Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CS7NS1 Scalable Computing
Project 3 Report

Group 18 - Pragati, Ted, Xin, Yi
26th of November, 2023

Group 18

Pragati Aboti - 23339410 - M.Sc Data Science
Ted Johnson - 19335618 - M.Sc CS (5 year)
Xin Wang - 23361770 - M.Sc Futured Networking Systems
Yi Yang - 23330087 - M.Sc Data Science

Declaration

We did not make use of AI tools in the preparation of this assignment.

1 Introduction

1.1 Problem

This project tasked us with designing and implementing a sufficiently scalable and secure peer-to-peer (P2P) networking protocol based on Information-Centric Networking (ICN) principles. The implementation is to be deployed to the Raspberry Pi cluster, with the requirement to demonstrate its ability to overcome typical scalability challenges, such as limited computing and networking resources, intermittent peer connectivity and device power failure.

1.2 Chosen Scenario

In order to guide our solution towards more practical work, it was necessary for our project to target a realistic, highly-disconnected and scalable use case. We have chosen to imagine fleets of near-future mobile construction and repair drones operating within hazardous or inaccessible environments. Specifically, we aimed to facilitate the secure monitoring and control of the fleet without the need for direct drone-to-operator communication. That is, messages need to be produced, relayed and consumed across ad hoc networks by utilising intelligent discovery and routing techniques. On the surface, this system consists of three applications:

- **Drones:** Reports vehicle sensor readings and executes received commands
- **Inspectors:** Collect readings of known drones and publish alerts and reports
- **Controllers:** Operator terminals which notifications alerts and send commands

We believe this is both a plausible use case for a P2P ICN protocol as well as a suitable environment to test the implementation’s scalability capabilities with the expectation that computing power similar to a Raspberry Pi would be available. One can envision this deployed as several handheld controller devices interacting with a decentralised and highly-disconnected wireless network of low-cost inspectors and active construction drones, granting the ability for entities to remain connected even if all but one other device is outside of communication range.

1.3 Additional Objectives

As a supplemental goal, we also wished to achieve cross-team communication and protocol interoperability. This allows for the Raspberry Pis of many teams to participate in a shared P2P ICN network, providing much higher resilience to intermittent peer networking and opening up the possibility for multi-path routing. Furthermore, we decided to pursue end-to-end (E2E) data encryption and identity validation based on a symmetric key common to invited group members exchanged through Public Key Cryptography (PKC).

2 Solution

2.1 Overview

Fundamentally, our networking architecture consists of two types of entities: A *node*, which simply facilitates connectivity between its peers on the network, and a *client*, which is actually just a node with a unique identity. Applications, namely drones, inspectors and controllers, operate on top of these clients on the network. Nodes are entirely transparent to these applications, as they only need to concern themselves with the labelled data they are interested in and the identities of clients they plan to perform group cryptography with.

With this established, the core of our solution is based on the discovery of peers and clients through UDP broadcasting and the propagation of interests and interest responses towards only relevant clients through TCP messages. In the following sections, we explain the purpose of each of these functionalities and how they attempt to address the assignment requirements.

2.2 Peer Discovery

At regular intervals, every node broadcasts a message to a specific port over UDP to everyone in the same IP subnetwork as it. If there are other nodes listening on this specific port which support our protocol, they now know that this node exists and can be found at the source address and port. This mechanism allows peers to find each other on the subnetwork without requiring every address and port to be queried. This significantly reduces the networking overhead needed to discover peers. We see in section 3 how this is optimised and in section 4.2 how it could be improved.

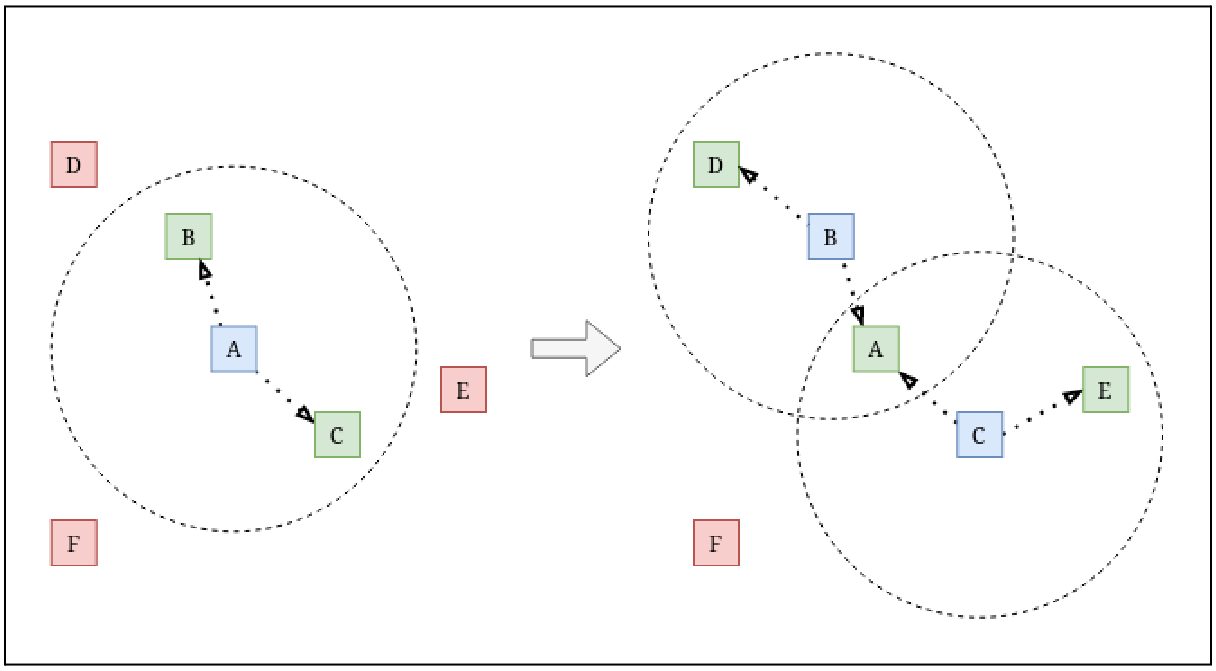


Fig 2.2.1 - A's broadcast is heard by B and C. Later, B's broadcast is heard by A and D…

2.3 Client Advertising

As an extension to the peer discovery broadcasts above, the identity of clients can be distributed throughout the network. Clients include their identity and data labels they publish to in their regular UDP broadcasts. They also include a score initially set to 1,000. Nodes which receive this advertisement must then include it in their next UDP broadcast, decreasing the score appropriately to reflect how capable they believe they are in forwarding messages towards the client. In this way, the entire network eventually learns of the existence of the new client as well as which peers are the most optimal to route messages via in order to reach the client. Again, we will see in section 3 how this method is implemented in a performant manor.

