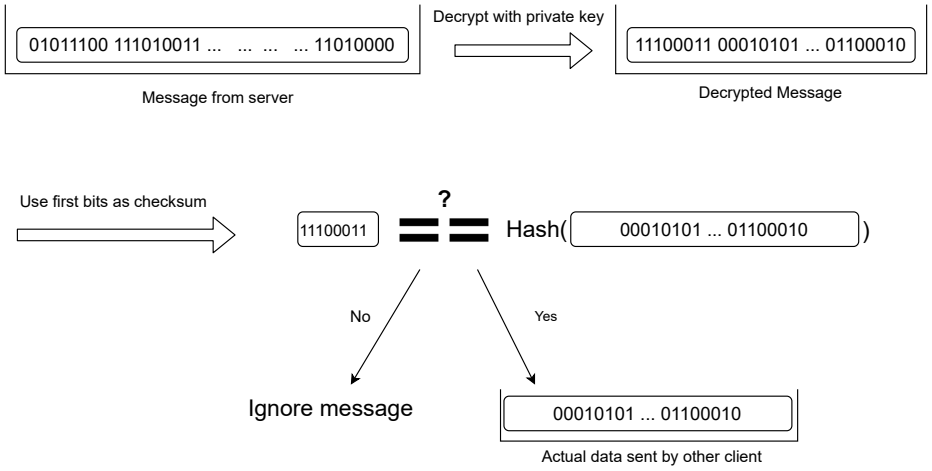
2.1.2 Receiving messages

Servers send all recovered messages to all clients. Once more than f servers send the same recovered messages for the round, the client "commits" / accepts them.

For every message received, the client uses its private key to decrypt the message.

It then uses the first bits (of the decrypted/attempted decryption) as *Checksum*#1 and checks for a match with the remaining bits.

If there is no match, the message is ignored. Otherwise, the message is considered to be a valid message that was sent to the client.



2.1.3 Handling client share

Servers receive all encrypted shares (for all d channels) from a client that wants to send a message. Using Raft protocol, these encrypted shares are replicated across all servers. It should be noted this means once a single non-faulty server receives the shares, we can consider **all** non-faulty servers to have their shares. Thus, if we wanted to, we could recover the secret at this point - we have enough servers that can decrypt their shares, broadcast them to each other (again via Raft) and use Lagrange Interpolation to recover the secrets.

To acheive anonymity, we can't just recover each message right away. Instead, we do secret-addition (as seen in class and/or previous solutions) with all client shares.

Rounds (sharing/recovering) are given a pre-defined length, and all clients will attempt to send a message every round (if they have nothing to say, they send an empty message).

The resulting secrets/messages are broadcasted to all clients.

2.1.4 Collision detection

Checksum#2 is used to detect collisions (2 messages being sent over the same channel).

When only a single message is sent on a given channel, after peforming addition between the secrets and recovering the final "secret" for that channel, the uncovered secret will be the single message sent (well, $[Checksum\#2, EncryptedMessage]$) since

all other secrets in the addition will be empty (all zero bits). Thus, *Checksum*#2 $==$ $Hash(EncryptedMessage)$ in this case.

On the other hand, suppose more than a single message was sent on a given channel and *Checksum*#2 is 128 bits long. The probability that *Checksum*#2 will match the hash for $EncryptedMessage$ (the rest of the bits) is negligible, $\sim\frac{1}{2^{128}}$. Although a cryptographic hash function is "overkill" (since it is a checksum for already encrypted and known-to-all data), using one would aid in avoiding false-negatives (getting closer to $\frac{1}{2^{128}}$/ randomness in the eyes of a PPT adversary).

2.2 Implementation details

This section is aimed towards explaining the main differences between the implementation to the the high-level protocol that was just shown.

2.2.1 Secret sharing and anonymity

To preserve anonymity between client and the servers, all clients send a message every round. If they have nothing to send, the message will be empty. When the servers later recover the secret for each channel they have no indication as to which client sent which message.

