Server Fault Tolerant DC-Net Protocol for Dissent

Brandon Li
Dr. David Wolinsky, Mentor
Prof. Bryan Ford, CPSC 490 Advisor

1 Introduction

Anonymous online communication has become ubiquitous and commonplace enough for laypersons to have dinner table conversations about it. Several online anonymity techniques, most popularly Tor, have even made recent news headlines. But with these anonymity techniques a number of de-anonymizing methods have been developed as well. For example, Tor is susceptible to traffic analysis attacks. Therefore, modern anonymity techniques are commonly forced to weigh a tradeoff between scalability and strength of anonymity. Dissent is a protocol that attempts to circumvent this tradeoff and achieve both scalability and strong anonymity [1].

The Dissent protocol published in the OSDI paper [1] combines DC-nets and verifiable shuffles in a client-server architecture to provide scalable strong-anonymity. In this client-server architecture, the clients represent users who desire to communicate anonymously, and the servers simply facilitate this exchange. The verifiable shuffle techniques are used to securely assign pseudonyms to each of the participating nodes while the DC-nets are responsible for providing anonymity during message exchanges between the clients.

DC-nets are protocols designed to solve the dining cryptographers problem [2]. In the original statement of the problem, three cryptographers at a dinner table want to know whether or not somebody has paid for the bill without revealing the identity of the person paying. The solution to this problem requires all of the cryptographers to submit some notion of encrypted information that when pooled together and jointly manipulated, some information about the group is revealed, which in this situation would be whether or not somebody has paid. To ensure scalability, Dissent is designed such that instead of constructing a DC-net among all of the participating clients, each message exchange is performed through a DC-net among the servers in the system, which are presumably selected to be fewer in number than the clients.

One limitation of DC-nets is that the protocol’s success is often contingent on total participation of all nodes in the network. In the original example of three cryptographers, if for whatever reason any of the three cryptographers were to not send its encrypted message, whether intentionally, unintentionally, or as the result of a malicious attack, the group would not be able to compute the correct answer to their question. And depending on the specific protocol implementation, the entire system may either stall until the unresponsive node eventually submits its message or abort the session.

Dissent’s DC-net protocol is no exception to this problem [1]. In the implementation, the current policy for handling server disconnects is to abort the session, after which we can create a new session using the updated set of online servers. However, this type of operation is far too costly for practical use. Therefore, the main contribution of this paper
is to define a new type of DC-net round that can resist some server churn by operating on an active ‘view’ which involves a subset of the total servers and can change when certain servers go down.

2 Protocol Design

Supporting liveness in the case of server failures requires some changes to how message exchanges are conducted.

2.1 Original DC-net Protocol

Prior to any message exchange, first all server and client identities are established, all pairs of servers and clients generate shared secrets, each client is assigned a message slot (the collection of all decrypted messages will be broadcast to each node at the end of an exchange, but no node should be able to associate a particular message with a particular node), and each client determines the server that they would like to connect to, which is the server that they will be sending their inputs to and receiving the outputs from.

Once all of the initialization is done, servers cycle through the phases of: 1) submission, 2) inventory, 3) commitment, 4) combining, 5) certification, 6) output. Clients only have three phases: 1) scheduling, 2) submission, and 3) output. You can read about the full description in the original OSDI paper [1]. To summarize what is happening in this protocol, during a single round clients are first assigned a message slot. Client i then inserts the message they would like to send during this exchange into their slot, and then they xor that string (or byte array) with all $s_{i,j}$ where $s_{i,j}$ is a pseudorandom secret string client i shares with each server j. Note that this is every server, not just client i’s upstream servers. Servers collect these client cipher texts from each of their downstream servers during their own submission phase. Then, servers broadcast the list of clients they each successfully received cipher texts from to know which clients participated in the round and thus which shared secrets need to be cancelled out in order to decrypt the message. Then, each server xors the cipher texts that they directly received with each other, and then they xor this result with each shared secret $s_{i,j}$ for every client i that actually submitted a message during this exchange. The purpose of this is to cancel out the encryption by xoring out all of the shared secrets. After some verification and signing, the decrypted string (which now contains all client messages as opposed to a single one) is outputted to the clients.

