**Project BACnet**

# Project BACnet (Basel Citizen Net) - Networking without Internet

The Internet has become so ubiquitous that we take it for granted, forgetting completely how much its use depends on political goodwill: Access restrictions are quite common worldwide, be it the "Chinese firewall", India cutting the Internet for Kashmir, the Spanish Police seizing the DNS name of the Catalan independence movement or Iran, Hong Kong and Burma/Myanmar Iran switching of the Internet during civil protests to cut of peoples communication.

**Github:** `https://github.com/cn-uofbasel/BACnet`

## Idea

BACnet (BAsel Citizen Net) followed concepts developed in 'Secure Scuttlebutt', a decentralized approach for implementing server-less distributed applications using replicated append-only logs. The motivation for BACnet was the hypothesis that sensor-resistant communication systems can be built from scratch with reasonable effort and time, covering fancy Wifi as well as more low-tech USB stick swapping, yet serving humans with basic inter-person messaging.

Instead of using the Internet, USB sticks are used to implement a so called "Sneaker Net" where content is propagatedamong friends. Wireless communication using the unlicensed WiFi frequency band was included for speeding up content dissemination over a distance of more than 10km. Finally, two groups also looked into forwarding sensor data over LoRa, along-range and low-energy communication technologies.

In spring 2020, BSc students of the lecture *Introduction to Internet and Security* at the University of Basel built a first BACnet system from scratch. Students of the seminar Re-Decentralization continued working on the project in autumn 2020. They build projects and application on top of BACnet, such as encrypted connections, file systems, secure-team chat and distributed games.

## Content

1. Decentralized File System – (Anna Diack, Jonas Wittmer, Leonhard Badenberg)
2. Union Directory Service – (Leonardo Salsi, Ken Rotaris, Tunahan Erbay)
3. Double Ratchet - Encrypted Messaging – (Moritz Würth, Oliver Weinmeier, Renato Farruggio)
4. Secure Team Chat – (Rahel Arnold, Maurizio Pasquinelli, Tim Bachmann)
5. Decentralized Games – (Carlos Tejera, Alexander Oxley)
6. Decentralized TCP – (Günes Aydin, Pascal Bürklin, Damian Knuchel)

**Jonas Wittmer** **Leonhard Badenberg** **Anna Diack**

# Projekt: DecentFS

**Abstract:** Das DecentFS ist ein file system welches auf BACnet append only log implementiert wurde.

**Github:** `https://github.com/cn-uofbasel/BACnet/tree/master/redez-sem-hs20/groups/01-decentFS`

## Projekt Idee

Das Dateisystem besteht aus zwei Dateien, von denen eine die Metadaten und die andere den Inhalt enthält. Die Metadateneinträge können auf mehrere Inhaltseinträge verweisen. Das Inhaltsprotokoll wird nur benötigt, um den Inhalt einer Datei abzurufen. Dies ermöglicht schnellere Operationen an Dateien, die keinen Zugriff auf den Inhalt und die Deduplizierung des gleichen Inhalts erfordern.

Aufgrund der Art, dass nur Protokolle angehängt werden, können veraltete Einträge nicht aktualisiert oder gelöscht werden. Jede Änderung wird am Ende des Protokolls angehängt. DecentFS muss das gesamte Metadatenprotokoll nacheinander scannen, um den aktuellen Status eines Pfads zu bewerten. Der letzte Eintrag eines Pfads im Protokoll definiert den aktuellen Status.

Für das Inhaltsprotokoll spielt die Reihenfolge der Einträge keine Rolle. Auf jeden Eintrag im Inhaltsprotokoll muss mindestens ein Eintrag im Metadatenprotokoll verweisen (1-n- oder 1-many-Beziehung).

Die Einträge in den Protokolldateien werden als cbor2-serialisierte Listenobjekte gespeichert. Der erste Eintrag beider Protokolldateien kann Metadaten zum Dateisystem selbst enthalten, um Kompatibilität und Migrationen zu gewährleisten.