Secret sharing requires[1] a finite field. This creates an upper bound on the amount of information a secret may encode - since we have a finite number of elements. If one has a secret that is "too big", the secret can simply be split into multiple secret shares. On the recovery side, the recovered secrets are concatenated into the original secret.

This creates an issue for doing addition between shares during the recovery phase: if one secret fit in a single secret-share but another secret had to be split into 2 secret-shares - we cannot perform addition, a binary operation.

To avoid this issue (as well as to prevent leaking information about message size), an arbitrary size is used for all secrets. All secrets are padded to reach this arbitrary size, which is also the maximum secret size. If a client attempts to send a longer message it will be trimmed. In our case, that arbitrary length is 31 bytes.

2.2.2 Encryption

Although we assume we have a PKI "out-of-the-box", asymmetric encryption is slower than symmetric encryption which makes a hybrid[2] approach more appealing.

Hybrid could mean different things:

1. For every message: create a symmetric key, use it to encrypt the message, and passing the symmetric key encrypted with the asymmetric encryption scheme together with the encrypted message. The receiver decrypts the symmetric key, and then the message.

2. Only use the asymmetric encryption as a bootstrap for symmetric encryption - meaning it is used once initially to exchange a symmetric key that is used thereafter.

The second version is more performant, and it is what we use for client-server encryption.

However, if we think about how clients need to decrypt messages sent to them - the second version would require all clients pre-exchanging symmetric keys (which is not

[1] https://crypto.stackexchange.com/questions/9295/why-does-shamirs-secret-sharing-scheme-need-a-finite-field

[2] https://en.wikipedia.org/wiki/Hybrid_cryptosystem

given “out-of-the-box” as part of the assumed PKI) as well as trying to decrypt every message with all possible symmetric keys (those of all the other clients), which throws the performance gains down the drain.

For this reason, we use what is essentially a modification of the first version for client-client encryption - an NaCl crypto-box.

3 Theoretical analysis

3.1 Proof of correctness

Proofs for correctness will all be based on assuming Shamir Secret Sharing and the PKI are secure.

We did not define the exact properties these protocols should have in class so we won't do formal proofs, but it should be intuitively clear how our "proof in words" could be converted into formal adversarial reductions given concrete definitions for the properties / "tests" (like CPA) that these algorithms/protocols should have. (At least, that is the type of proof's we've seen in the Introduction to Cryptography course).

3.1.1 Privacy

Since the message is encrypted with the recipient's public key and is never transmitted in any other way, and of course private keys are kept secret - any party other than the recipient gaining any information about the message necessarily means there is a flaw in the PKI / public-key encryption algorithm, because at least some information is gained about the message.

In other words, the only attack vector to decipher or gain any knowledge about the message is through the PKI.

A reduction would emulate this protocol-breaking-adversary to create an adversary for the PKI.

3.1.2 Anonymity

Recipient

Assume by contradiction that the adversary knows learns something about the identity of the receiver of a message M that was not sent from an adversary-controlled client (and the receiver is not adversary-controlled either). Note the only way to confirm the identity of the recipient is by decrypting the message with its private key and verifying data-integrity via *Checksum*#1. The probability for *Checksum*#1 "accidentally" being a match for the message in negligible and in any case gives the adversary no information regarding receiver's identity (if it does, then the PKI is insecure..). As such, it must be the case that the adversary was able to guess the receiver's private key. Since our protocol never trasmits/advertises private keys, the flaw must reside in the PKI.

Sender

For uncovering the sender's identity there are 2 possible attack vectors:

1. Discovery via the metadata the sender encrypts. Since the metadata is encrypted with the recipient's public key (along with the message itself), uncovering via this method means there is a flaw in the PKI, like above for privacy.