2.2 Project Scope

For the purposes of this project, I only study the failure condition of global server disconnects, which means that when a server disconnects, it disconnects from all other nodes in the system. This is used to simulate failure of the server itself, as opposed to a network failure between two particular nodes. I do not consider scenarios involving malicious activity and assume ‘honest’ nodes that adhere to protocol. I also assume that once a server has disconnected, it can no longer reconnect to the current session. I do not address key generation or scheduling and will assume that each keys and slots will be distributed as they previously were. I assume each client and server have known public identities.
2.3 Modified Protocol

The client protocol remains the same except that instead of computing a ciphertext using shared secrets that client i has with each server, client i simply does so with each server in the current view, which again is a subset of the servers that in some sense constitute the DC-net participants for the message exchange (calculation and further description in Section 2.4). Additionally, a client that encrypts a message using an incorrect view will be notified of the current view by its upstream server.

As for the servers, not all currently online servers will be in the view, although they still participate to some degree in the message exchange so that their downstream clients are still able to send messages. Here is the modified server protocol:

Some key changes:

- Servers not in the view simply xor all ciphertexts without xor-ing in their shared secrets.
- Whenever a server perceives that another server has disconnected, it will exit the current message exchange and go into a voting phase in which it recomputes the view and sends a view change proposal to all of the other servers.
- Once a server has collected enough votes for a view change, it will broadcast this view change to its downstream clients and begin a new message exchange.

![Figure 1: The control flow graph for the new, modified protocol](image)

The modified protocol control flow is illustrated by Figure 1. Essentially, phases 1-6 proceed as normal (except the special handling of live but not-in-view servers mentioned above), but when an in-view server disconnects, then the servers immediately go into view change...
voting. If the servers can successfully agree on a new view number, then they start a new message exchange from phase 1: submission.

2.4 Views

The notion of a ‘view’ is first discussed in a Byzantine Fault Tolerance paper [3], but there the author describes a more complex system for state machine replication, whereas here we will use a much simpler idea of a view. For this paper, a view represents a subset of the servers that are known to be online. There may be other online servers outside of the view, but the servers in the view are the only ones whose shared secrets are used for cipher text generation and decryption. Each client and server in the system can independently and directly calculate the view based on a view number using a simple algorithm:

Algorithm 1 Calculating View Membership

| Seed PRNG using view number |
| memberCount ← 0 |
| viewSize ← $\alpha \times \text{numServers}$ |
| view ← $\emptyset$ |
| while memberCount < viewSize do |
| next ← randomInt() $\%$ viewSize |
| if next $\notin$ view then |
| view ← view $\cup \{next\}$ |
| memberCount ← memberCount + 1 |
| end if |
| end while |
| return view |

When a server perceives that another server has gone down that is in the current view, it calculates the next good view and proposes to move to it:

Algorithm 2 Selecting Next Good View

| $i \leftarrow$ currentViewNum + 1 |
| while view$_i$ contains a failed server do |
| $i \leftarrow i + 1$ |
| end while |
| return view$_i$ |

Figure 2 gives an illustration of a hypothetical view change scenario.

3 Implementation

Dissent is built on the Qt framework.

3.1 Null Round

Instead of implementing the view membership protocol on top of the full Dissent DC-net protocol, I simplified the original DC-net protocol to construct a “null round” on which I tested the view logic.
Figure 2: A sample view change scenario where an in-view server failure causes a view change from (a) starting state $view_i$ to (b) end state $view_j$.

Some simplifications:

- Instead of using Diffie-Hellman keys as secure shared secrets, I simply generated publicly available unique identifiers for each client-server pair for the purpose of the “encryption” step. This obviously does not provide truly secure encryption but is sufficient to help simulate the xor-ing involved in the DC-net protocol.

- I did not generate slot numbers based on pseudonyms. Instead, each client’s slot number was simply the index in the client list. Again, this would not truly provide encryption but is sufficient for the purposes of the algorithm.

- Previously, each round only performed a single message exchange, but instead, I allow the same round to perform multiple message exchanges. This reduces some initialization overhead and also helps contain the handling of server disconnects in the round code as opposed to spilling into the sessions establishment code.

- I omit sending and verifying the hash value for each server’s cipher text, and I also subsequently omit attaching signatures to the final output. The purpose of this is simplicity, as these steps are unnecessary for the evaluation of the view change protocol and mainly exist for security purposes.

- The original protocol contained a blame mechanism for providing an “accountability” feature in case something were to go wrong. The current implementation of the fault tolerant protocol does not yet include this feature.