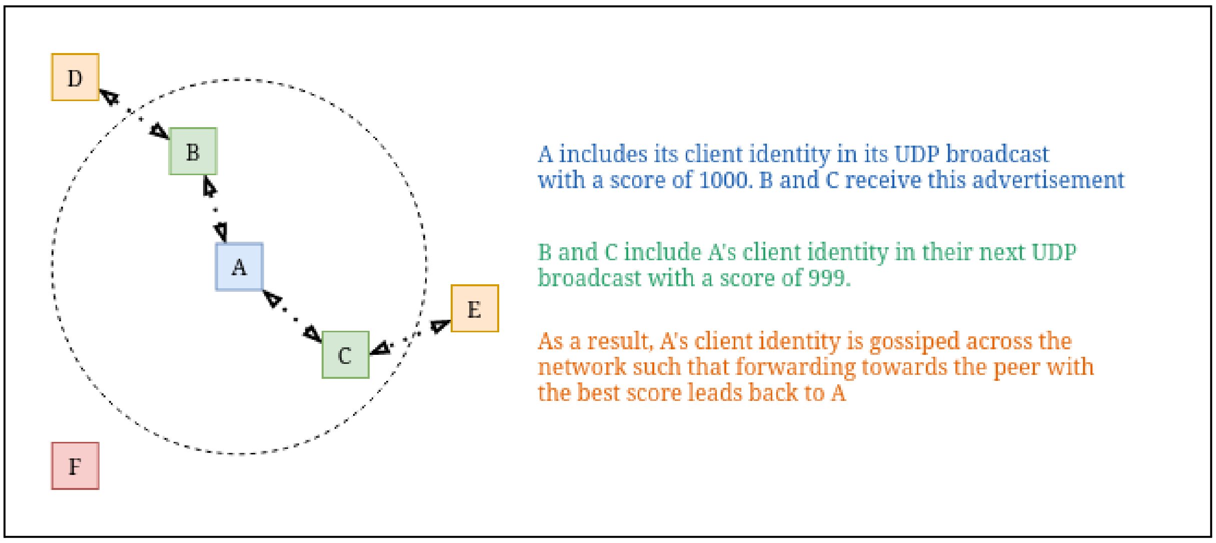

D
B
A
C
E
F
A includes its client identity in its UDP broadcast with a score of 1000. B and C receive this advertisement
B and C include A's client identity in their next UDP broadcast with a score of 999.
As a result, A's client identity is gossiped across the network such that forwarding towards the peer with the best score leads back to A

Fig 2.3.1 - An example of a client's identity being propagated through a network

2.4 Subscriber / Publisher Pattern

With the above UDP broadcast mechanisms, each node in the network eventually obtains a list of known peers and known clients, as well as collecting an understanding of how to best route messages via peers to get to a client. We now introduce the subscriber / publisher networking pattern on top of this platform as a means of implementing ICN functionality. In this project, a subscriber issues notifications of interest in some labelled data (indicated as "get" messages in our code). As we already know which clients have advertised as potentially publishing to this label, we are able to propagate this notification of interest over TCP connections directly towards these clients via the best path. Furthermore, if a fulfilment for this interest is encountered, whether at the publisher client or cached in an intermediate node, the response can be similarly propagated back towards the interested clients over TCP via the best paths. Along the way, the labelled data will be temporarily cached within the intermediate nodes in anticipation for more interest.

This dramatically reduces the number of TCP connections needed to be established when compared to naive gossiping of interest, while still providing a reliable (best-effort) communication channel for interests and data to pass through. This is undoubtedly true in instances of dense mesh-like networks. Note that in the event of TCP communication failure, such as a peer leaving the network, nodes will then choose the next best peer to reach the target destination - This may even result in backtracking in the exhaustive case.

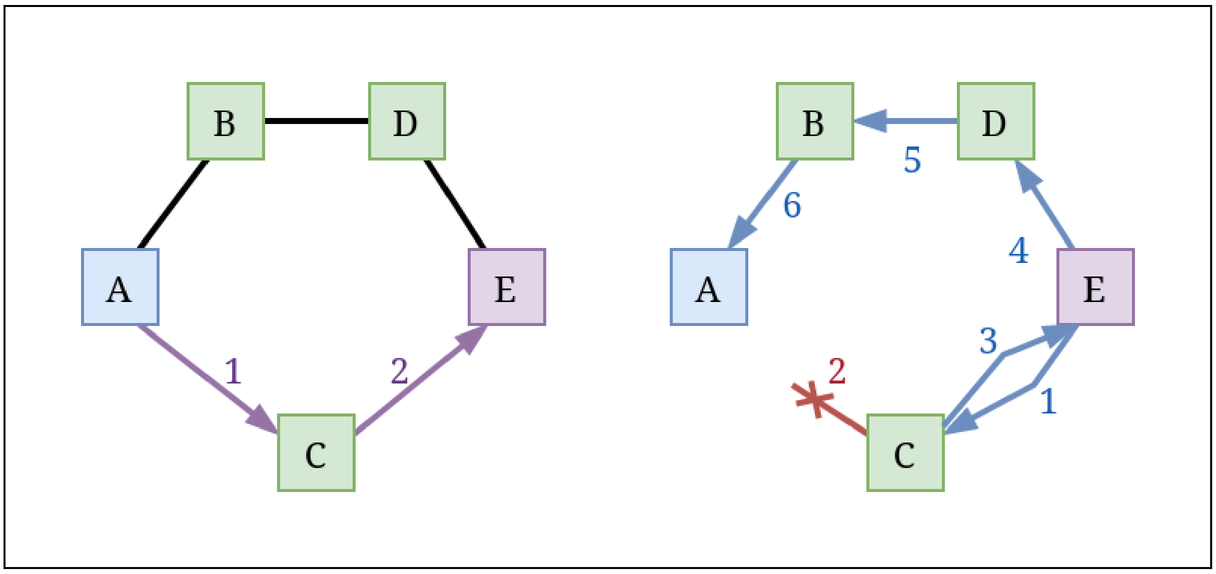


Fig 2.4.1 - A's interest pushed towards E, E's response detoured back towards A

2.5 Group Cryptography

To provide secure communication between clients on our ICN-based network, we first needed to determine our desired security model. For our scenario, it makes most sense to prevent eavesdroppers from either reading or interfering with the data. It is also important to prevent malicious users from masquerading as another identity. However, we decided it was not important to provide a centralised authority for verifying these identities, and instead choose to rely on pre-established trusted public keys. Furthermore, we adopted an entirely decentralised group establishment mechanism, in which any member client can invite any other trusted client on the network. We comment on this further in section 4.