2. Discovery via the secret-sharing. Since we do secret-sharing addition across many channels and force all clients to secet-share a message (possibly empty), the message sample for a given round contains at least $m - g$ secret-shares that

the adversary does not control. To gain any information on the sender's identity, it must be the case that the adversary was able to recover the message/secret prior to the recovery phase (thus knowing that client A sent message M). Meaning, without a non-controlled server's post-secret-addition shares (since $f < THRESHOLD$). This is a contradiction to the security properties of SSS. (Alternatively the adversary recovered with a non-faulty server's shares - but that server's shares are necessarily post-secret-addition, so that means the adversary's recovered secrets would be incorrect.)
By reduction; Assume SSS is secure, have a client share a secret with threshold $f+1$ and add another $m-1$ "empty" secret shares with threshold $f+1$. Give f of these shares to the protocol-breaking-adversary, and $n-f$ shares **post-secret-addition**. Since the protocol-breaking-adversary has a non-negligible chance of success (and necessarily recovering the secret along the way), we have devised a SSS-breaking-adversary with non-negligible probability success. Contradiction.

3.1.3 Liveness

Clients send a single (large) message containing all shares for all servers. Since we use Raft to ensure replication amongst all servers, we obtain liveness for the share phase via Raft's liveness.

Since the ability to recover is guaranteed at this point (because all non-faulty servers have all of their shares), liveness of recovery is guaranteed because the non-faulty servers can successfully send each-other their shares.

After recovering the secrets, servers send the results to the clients (and starts a new round).

Thus, a non-faulty sender and recipient are guaranteed liveness for their communication - since we are guaranteed that all of the non-faulty servers will recover the secrets and send the messages to all clients. A non-faulty client, once receiving more than f of the same results for a round - will "commit" them.

3.1.4 Safety

Safety is acheived via the safety properties of SSS and Raft.

SSS guarantees that any combination of enough shares will uncover the same secret. Since the shares are replicated via Raft, all servers use the same batches of shares for secret addition, which in turn means that due to SSS's guarantee for recovery of same secret regardless of which shares are used (in interpolation) - all servers that got all shares (== all non-faulty servers) will recover the same messages. Since the faulty servers are at most omissioned but not byzantine/malicious, we don't need verifiable secret sharing.

The recovered messages are are sent to all clients, and clients only "commit" to results once more than f servers send the same results.

3.2 Analysis

3.2.1 Message complexity

In class we talked about run-time analysis in terms of number of rounds. Surely, this can't be the only measure for protocol complexity unless message length is fixed, but even if we consider that the payload(strings) is fixed, there's a non constant overhead to the message size, which is the number of channels d and the number of servers n. Therefore, we will talk about the size complexity of messages instead.

First, we'll analyze the stable case where the Raft leader is not faulty, and there are no collisions:

- During the share phase, a client sends a *ClientShare*(see logic.rs) message of size $O(n \cdot d)$ to the leader, which then replicates it to all other $n-1$ nodes via Raft, therefore the size complexity of a single client's share is $O\left(n^2 \cdot d\right)$, and for the entire phase, it is $O\left(m \cdot n^2 \cdot d\right)$
- During the recovery phase, a server submits a *ServerReconstructShare* message of size $O(d)$ to to the leader which then replicates it to all other $n-1$ nodes via Raft, therefore the size complexity of this phase is $O(n \cdot d)$
- Once the recovery phase is done, a *ReconstructResult* is broadcast by each server to all clients. This message's size is $O(d)$ as it consists of up to d decrypted values, so the size complexity of this phase is $O(n \cdot m \cdot d)$
- There's also a *ServerBeginReconstruct* message of size $O(1)$ which is submitted via Raft and used to indicate the beginning of a new round either after timeout or after the recovery phase is done. Often all n servers will try submitting this value if there's a timeout, resulting in all those n messages being replicated, thus the size complexity for beginning a new round is is $O(n \cdot m \cdot d)$

To conclude, the size complexity of all messages sent for a single round is $O\left(m \cdot n^2 \cdot d\right)$ in case there are no faults nor collisions.