## Komponenten

Die Protokolle werden nacheinander vom ersten bis zum letzten Eintrag gescannt. Dies kann eine Leistungsbeeinträchtigung für große Protokolle sein. Es gibt jedoch einige Minderungsansätze.

- Metadatenprotokoll

  Ein Metadateneintrag besteht aus 5 Feldern.

- Inhaltsprotokoll

  Ein Inhaltseintrag ist ein einfaches Tupel. Um Duplikate oder Redundanzen zu vermeiden, sollte jeder Eintrag eindeutig sein.

## Bereitstellung/Integration

Um DecentFS verwenden zu können, müssen Sie diesen gesamten Ordner in Ihr Projekt importieren. Fügen Sie diesen Ordner einfach Ihrem source Tree hinzu und folgen Sie der obigen Installationsanleitung. Die gesamte API befindet sich in der Datei crypto.py und api.py. crypto.py, feed.py cli.py etc. wären nochmal eigene Komponenten.

## Selbstreflexion

Um dieses Projekt umzusetzen haben wir gelernt wie wichtig Kommunikation ist. Um sich besser organisieren zu können war es wichtig regelmässig nach dem aktuellen Stand zu fragen und eine klare Verteilung festzulegen. Bei uns hätte die Organisation besser laufen können und hätten wir früher begonnen wären sicherlich mehr Fragen geklärt worden. Im Allgemeinen war uns die Struktur des Projektes zu Beginn noch nicht ganz klar und wir hatten deshalb Startschwierigkeiten, wobei wir uns allerdings Hilfe per Mail gesucht haben. Zum Schluss haben wir noch die cat, cd, pwd und ls commands zur loop hinzugefügt. Wir haben mit absoluten Pfaden gearbeitet da dort der Speicherort einer Datei/Verzeichnis definiert ist und somit ein vollständiger Pfad ist.

**Leonardo Salsi** **Ken Rotaris** **Tunahan Erbay**

# Project: 02-unionDir

**Abstract:**
UnionDir is an application that applies a centralized approach to file systems. Our implementation is accessible in the form of an intuitive command line program, which incorporates a lot of the usual commands present in regular terminal programs like cd, ls or cat. Each user has complete control over their data. The program's main objectives are to show multiple files with the same name in the same folder (name-collisions) and to mount an external file system into an existing one. This is done by using actively managed namespaces. In this context, namespaces are defined as a set of resources, which are partitioned in a way that there are multiple namespaces to which resources refer to. To be able to synchronize files and permissions across multiple users, a server of choice is accessed, which actively processes and manages all the incoming commands from the user terminal. UnionDir allows going beyond the limits of superficial file or folder names and allows users to cooperate directly by mounting file systems.

**Github:** https://github.com/cn-uofbasel/BACnet/tree/master/redez-sem-hs20/groups/02-unionDir

## Project Idea

UnionDir is intended to extend BACnet. It is able to merge different file systems via a command-line interface. It allows the user to mount an external filesystem into his own creating a uniform filesystem. Files with the same name can be distinguished by the fingerprint and the last change date. This means that there is not just one truth for a file, but several from which the user can choose.

## Components

The client is the terminal interface (terminal.py) with which manipulations of the file system are done and which were specifically designed to be as user friendly as possible (including problem specific help). The server is responsible for handling all permissions and synchronisation requests by all the connected clients. The server also manages the namespaces in order to know where a mount is made / can be made. The mounting functionality is mainly defined in "protocol.py" and is responsible for merging two given file systems. The file "inputhandler.py" catches all commands typed into the terminal and calls the correct operator functions.

## Deployment / Integration

Please have a look at our README.md file in our git repository. We provided a step-by-step guide on how to run the project together with a detailed list of all commands.