As an added constraint, all client-to-client group communication occurs on top of our clear P2P ICN protocol. There are no special message formats or alternative communication channels used to establish and use client groups. Instead, group symmetric encryption keys are generated, shared and negotiated secretly between prospective member clients attempting to join together. This is accomplished through each client publishing an "invite list" for each group they belong to. This list potentially contains the group's symmetric key encrypted with each invited client's public key. In this way, any group member can invite any trusted clients. Note that all of this is using the properties of PKC, so we can also provide author authentication of the invite list by signing with a private key and presenting the signature alongside.

Two edge cases that need to be addressed are the establishment of an entirely new group and the merging of groups which had initially been established in isolation of each other, therefore members possessing entirely different keys. These cases are both handled in a similar fashion: Upon receiving a new invite, clients will compare the creation date of the new group key with the one they already possess:

1. **Neither invite or client have a key:** Generate new key and update invite lists.
2. **Invite has a newer key than client:** Accept new key and update invite lists.
3. **Invite has an older key than client, or no key at all:** Ignore.

The first clause handles exactly the case that neither client is yet part of the specified group as neither possesses a key yet, so a new group is established by generating a new key. The original inviter might then encounter the second clause, as they receive the new invite containing the new group key from the client they invited with no key. Finally, notice that if two groups were established in isolation and then later came into contact through mutual trusted clients, the second and third clause will ensure that only the newer group key will remain after a brief period of negotiation.

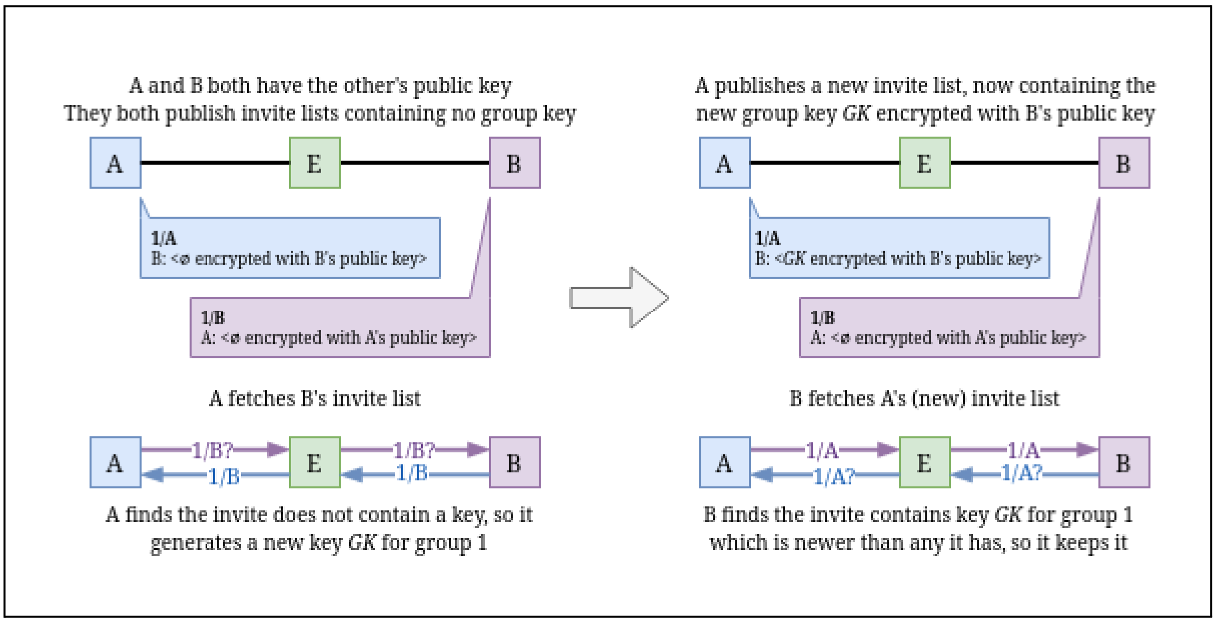

A and B both have the other's public key
They both publish invite lists containing no group key
A
E
B
1/A
B: <ø encrypted with B's public key>
1/B
A: <ø encrypted with A's public key>
A fetches B's invite list
1/B?
1/B
A finds the invite does not contain a key, so it generates a new key GK for group 1
A publishes a new invite list, now containing the new group key GK encrypted with B's public key
1/A
B: <GK encrypted with B's public key>
1/B
A: <ø encrypted with A's public key>
B fetches A's (new) invite list
1/A
1/A?
B finds the invite contains key GK for group 1 which is newer than any it has, so it keeps it

Fig 2.5.1 - Walkthrough of how A and B can form a new group called 1

3 Implementation

3.1 tcdicn

Almost immediately after beginning work on this project, we decided to work together with about five other teams on formulating ideas and conceptualising potential solutions. During one of our early inter-team meetings, the proposal to work on a shared protocol capable of unifying our Raspberry Pi devices was put forward. This plan quickly took off so during the following meeting, we each discussed what requirements such a protocol would need in order to meet each participating team's use cases. Eventually, it was decided that we should attempt to maximise both ease-of-use and flexibility, and that development would be split into three phases:

- **v0.1:** Peer discovery based on UDP broadcasting and prototype ICN functionality
- **v0.2:** Subscriber / Publisher pattern with smart interest and fulfilment routing
- **v0.3:** Optional group encryption based on a symmetric key shared using PKC

The protocol was to be realised as JSON-encoded text that could be sent over both UDP and TCP. Each JSON message was to consist of a version number and a list of "message items". The version number allows different implementations to co-exist on the network without interfering with each other. Interestingly, it later became apparent that v0.3 would remain entirely compatible with v0.2, so the version number has remained at v0.2. The three types of message items were to be peer advertisements, interests requests and fulfilment responses. During development of v0.1, peer advertisements items were split into both peer and client advertisements.