3.2 Code Organization

SFTNullRound contains some basic round initialization, handles server disconnections, is the entry and exit point for input and output data.

SFTMessageManager contains all of the logic for processing messages and manages the control flow for the nodes (determines when we enter a new round phase).

SFTViewManager a utility class that helps determine view membership.
3.3 Round Phases

Clients do not have much in the way of phases. They simply send messages and receive the group of client’s decrypted messages when the servers are done processing the messages. Clients may also receive view change notifications from servers, in which case they use the new view number for all subsequent messages.

Servers, on the other hand, have several different phases that they cycle through as per Section 2.1.

- CollectionPhase
- ClientAttendancePhase
- ExchangeCiphersPhase
- ViewChangeVotingPhase

The CollectionPhase, ClientAttendancePhase, and ViewChangeVotePhase’s correspond to the Submission, Inventory, and Combining phases described in the original algorithm. Again, as described in Section 3.1 some sections of the protocol (the second half of the commitment phase and the certification phase) were omitted for the sake of simplicity.

Servers may abort the current message exchange and enter the ViewChangeVotingPhase any time they sense a server disconnection. After they agree on a new view number, they can start a new message exchange by re-entering the CollectionPhase.

Each server broadcasts information to the other servers when appropriate. However, other servers only accept this information if they are in the correct phase of the message exchange to be doing so. We could run into problems if a server rejects a message due to not being ready for it and then later ends up stalling because its peer servers do not know to resend the information. Therefore, in addition to broadcasting its own information, each server can also request information from all other servers, which other servers reply to with appropriate information.

3.4 Views

In principle, the view size as a proportion $\alpha$ of the total number of servers could be specified in a configuration file. I set the view size as at least $2/3$ of the number of total servers. When the number of total servers drops below this number, the session aborts.

A view change is established when a server has received a view from at least $\beta$ of the servers on a particular view number, where $\beta$ is a proportion also specified in the configuration file. I set this to be $2/3$ as well.

3.5 Testing

First, rounds with between 5-10 servers and 15-20 clients were tested using Google’s testing framework. Message exchange durations as well as view change durations were recorded. All time metrics were measured using Qt’s QDateTime class. Stress tests were done with up to 15 clients and 80 servers. Tests with multiple disconnections were done with a starting state of 10 clients and 20 servers, and message exchanges were performed one at a time after successively disconnecting a single server, until there were now enough servers to be able to execute a message exchange.
4 Results and Discussion

The server failure tolerance was successfully demonstrated using the specified number of clients and servers. Once too many servers disconnected, the current round aborted, as intended.

4.1 Design Decisions

View Generation There were many tradeoffs that were thought through in designing this protocol. The first major consideration was how to generate the views. One option was to enumerate all possible permutations of viewSize-many servers and to standardize an indexing scheme for these enumerations by having each node locally compute these enumerations. Another option was to have the servers notify the clients of the view membership rather than sending the view number and having the clients locally compute it. The current implementation (using views generated on the fly given a view number) was favored over these other two options because of its simplicity and space-efficiency. Instead of pre-computing and storing all possible permutations or sending the entire list of servers in a view every time we have a view change, we can simply store a single integer signifying the current view number at each node.

View Change Vote Broadcast Another major consideration was the handling of the view change proposals. The current implementation essentially has each server broadcast a vote to all of the other servers. The problem with this approach is that if many servers vote on different new view numbers, then they can never reach a consensus on what view number to switch to. Fortunately, this becomes less likely when we assume that servers either disconnect from all servers or none. If a server can disconnect from only some of the other servers, then different servers will have a different understanding of the current state and may propose different new view numbers. The other option was to create some notion of a proposer/leader that proposes one specific value (in this case a view number) for all of the other servers to vote on. The benefit of this approach is increased synchronizan. There is a lot of literature discussing implementations of proposer-based systems like Paxos, but because they are difficult to implement, this was avoided during the duration of this project, so a proposer-based-model remains an area for potential future work.