**Moritz Würth** **Oliver Weinmeier** **Renato Farruggio**

# Project: 03-doubleRatchet

**Abstract:** We extended BACnet with a forward- and backward-secure chat, by implementing the Extended Triple Diffie-Hellman key exchange protocol X3DH and a double ratchet algorithm for generating subsequent message keys. Our implementation runs on Linux and MacOS. In a first step we implemented X3DH and double ratchet and testet its functionality over TCP. In a second step, we replaced TCP in the transport layer with cbor files so that it is BACnet conform. The X3DH algorithm is designed to have a server to support the key exchange. Since BACnet is decentralized, we had to slightly adapt our protocol so that it works for a serverless setup, such as BACnet. The double ratchet protocol is designed for secure asynchronous messaging, and we were able to implement it as defined on the Signal website [1]. Finally we wrote an implementation that works over rsync.

**Github:** https://github.com/cn-uofbasel/BACnet/tree/master/redez-sem-hs20/groups/03-doubleRatchet

## Project Idea

Secure Chat is an application intended to extend BACnet. It allows two parties to have Signal-like secure chats that follow the the double ratchet algorithm, thus guaranteeing forward-secure and backward-secure chats. This also means that all messages are being saved encrypted and are being decrypted only once.

## Components

There are 4 classes that are not being used directly by the user (alice.py, bob.py, helpers.py, signing.py).
The class TCP_chat.py is being used for secure chat over TCP. This is our prototype for testing doubleratchet.
The class BACnet_local_chat.py is being used for chatting over a log file.

Both of these classes offer secure chat.

## Deployment / Integration

### 1. TCP

Navigate to the nonRedezDoubleRatchetDemo folder.
1. Bob starts TCP chat:
python3 TCP_chat.py localhost 1234
Then an ip address and port will be shown, for example:
server started with: ip 192.168.1.103 port 62953

[1] https://signal.org/docs/specifications/doubleratchet/

2. Alice starts TCP chat:
python3 TCP_chat.py 192.168.1.103 62953
3. Alice writes and sends an initial message

Both parties are now free to go and can communicate freely and securely over TCP. To ensure that there is no man in the middle, they can compare their shared keys offline. The shared key is shown the first time they establish contact.

### 2. BACnet

Navigate to the nonRedezDoubleRatchetDemo folder.
1. Bob starts chat:
python3 BACnet_local_chat.py Bob
Then a file "cborDatabase.sqlite" will be generated. Share it with Alice (via USB for example).
2. Alice starts chat:
python3 BACnet_local_chat.py Alice
3. Alice writes at least one message.
4. Alice writes "quit" to close the program.
5. Transfer the file cborDatabase.sqlite back to Bobs directory.
6. Bob starts chat to write his answer:
python3 BACnet_local_chat.py Bob
7. Bob writes "quit" to close the program.

### 3. chat_over_rsync

Navigate to the nonRedezDoubleRatchetDemo folder.
1. Bob starts chat:
python3 chat_over_rsync.py Bob
The first time this is run, 2 files will be generated. In "other_directory.txt" the user should write the absolute path to the log file from the other party.
2. Alice starts chat:
python3 chat_over_rsync.py Alice
3. Bob starts the chat again:
python3 chat_over_rsync.py Bob
4. Both parties can now send and retrieve messages. To retrieve new messages, write "rsync" in the chat.

Both parties are now free to go and can communicate freely and securely. To ensure that there is no man in the middle, they can compare their shared keys offline. The shared key is shown the first time they establish contact.

To delete all conversations run reset.py.