This team took lead responsibility on implementing the protocol using Python. What was originally planned as a reference implementation soon became a module which could be imported and used within any team's project. As such, in order to remain as forward-compatible as possible, the module API was fixed early as:

```
import tcdicn
node = tcdicn.Node()
await node.start(port, dport, ttl: float, tpf: float, client: dict)
await node.get(label: str, ttl: float, tpf: float, ttp: float)
await node.set(label: str, data: str)
```

This allows application code to start a node with a number of configurable networking parameters and an optional client identity, then to express interest in a label ("get") and publish data to a label ("set"). Later with v0.3 came the inclusion of one additional method:

```
await node.join(group: str, client: str, key: bytes, labels: List[str])
```

This allows applications to join together two clients which can trust the identity of each other using their pre-shared public keys. The result of these work can be found here: https://github.com/tedski999/tcdicn

3.2 TTL & TPF

One of the technicalities that arose during development of our protocol was how to handle nodes disappearing from the network if a majority of state communication was accomplished over UDP broadcasting. The solution we devised was to give every peer advertisement, client advertisement and notification of interest a Time To Live (TTL). This specifies how long the intent of the message should last for. That is, peers, clients and interests are not expected to be resent before their previous TTL expires, but, upon expiry, any state associated with the message is completely forgotten. This is necessary to remove old message states such as from peers which have since gone offline.

One wrinkle is that messages need to be resent *before* their previous TTL expires rather than *upon* expiry. Otherwise, the previous message state would very likely be erased right before being refreshed causing unbecoming flapping. Worse still, if even a single message gets lost or delayed, the intent of that message will be missing from the network until the next TTL expiry. Therefore, it is necessary to configure a TTL PreFire (TPF) which sets how many times the message intent should be transmitted before the next TTL expiry. That is, messages will instead be resent every TTL/TPF seconds.

A further wrinkle is that TTL is actually sent as End Of Life (EOL). These EOLs specify the absolute time that a message intent should be forgotten, instead of the duration it should be kept. This ensures both that messages do uniformly expire across the network and that messages with the same EOL will be treated as duplicates and ignored.

3.3 TTP & Batching

In order to reduce the number of UDP broadcasts and TCP connections required to perform this protocol across a network, Time To Propagate (TTP) was introduced. This variable permits nodes and clients to refrain from forwarding or responding to your messages for up to TTP seconds. This small buffer period allows the node time to aggregate potentially many messages from potentially many sources into a single transmission. For example, both many interest messages and data response messages can be batched together if they are destined to the same peer and each TTP permits it, as the one TCP connection can be used to transmit all at once. Similarly, upon receiving many separate client advertisement broadcasts within the last second, if permitted, a node preferes to batch all these advertisements into one single broadcast. However, note that UDP broadcast payloads are capped to 512 bytes to reduce the likelihood of the datagram from being fragmented due to exceeding the network's Maximum Transmission Unit (MTU), which increase datagram loss rates.

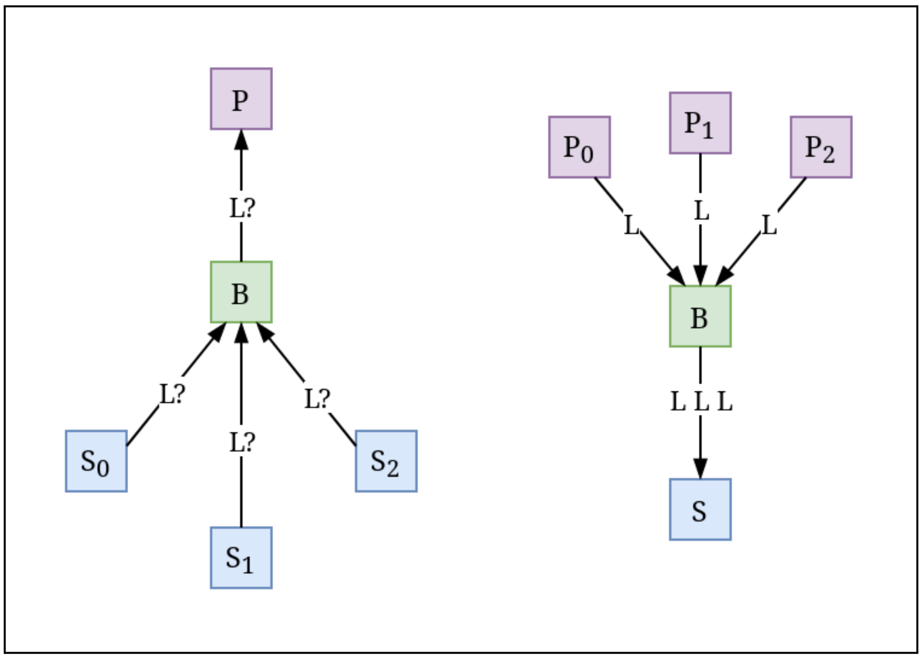


Fig 3.3.1 - Batching reduces TCP connections by merging payloads to the same peer

3.4 Deployment

Some challenges were faced while deploying to the Raspberry Pi cluster in accordance with the assignment specifications. The earliest issue we encountered was determining how we could host multiple clients and nodes on a single machine, as only one instance would be able to bind to the designated port. We determined that the best solution for our project was to allow instances to bind and listen on any port on the system but to share one “discovery” port, such as :33333. That is, each process occupied its own port but would only broadcast towards the designated discovery port. In this way, there would only be one “main” node on the machine which listens for peers and client adverts on port 33333 (AKA discovery), while other nodes could still broadcast to port 33333 which allows them to be discovered, but would listen on non-standard ports and therefore could not do their own discovery. Instead, they must route all outgoing TCP messages via the main node on the system. This is a somewhat undesirable dependency, but it is an adequate solution for deploying multiple nodes as multiple processes on a single machine.