Generally, in case of message drops, replicating a message can take more iterations where the message is re-sent to the new leader(and that leader might also happen to be faulty, causing the process to repeat). The process of electing a leader is more or less random[3], and since half of the servers are non-faulty, the expected number of elections until getting a valid server is 2(expectation of Bernouli variable). Since the messages involved in a Raft election(vote request, vote response) are $O(1)$, and there at most $O\left(n^2\right)$ messages (e.g, if every node begins an election at the same time, he asks other n nodes for their vote, so n^2 vote requests and responses are transmitted), the expected size complexity of a single round is still $O\left(m \cdot n^2 \cdot d + 2 \cdot n^2\right) = O\left(m \cdot n^2 \cdot d\right)$

Now, let's put collisions into the equation: In our implementation, whenever there's a collision, the client tries re-sending his value at the next round. So if we denote c the

[3]The randomness of Raft leader election mostly comes from the fact that the election timeout is uniformly randomly generated, but there are other factors such as network congestion and the log length, see Raft paper for more details. The most important takeaway is that a node which (appears to be) non faulty is preferred, since his log length would be more up-to-date, and if there are multiple non faulty nodes, all of them have the same probability of being chosen.

expected number of collisions when sending some client value over multiple rounds, then the exxpected message size complexity of sending a single value is $O\left(m \cdot n^2 \cdot d \cdot c\right)$.

3.2.2 Optimal channel count

Assuming the number of servers is constant (which seems quite reasonable, considering the fact that horizontal scaling only adds complexity, and fault-tolerance surely has diminishing returns after geographic/network and hardware distribution etc), there should probably be some optimal choice for d. Meaning, one that minimizes the expected message size over several rounds necessary for a successful client-share of some value. This is a tradeoff between the number of channels vs probability of collision. More channels means larger message sizes each server per attempt, but it also means a lower probability for a collision which lowers the expected number of attempts, and vice-versa.

Formally, we are interested in $\text{argmin}_d m \cdot n^2 \cdot d \cdot c$. Let us first express c: for any certain channel and round, calculating the probability of collision is a classical formulation of the birthday paradox. Denote p the probability of a colision, then

$$\begin{aligned} 1 - p = P(\text{NoCollision}) &= \prod_{i=0}^{m-1} \frac{d-i}{d} \\ &= \frac{\prod_{i=0}^{m-1} d-i}{d^m} \\ &= \frac{m!\binom{d}{m}}{d^m} \end{aligned}$$

Denote C the number of rounds until a client successfully sends some value. The transmission of a round is independent from previous rounds, and a success occurs at probability of $1-p$, so we have $C \sim \text{Geom}(1-p)$.

So by expectation of geometric variable we get:

$$c = \mathbb{E}[C] = \frac{d^m}{m!\binom{d}{m}}$$

To summarize, we are interested in

$$\begin{aligned} &\text{argmin}_d m \cdot n^2 \cdot d \cdot \frac{d^m}{m!\binom{d}{m}} \\ &= \text{argmin}_d \frac{d^{m+1}}{\binom{d}{m}} \end{aligned}$$

And also $d \geq m$ otherwise collision is guaranteed. Generally, integer optimization is NP hard and we don't see an analytic way of solving this - perhaps an approximation can be reached via the Gamma function or Stirling's approximation.

It should be noted that message size isn't the only criteria - if we are interested in responsiveness, increasing the channel count beyond the above optimum would help in ensuring messages are committed in fewer rounds.

3.2.3 Additional ways of handling channel collisions

There are a few other viable approaches that we haven't implemented but can also mitigate/solve the issue of collisions:

- Use an exponential backoff strategy(often seen in communication networks) - in this strategy, clients don't always send a (non-zero) message every round(even if they have something to say) - in case of a collision, a client waits a certain(random) amount of rounds before trying to re-send, and that amount is an exponential function of the number of consecutive collisions. This allows us to use $d < m$, which is essential for a real world scenario, since scaling the number of channels into the number of clients(which can be millions) would make messages prohibitively large.