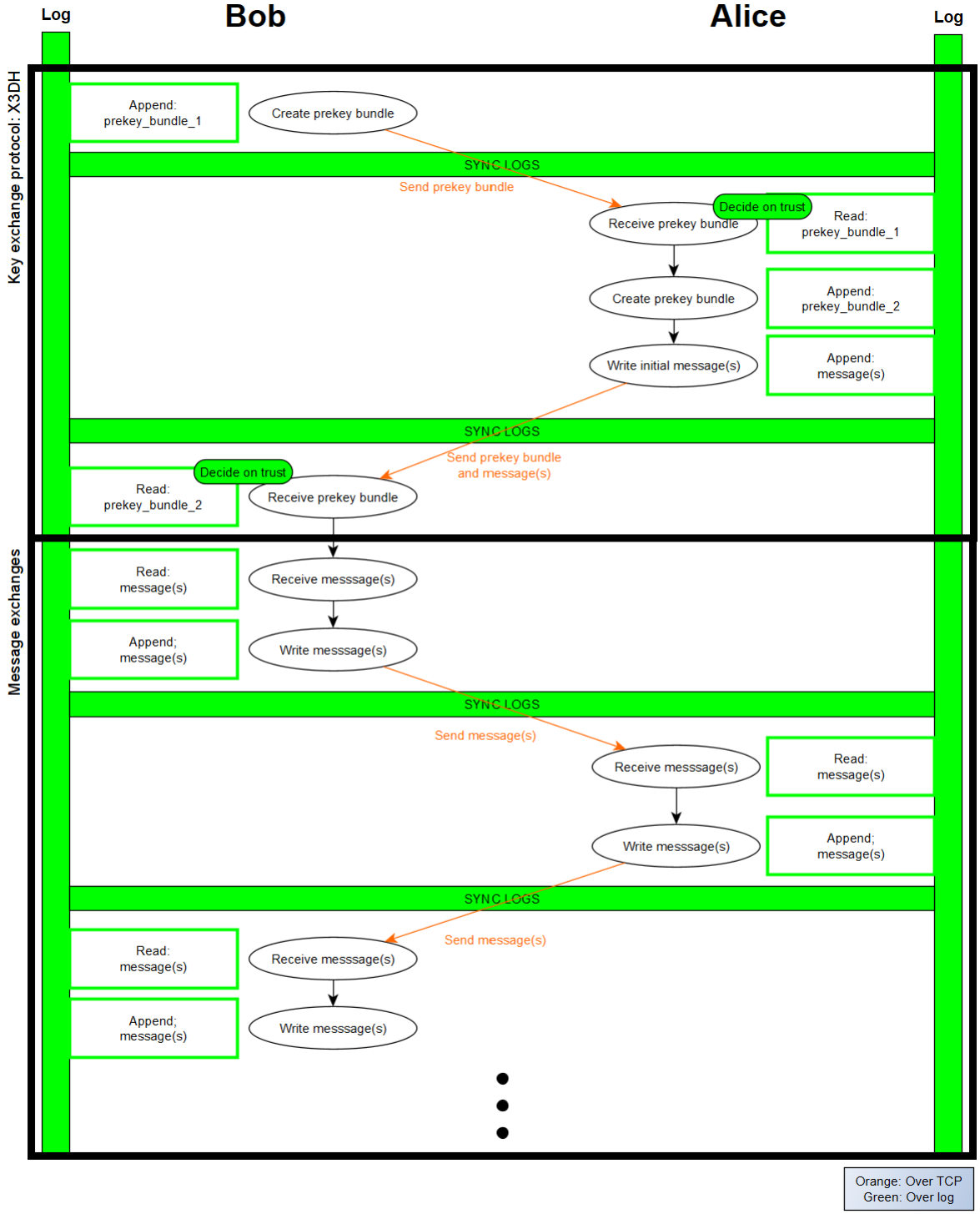
## Conclusion

We managed to implement a chat that is encrypted and offers forward- and backward security by design. The BACnet structure partly comes in handy, since we don't have to worry about out-of-order messages because the correct order of messages is guaranteed in a log.

Solvable but unsolved problems:
We currently save our private key locally, encrypted by a password. The current password is hardcoded as "pw". To make it secure, even when the data on the phone is compromised, the password should be entered by the user, everytime he runs the program.
The key exchange protocol is prone to errors. When Bob writes a message before he received the first message from Alice, all messages will be shown as empty, and there is no way of recovering the conversation, other than resetting both sides, by deleting the folders "key_files" and "public_key", aswell as the "cborDatabase.sqlite" file, on both sides.
Trying to retrieve a message when there is no new message will break the algorithm, and then both sides have to reset the conversation.