It was also necessary to write a Systemd user service file for the Raspberry Pis to allow node processes to remain running after the user has logged off, the process crashes or if the machine reboots. It is possible to enable this user service with the following commands:

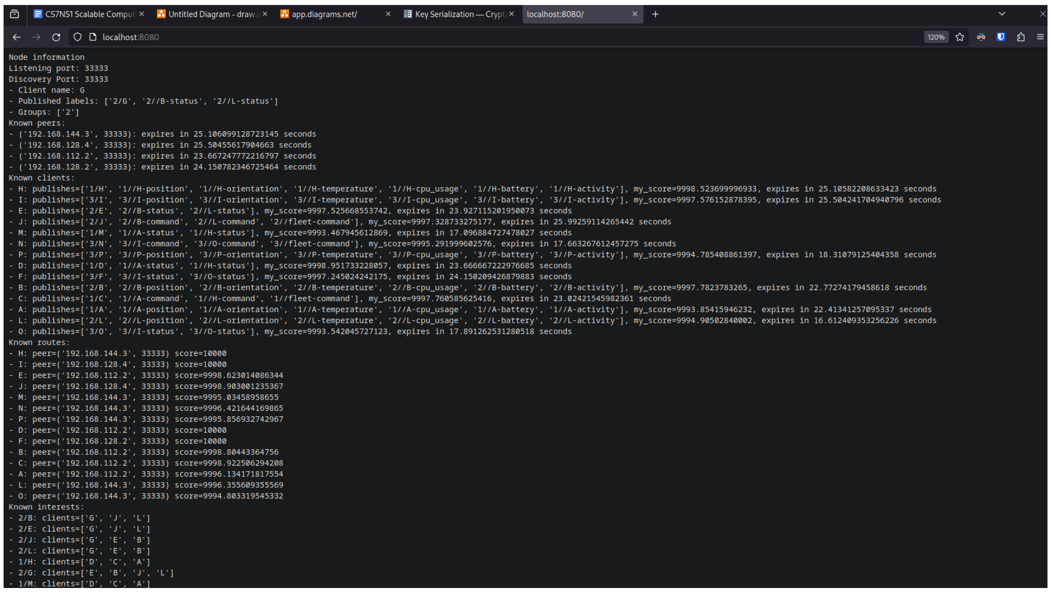
```
mkdir -p ~/.config/systemd/user/
cp ~/tcdicn/resources/tcdicn.service ~/.config/systemd/user/
loginctl enable-linger  # So our service stay running after logout
systemctl --user start tcdicn  # Start our service
```

Finally, as each client must possess at least one other client's public key to join a group with them, it is also necessary to exchange these public keys over another channel. This channel does not need to be private, but it must provide author authentication (i.e. prevent tampering of the key). This is required because the tcdicn protocol by itself cannot guarantee a message has not been manipulated by an intermediate node until the communicating nodes form a group.

3.5 Simulation & Evaluation

While testing our implementation on the provided Raspberry Pi's is possible, we found it to be excessively cumbersome and lacked adequate coverage for many of the scalability challenges we were aiming to solve. As such, we chose to simulate much of our testing within virtual networks offered by modern software containerisation. In particular, we leveraged Docker to deploy our applications, clients and nodes as isolated containers only connected via a limited number of bridge networks. This allowed us to easily test our implementation in various network configurations and against communication failures without limiting ourselves to what is available on the cluster. Of most use was the ability to disable and re-enable individual nodes in order to observe if and how our protocol adapts. This significantly helped us investigate and triangulate implementation bugs and obscure edge cases.

In addition to this, we invested some time into integrating an optional debug web server into every node which displays the current state of the node. This was again very useful in understanding what exactly was occurring and where something could have gone wrong. This functionality is available by prepending "TCDICN_WPORT=<port>" before any python3 execution command, and then navigating to "http://localhost:<port>" in your web browser (You may need to expose the corresponding port if within Docker, or tunnel your SSH connection if running on one of the Raspberry Pis)



```
Node information
Listening port: 33333
Discovery Port: 33333
- Client name: G
- Published labels: ['2/G', '2//B-status', '2//L-status']
- Groups: ['2']
Known peers:
- ('192.168.144.3', 33333): expires in 25.106099128723145 seconds
- ('192.168.128.4', 33333): expires in 25.50455617904663 seconds
- ('192.168.112.2', 33333): expires in 23.667247772216797 seconds
- ('192.168.128.2', 33333): expires in 24.150782346725464 seconds
Known clients:
- H: publishes=['1/H', '1//H-position', '1//H-orientation', '1//H-temperature', '1//H-cpu_usage', '1//H-battery', '1//H-activity'], my_score=9998.523699996933, expires in 25.10582208633423 seconds
- I: publishes=['3/I', '3//I-position', '3//I-orientation', '3//I-temperature', '3//I-cpu_usage', '3//I-battery', '3//I-activity'], my_score=9997.576152878395, expires in 25.504241704940796 seconds
- E: publishes=['2/E', '2//B-status', '2//L-status'], my_score=9997.525668553742, expires in 23.927115201950073 seconds
- J: publishes=['2/J', '2//B-command', '2//L-command', '2//fleet-command'], my_score=9997.328733275177, expires in 25.99259114265442 seconds
- M: publishes=['1/M', '1//A-status', '1//H-status'], my_score=9993.467945612869, expires in 17.096884727478027 seconds
- N: publishes=['3/N', '3//I-command', '3//O-command', '3//fleet-command'], my_score=9995.291999602576, expires in 17.663267612457275 seconds
- P: publishes=['3/P', '3//P-position', '3//P-orientation', '3//P-temperature', '3//P-cpu_usage', '3//P-battery', '3//P-activity'], my_score=9994.785408861397, expires in 18.31079125404358 seconds
- D: publishes=['1/D', '1//A-status', '1//H-status'], my_score=9998.951733228057, expires in 23.666667222976685 seconds
- F: publishes=['3/F', '3//I-status', '3//O-status'], my_score=9997.245024242175, expires in 24.150209426879883 seconds
- B: publishes=['2/B', '2//B-position', '2//B-orientation', '2//B-temperature', '2//B-cpu_usage', '2//B-battery', '2//B-activity'], my_score=9997.7823783265, expires in 22.77274179458618 seconds
- C: publishes=['1/C', '1//A-command', '1//H-command', '1//fleet-command'], my_score=9997.760585625416, expires in 23.02421545982361 seconds
- A: publishes=['1/A', '1//A-position', '1//A-orientation', '1//A-temperature', '1//A-cpu_usage', '1//A-battery', '1//A-activity'], my_score=9993.85415946232, expires in 22.41341257095337 seconds
- L: publishes=['2/L', '2//L-position', '2//L-orientation', '2//L-temperature', '2//L-cpu_usage', '2//L-battery', '2//L-activity'], my_score=9994.90502840002, expires in 16.612409353256226 seconds
- O: publishes=['3/O', '3//I-status', '3//O-status'], my_score=9993.542045727123, expires in 17.891262531280518 seconds
Known routes:
- H: peer=('192.168.144.3', 33333) score=10000
- I: peer=('192.168.128.4', 33333) score=10000
- E: peer=('192.168.112.2', 33333) score=9998.623014086344
- J: peer=('192.168.128.4', 33333) score=9998.903001235367
- M: peer=('192.168.144.3', 33333) score=9995.03458958655
- N: peer=('192.168.144.3', 33333) score=9996.421644169865
- P: peer=('192.168.144.3', 33333) score=9995.856932742967
- D: peer=('192.168.112.2', 33333) score=10000
- F: peer=('192.168.128.2', 33333) score=10000
- B: peer=('192.168.112.2', 33333) score=9998.80443364756
- C: peer=('192.168.112.2', 33333) score=9998.922506294208
- A: peer=('192.168.112.2', 33333) score=9996.134171817554
- L: peer=('192.168.144.3', 33333) score=9996.355609355569
- O: peer=('192.168.144.3', 33333) score=9994.803319545332
Known interests:
- 2/B: clients=['G', 'J', 'L']
- 2/E: clients=['G', 'J', 'L']
- 2/J: clients=['G', 'E', 'B']
- 2/L: clients=['G', 'E', 'B']
- 1/H: clients=['D', 'C', 'A']
- 2/G: clients=['E', 'B', 'J', 'L']
- 1/M: clients=['D', 'C', 'A']
```