- Use a dynamic channel count dependant on the application's usage and network congestion. A higher channel count allows for increases the network congestion(and also the CPU & RAM usage) but improves latency(messages have a higher probability of being successfully delivered in one round) and vice versa. We'd still need an upper bound on the channel count(see above), as well as a lower bound(the minimal anonymity set size - for exaple, if $d = f + 1$, there's a chance that all adversarial clients and one honest client will be chosen, this will allow inferring which message was sent by the honest client)

- Use a randomness beacon to determine which clients for a certain round are supposed to send a message. Namely, the beacon can be used to randomly select a d-sized subset of m clients. This eliminates collisions, but brings its own set of issues of having another service(thus having to deal with the reliability and security of the beacon), and like before, d has to be big enough to ensure a large enough anonymity set.

4 Additional thoughts

4.1 Comparison to a player elimination solution

Player elimination, the solution to the second theoretical exercise set (since there was no PKI) is a legitimate alternative solution.

A refresher on player elimination without secrecy

A player elimination solution would not send all (encrypted) shares to all servers and have the servers replicate the shares amongst themselves. Instead, it would send each servers its appropriate share only - unencrypted. If more than $THRESHOLD$ (the threshold for the client's secret share) servers got a share, we would continue on to the next phase where each server would secret-share the share that it got from the client with all other servers (and the same threshold as the original client's secret-sharing). The goal being that any $THRESHOLD$ number of servers would be able to uncover the client's secret - regardless of whether they got their share from the client (which was perhaps lost due to omission failure).

Since servers have omission failures as well, this would require ensuring that at least $THRESHOLD$ servers successfully secret-shared their client-share with all other servers. This can be done with an ACK, but still does not guarantee delivery.

So when things don't go smoothly, what do we do? The simple but key observation is that if either a share or an ACK wasn't received, one of the 2 parties is omission failed. Therefore we can "eliminate" both servers and have the client redo its secret-sharing, but those two servers will be "out of the game" for this particular client-share.

The client will re-do its secret sharing, but with $THRESHOLD$ lowered by 1, and it won't send shares to any servers that have been eliminated.

Essentially this means a single client-share can consist of several attempts until success(and probability-wise, will consist of many); in each attempt, the client secret shares and the servers try to secret share amongst themselves.

Player elimination with secrecy, a modification

The 2nd theoretical exercise set was for a single secret-sharing.

Here we have the requirement of secrecy, which is achieved by doing secret-addition between many client's secrets (and having many channels, to avoid babel). If we want to be able to do secret-addition, the number of shares must be fixed. We can't have a variable number of shares depending on how many rounds each client-share took, because then some shares needed to perform addition will be missing. After all, addition is a binary operation and "it takes two to tango".

An immediate remedy that comes to mind: the client always creates n shares with constant $THRESHOLD = f + 1$, but does additional secret sharing for every share of an eliminated server.

This isn't an efficient remedy because if the probability of success prior to the last few rounds was already very low, this modification makes things worse much worse. Now we also ruin our chances of success as we advance in player elimination rounds. It seems a client-share would almost always only be successful once all servers except one non-faulty server have been eliminated...

Message complexity

(Expected) message complexity analysis for this modified version proved rather difficult.

Our intuition is that in an unreliable network our solution's message complexity is better, while the modified player elimination solution would be the preferable choice for a highly-reliable network.

Of course, there are many further aspects that could influence that preference: local time and space complexities, ease of implementation, scalability etc.

4.2 Possibilities for further development, improvements and/or making a real-world app

1. Dynamic number of channels depending on number of clients connected. Further analysis required to find an optimal number that balances the probability for collisions with the required compute and amount of data passing on the wire.
2. Implement the PKI. Must ensure all servers and clients are in sync regarding public keys of all parties.
3. Identities and authentication/verifiability: our project simply uses incrementing IDs for clients.