Inherent problems:
BACnet has no server. The key exchange protocol X3DH requires Bob to publish his information (prekey_bundle) on a server, so that Alice can ask the server for Bob's information. Normally, Bob would publish several signed onetime prekeys, and the server would make sure that keys that are being used are being deleted, and so no other person can use it. In a log, as BACnet, information is not being deleted, and keys could potentially be used by several people, which would result in a problem (the key exchange would not work for them). Therefore, Bob only publishes one onetime prekey when he wants to establish contact. Alice then has to know that Bob wants to establish contact, so that she can complete the key exchange.

## Protocol


Log
Bob
Alice
Log
Key exchange protocol: X3DH
Message exchanges
Append: prekey_bundle_1
Create prekey bundle
SYNC LOGS
Send prekey bundle
Decide on trust
Receive prekey bundle
Read: prekey_bundle_1
Create prekey bundle
Append: prekey_bundle_2
Write initial message(s)
Append: message(s)
SYNC LOGS
Send prekey bundle and message(s)
Decide on trust
Read: prekey_bundle_2
Receive prekey bundle
Read: message(s)
Receive messsage(s)
Append; message(s)
Write messsage(s)
SYNC LOGS
Send message(s)
Receive messsage(s)
Read: message(s)
Write messsage(s)
Append; message(s)
SYNC LOGS
Send message(s)
Read: message(s)
Receive messsage(s)
Append; message(s)
Write messsage(s)
Orange: Over TCP
Green: Over log

**Rahel Arnold** **Maurizio Pasquinelli** **Tim Bachmann**

# Project: Secure Team Chat

**Abstract:** Secure Team Chat is an application extending BACnet. It allows creating new users as well as following and unfollowing others. Moreover, creating team chats, inviting users to them and exchanging messages within such a group are available features.

**Github:** redez-sem-hs20/groups/04-secureTeams

## Project Idea

In Secure Scuttlebutt private chats between two persons and small group chats with less than 8 participants are possible. In this project, the idea was to implement a first group chat, called channel, in BACnet. A chat has always one owner, which can invite other users to his chat group. Within a channel, the messages are encrypted.

## Components

The application uses user-specific logs to store interactions. The content of those events can either be cleartext (readable for everyone) or cyphertext (readable for a specific set of recipients only). We introduced channels, an abstraction for a set of members (and owners) that share a common key for the encryption of messages within this specific group. Lastly and for end-user convenience, we store an alias file locally that maps a self-chosen name to a corresponding public identifier. So there is no further requirement of always having to insert public keys into each command. Additionally, these IDs can therefore be represented as those individually set aliases in the logs too.

## Deployment / Integration

The events and the append-only log files are provided by the BACnet code. The synchronization of the log files is achieved by reading all user logs that the current user actually follows. All events that are not yet in the log file of the current user are wrapped in a clear text event with type `log/sync`. If the event is already a sync event, it is first unwrapped.
The program is called with the command `./user.py (alias) <command> [options]`, where the alias is the human readable name of a user. The implemented commands include `create`, `log`, `message`. The command `create` will create a new user including all data like private and public key. The command `log` will display the contents of the log file as a list of formatted messages. The `message` command will send a new message to a specified channel. A complete list of all commands can be found in the README.md file in the GitHub repository.

**Carlos Tejera** **Alexander Oxley**

# Project: Dezentralized Games

**Abstract:** An Investigation on how to implement turn-based online multiplayer games using append-only logs instead of centralized servers