Fig 3.5.1 - Screenshot of debug web server response running on port 8080

4 Discussion

4.1 Results

Our project fulfils all the goals we set ourselves, including overcoming scalability issues, implementing UDP peer discovery for P2P networking, providing ICN functionality based on the subscriber / publisher networking pattern, developing group cryptography backed by a shared symmetric key and established through PKC on top of the ICN protocol, and cross-team interoperability through an open standard and reference implementation. We are happy with the results we have accomplished through our research and technical work.

4.2 Limitations & Future Work

Considering the assignment timescale and the additional goals we set ourselves, we believe our submission to have achieved all learning objectives in sufficient merit. However, it is certainly not a perfect design nor implementation. Looking back, there are a number of design decisions and technical aspects we would have liked to have done differently given either our current knowledge. With more time to research and test, we can see that most of these are conceivable with only some changes to our codebase.

One of the most obvious changes is to use UDP multicast instead of UDP broadcast. This immediately lets us upgrade to support IPv6 and eliminates a very large majority of the network noise currently caused by this protocol. By extension, this would also reduce a lot of the CPU time currently wasted by every device on the subnetwork needing to ingress our broadcasted datagrams.

Another straightforward improvement to the implementation would be to improve the client route scoring system with a method to determine the current congestion felt on the node. Instead of simply subtracting the route score by one, the score should take into consideration the current networking and computing load the node is under so that alternative routes can be taken in deemed suitable. To gain some of these benefits, our implementation already introduces a random variable into the score subtraction in order to break ties and use similar scoring routes that might be under utilised.

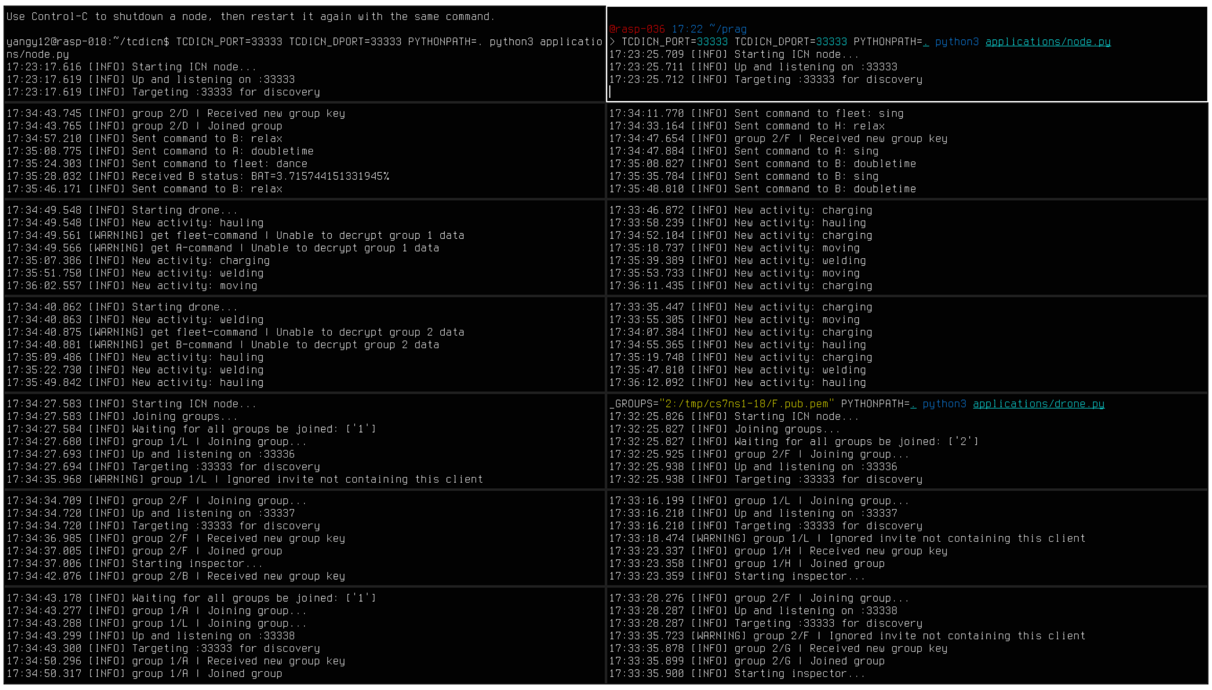
An interesting problem we are unaware of how to solve today without compromising our group cryptography security is the encryption of labels. As of now, only data is encrypted between group clients, while the label must remain in the clear. This is because our implementation depends on each data mapping to exactly one label, which does not allow for non-deterministic label encryption. However, deterministic encryption is not supported within the cryptography library we are using. It would also significantly reduce the strength of our cryptosystem as ciphertext analysis on deterministically encrypted labels could potentially reveal bits of the group symmetric key.

One suggestion provided by Dr. Ciarán Mc Goldrick during our demonstration was the use of distributed ledgers in order to avoid granting any single group member unregulated privilege to add new members, which could jeopardise the group. While a significant amount of work to implement, it does also help to alleviate the catastrophe of a single compromised private key, which a malicious user could use to masquerade as the group member and gain access to the newest group key. Currently, the only method to combat this situation is through publishing a private key revocation certificate to alert other members that your key can no longer be trusted, and then generating both a new keypair and a new group key.