Online, Not-in-view Server Behavior Finally, there was a decision to be made for what to do with online servers that were not in the active view. Since clients only have a connection to one server for the duration of the round, it is currently not possible for them to simply switch to another server, although switching servers in the case of server disconnects could be an interesting future project. And not allowing clients whose servers were not in the view to participate in the message exchange is very undesirable in the general case and could also lead to some pretty bad denial-of-service attacks on clients. Therefore, the remaining two options were to either have each server simply forward each message to some designated server or to participate in the exchange but not use any of its shared secrets to generate the cipher texts, and simply xor together the client messages that they received. The challenge with the first approach is that there needs to be some way for the servers in the view to know when to terminate the collection period, and this is not easy to do if the servers do not know who to expect forwarded messages from. This would then require some designated destination for the online-but-not-in-view servers to forward their messages to.
This may also result in increased network traffic. For these reasons the second approach was implemented.

4.2 Performance

In the range of testing conditions, the message exchange times were fairly consistently in the tens of milliseconds regardless of the number of clients and servers and whether or not nodes had been disconnected. On the other hand, Figure 3 the time for view changes seemed to increase with an increase in server pool size. However, this may just have been a result of the fact that disconnecting a server takes a nontrivial amount of time and was done sequentially in the test (i.e. server i sees server j disconnect before server k sees server j disconnect). It is also possible that after more servers disconnect, it takes a longer time to find an appropriate view that does not contain any disconnected servers. Though, realistically, with such a small number of clients and servers, that sort of effect may not be noticeable. A larger number of nodes may be tested in the future.

![Figure 3: The amount of time it took to perform a view change with various numbers of servers. The number of clients was fixed to 20.](image)

The view change protocol was able to successfully execute under the “stress” conditions, which consisted of 15 servers and 80 clients, and produce the correct output in the send test following the disconnection.

<table>
<thead>
<tr>
<th></th>
<th>First</th>
<th>Second</th>
<th>Third</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>2420 ms</td>
<td>2263 ms</td>
<td>2506 ms</td>
</tr>
</tbody>
</table>

Table 1: Time to perform the nth view change on a network of 10 servers and 20 clients.

Based on the small amount of successive disconnection testing that I did, Table 1 shows that when starting with 10 servers and 20 clients, there was not much of a difference in view change time during each successive view change.

5 Future Work

The most immediate next step would be to integrate this server-failure-tolerant system with a truly-anonymous DC-net round. But beyond that, there are still many other interesting
questions to consider, especially because I established many assumptions at the beginning of this paper that need not hold.

5.1 Servers Rejoining

One such question is how to extend this system to allow servers to rejoin after they have been disconnected. This would require some re-architecting given the current implementation, since once a server has disconnected, the session layer does not allow it to rejoin. Assuming that this limitation were removed, we would need a more sophisticated notion of a view that encapsulates more state than simply a view number in order to catch servers up. There would also need to be some work to reconnect the server to its downstream clients.

5.2 Multiple Servers per Client

In the current implementation, if a server disconnects from the rest of the network, then its downstream clients essentially disconnect from the session as well, as they each start with a connection to only a single server. Thus, each client is in some sense linked to the server it selects at session establishment. This seems like undesirable behavior, as a single server disconnect could potentially affect hundreds of client nodes in a real system.

There are two possible obvious solutions to this. Either each client starts off connected to multiple servers and can choose at will a server to send its message to for a particular exchange, or each client has the ability to switch servers during a round, either from the entire registered server list or from some predefined subset of potential upstream servers. In the latter solution, it still may be the case that at a given time a client is locked onto a single upstream server and may switch that server when its currently upstream server disconnects, for example.

Both of these options pose some design challenges but should be implementable given the proper restructuring at the session level.

5.3 Handling Malicious Activity

The original intent of this paper was to explore how to handle server churn, since the original Dissent paper already discussed how to handle client churn. One of the assumptions that I made was that nodes would try to adhere to protocol and would not act maliciously. However, a stronger fault tolerance claim would show that this view change procedure can handle some degree of malicious behavior.

6 Conclusion

This project demonstrates a successful step towards a fault-tolerant DC-net based anonymous communication protocol. We are able to use a view representing subset of the server list to ensure liveness across message exchanges despite server failures. Some simplifying assumptions were made in the implementation described here, but there is strong reason to believe that this view change protocol can be applied to the full Dissent DC-net round as well.
7 Acknowledgments

I would like to express many thanks to Prof. Bryan Ford for his invaluable guidance as my advisor on this project, Dr. David Wolinsky for so generously meeting up with me every week to work through many of the technical details, and the Yale CS department for making this project possible and for teaching and stretching me in so many different ways these past 4 years.

References