**Github:** `https://github.com/cn-uofbasel/BACnet/tree/master/redez-sem-hs20/groups/05-decentGames`

## Project Idea

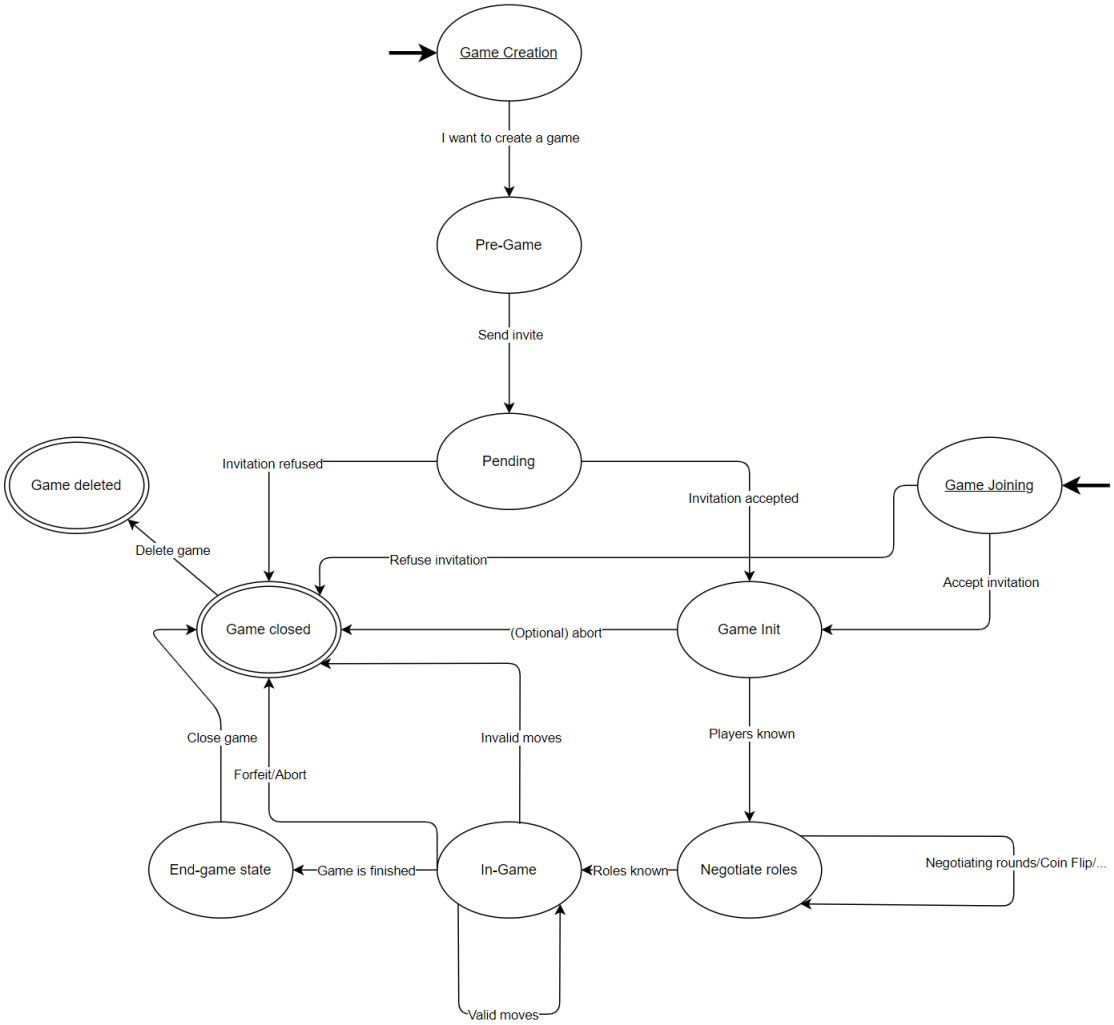
Traditionally, online multiplayer games use centralized servers for the communication between clients and for data storage. As a part of our multi-semester research into the re-decentralization of internet communication and other internet usage, we implemented three examples of online turn-based multiplayer games using decentralized means. Our games no longer communicate with a server and store their data using logs. Clients communicate with each other and logs are then appended-on after each turn.

## Components

- CLI chess, implemented with chessnut
- GUI Mensch ärgere dich nicht, integrated into BACNET
- CLI Mensch ärgere dich nicht, implemented with RCP

## State Diagram

The state diagram shows the different states that must be considered when programming a game. There are two initial states when a game is created or when a game will be joined. The game will be initiated when all players accepted the invitation to get to the negotiation of roles (see section The Question of Random Dice Throws and Role Assignment). After that, the game is ready to be played. The goal states are when a game is closed (respectively finished, aborted or refused) and when a game is deleted. When a game is closed, there is the possibility to take a look at the end-game, otherwise it can be deleted.