5 Demonstration

5.1 With Raspberry Pi Cluster

It is possible to deploy our implementation to the Raspberry Pi cluster with the two scripts located in ./simulations/pi/. These scripts set up five clients and one main node as in accordance with the solution discussed in section 3.4. The two scripts must be run on different Raspberry Pi's, then two public keys must be manually exchanged. This allows the two networks on each machine to trust each other and from the two defined groups. See the project README and scripts for more details.



```
Use Control-C to shutdown a node, then restart it again with the same command.

yangy12@rasp-018:~/tcdicn$ TCDICN_PORT=33333 TCDICN_DPORT=33333 PYTHONPATH=. python3 applicatio
ns/node.py
17:23:17.616 [INFO] Starting ICN node...
17:23:17.619 [INFO] Up and listening on :33333
17:23:17.619 [INFO] Targeting :33333 for discovery
```

```
17:34:43.745 [INFO] group 2/D | Received new group key
17:34:43.765 [INFO] group 2/D | Joined group
17:34:57.210 [INFO] Sent command to B: relax
17:35:08.775 [INFO] Sent command to A: doubletime
17:35:24.303 [INFO] Sent command to fleet: dance
17:35:28.032 [INFO] Received B status: BAT=3.715744151331945%
17:35:46.171 [INFO] Sent command to B: relax
```

```
17:34:49.548 [INFO] Starting drone...
17:34:49.548 [INFO] New activity: hauling
17:34:49.561 [WARNING] get fleet-command | Unable to decrypt group 1 data
17:34:49.566 [WARNING] get A-command | Unable to decrypt group 1 data
17:35:07.386 [INFO] New activity: charging
17:35:51.750 [INFO] New activity: welding
17:36:02.557 [INFO] New activity: moving
```

```
17:34:40.862 [INFO] Starting drone...
17:34:40.863 [INFO] New activity: welding
17:34:40.875 [WARNING] get fleet-command | Unable to decrypt group 2 data
17:34:40.881 [WARNING] get B-command | Unable to decrypt group 2 data
17:35:09.486 [INFO] New activity: hauling
17:35:22.730 [INFO] New activity: welding
17:35:49.842 [INFO] New activity: hauling
```

```
17:34:27.583 [INFO] Starting ICN node...
17:34:27.583 [INFO] Joining groups...
17:34:27.584 [INFO] Waiting for all groups be joined: ['1']
17:34:27.680 [INFO] group 1/L | Joining group...
17:34:27.693 [INFO] Up and listening on :33336
17:34:27.694 [INFO] Targeting :33333 for discovery
17:34:35.968 [WARNING] group 1/L | Ignored invite not containing this client
```

```
17:34:34.709 [INFO] group 2/F | Joining group...
17:34:34.720 [INFO] Up and listening on :33337
17:34:34.720 [INFO] Targeting :33333 for discovery
17:34:36.985 [INFO] group 2/F | Received new group key
17:34:37.005 [INFO] group 2/F | Joined group
17:34:37.006 [INFO] Starting inspector...
17:34:42.076 [INFO] group 2/B | Received new group key
```

```
17:34:43.178 [INFO] Waiting for all groups be joined: ['1']
17:34:43.277 [INFO] group 1/A | Joining group...
17:34:43.288 [INFO] group 1/L | Joining group...
17:34:43.299 [INFO] Up and listening on :33338
17:34:43.300 [INFO] Targeting :33333 for discovery
17:34:50.296 [INFO] group 1/A | Received new group key
17:34:50.317 [INFO] group 1/A | Joined group
```

```
@rasp-036 17:22 ~/prag
> TCDICN_PORT=33333 TCDICN_DPORT=33333 PYTHONPATH=. python3 applications/node.py
17:23:25.709 [INFO] Starting ICN node...
17:23:25.711 [INFO] Up and listening on :33333
17:23:25.712 [INFO] Targeting :33333 for discovery
```

```
17:34:11.770 [INFO] Sent command to fleet: sing
17:34:33.164 [INFO] Sent command to H: relax
17:34:47.654 [INFO] group 2/F | Received new group key
17:34:47.884 [INFO] Sent command to A: sing
17:35:08.827 [INFO] Sent command to B: doubletime
17:35:35.784 [INFO] Sent command to B: sing
17:35:48.810 [INFO] Sent command to B: doubletime
```

```
17:33:46.872 [INFO] New activity: charging
17:33:58.239 [INFO] New activity: hauling
17:34:52.104 [INFO] New activity: charging
17:35:18.737 [INFO] New activity: moving
17:35:39.389 [INFO] New activity: welding
17:35:53.733 [INFO] New activity: moving
17:36:11.435 [INFO] New activity: charging
```

```
17:33:35.447 [INFO] New activity: charging
17:33:55.305 [INFO] New activity: moving
17:34:07.384 [INFO] New activity: charging
17:34:55.365 [INFO] New activity: hauling
17:35:19.748 [INFO] New activity: charging
17:35:47.810 [INFO] New activity: welding
17:36:12.092 [INFO] New activity: hauling
```

```
_GROUPS="2:/tmp/cs7ns1-18/F.pub.pem" PYTHONPATH=. python3 applications/drone.py
17:32:25.826 [INFO] Starting ICN node...
17:32:25.827 [INFO] Joining groups...
17:32:25.827 [INFO] Waiting for all groups be joined: ['2']
17:32:25.925 [INFO] group 2/F | Joining group...
17:32:25.938 [INFO] Up and listening on :33336
17:32:25.938 [INFO] Targeting :33333 for discovery
```