## Deployment / Integration

To run the project, run the Main.py file and enter the command /play or /create. To get more help, take a look at our Readme file.

## Our RPC system

We implemented a kind of RPC system where every player has a port on which it is listening to calls. Every time a move is made the players get notified about it and request for the move respectively the update of the game board. It is not necessary that everybody must be "online" when playing. Moves can be made and the other players can still fetch the update even though they were offline when the move was made. The player with the made move must be listening to the fetch requests though. In our case, we retrieve the file with the update to check if it was a valid move and append it to our own file. Nevertheless, it would be also possible to only send the move, so that the receivers could recreate the game board and check its validation.

## Bacnet Integration

Bacnet Integration was rather difficult, because the network wasn't originally made to support such functionality. In our readme, we describe how to setup master feeds, the trusting and feed sharing mechanism. We used two 'hacks' to make bacnet support a game:

- The game feed, seperate from the master feed, is technically called a 'chat' feed, because otherwise the feed isn't recognized. The feed itself is also built like a chat feed, so that each 'turn' is treated like a chat 'message'.

  The setup of a message/turn/log is: colour#split:#dicenumber.

- To speed up the bacnet udp log sharing, in the game we share logs with LEFT KEY and receive logs with RIGHT KEY. This is only possible with trusted game feeds, not the master feed.

## The Question of Random Dice Throws and Role Assignment

During the Hackathon, we discussed the question of random dice throws. In an ideal implementation of a BACNET game, the question of integrity of the turns is important. But how can we verify a random dice throw? Can we verify that it is truly random, how can we stop players from giving themselves a high throw every turn?

One solution mentioned was to use S/Key, using repeated hashing.

Another solution mentioned was to use a commitment scheme, where the client has to commit to a dice number while keeping it hidden to the other players. Commitment schemes mean the player cannot change the dicethrow, meaning it is secure.

The best solution we think, is the one Dr Scherb thought of: a preagreed 'seed' of all future throws, and then random turns.

### Further Work

Up to now, Secure Team Chat only allows inviting users to a group. In the future, it might be helpful to implement commands which enable the owner to pass on his role, close a channel, throw out participants or add the possibility for users to leave a group on their own.
For extremely secure messages, an option for self-destruction after some time might be useful.

**Günes Aydin** **Pascal Bürklin** **Damian Knuchel**

# Project: Decentralized TCP

**Abstract:** This project aims to give a link between append-only log based decentralized networks and applications using the tcp-layer. The main idea behind the link is, to create a proxy that opens the gate to the current Internet making it somewhat backward-compatible.

**Github:** https://github.com/cn-uofbasel/BACnet/tree/master/redez-sem-hs20/groups/07-decentTCP

## Project Idea

The Idea behind this Proxy is to connect the current internet with BACnet, to make it accessible from inside the append-only log infrastructure. It is meant to be setup as service beacons inside the decentralized network, to ensure scalability as well as usability.

## Components

This application uses two logs per user to mirror the tcp-packets. Currently their content is stored without being encrypted, but the log-setup is already creating key pairs to enable further improvements more easily. In every log-pair one log is created by the client and the other by the proxy, both are used as outgoing data-channels for each side. Those pairs are linked by the master-feed, which is created on the proxy side. To authorize a client, its public key is stored in a file named `.authorized_feeds`.

## Deployment / Integration

The events as well as the append-only log files are provided by the BACnet code. The synchroization of those files is currently achieved by continuously reading all of the clients (outgoing channel) logs. To run the tcp-echo test one first has to run the proxy to create its public key, then it has to be pasted into the demo script (at the `REMOTE_FEED`) further the client has to be run once to create its public key, then the client has to be authorized by invoking `proxy.py -a <pubkey>` once this is done the proxy has to be started (`proxy.py`) as well as the echo client `echo_client.py`