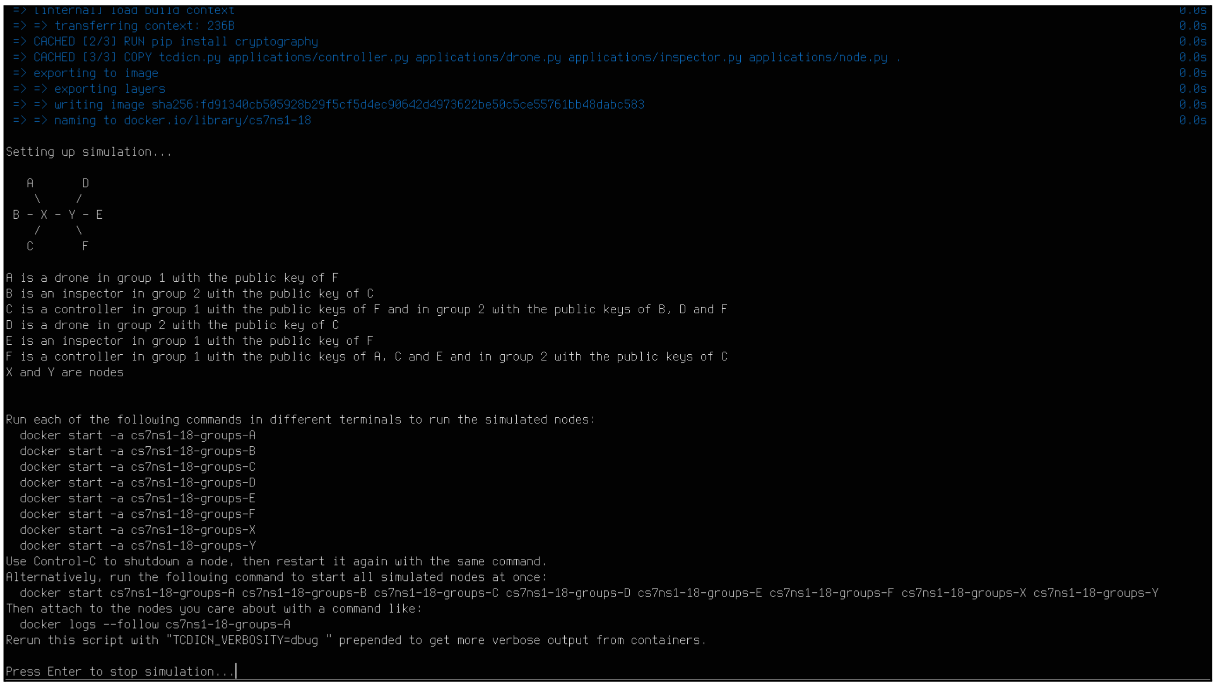
```
17:33:16.199 [INFO] group 1/L | Joining group...
17:33:16.210 [INFO] Up and listening on :33337
17:33:16.210 [INFO] Targeting :33333 for discovery
17:33:18.474 [WARNING] group 1/L | Ignored invite not containing this client
17:33:23.337 [INFO] group 1/H | Received new group key
17:33:23.358 [INFO] group 1/H | Joined group
17:33:23.359 [INFO] Starting inspector...
```

```
17:33:28.276 [INFO] group 2/F | Joining group...
17:33:28.287 [INFO] Up and listening on :33338
17:33:28.287 [INFO] Targeting :33333 for discovery
17:33:35.723 [WARNING] group 2/F | Ignored invite not containing this client
17:33:35.878 [INFO] group 2/G | Received new group key
17:33:35.899 [INFO] group 2/G | Joined group
17:33:35.900 [INFO] Starting inspector...
```

Fig 5.1.1 - Screenshot of ./simulations/pi/.sh running on rasp-018 and rasp-036*

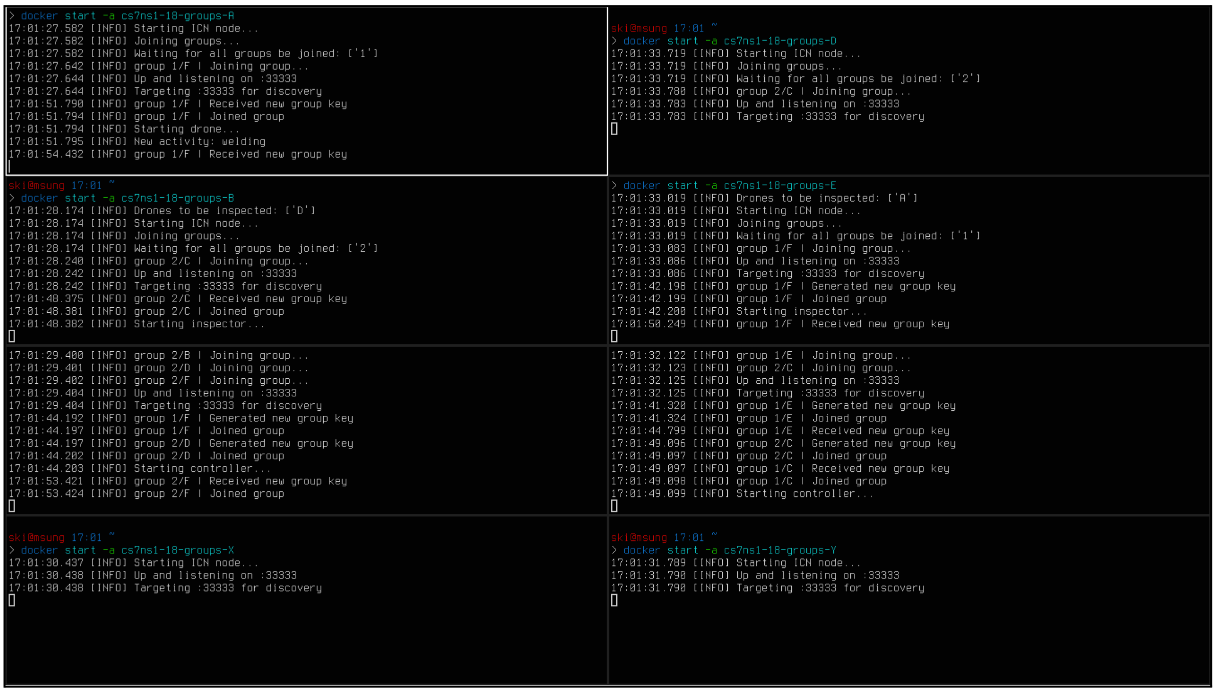
5.2 With Docker

For easier setup and control, we suggest instead deploying our applications as containers using Docker with any of the provided scripts in ./simulations/docker/. Each script sets up a different network configuration: *basic.sh* consists of just one drone, one inspector and one controller all in the same group. *groups.sh* showcases how two groups can be set up on the same network, and is similar in topology to the result of the Raspberry Pi cluster scripts. Finally, *huge.sh* is a mess of 16 drones, inspectors and controllers and a node split between three groups. See the projected README and scripts for more details.



```
 => [internal] load build context                                                                                          0.0s
 => => transferring context: 236B                                                                                          0.0s
 => CACHED [2/3] RUN pip install cryptography                                                                              0.0s
 => CACHED [3/3] COPY tcdicn.py applications/controller.py applications/drone.py applications/inspector.py applications/node.py .   0.0s
 => exporting to image                                                                                                     0.0s
 => => exporting layers                                                                                                    0.0s
 => => writing image sha256:fd91340cb505928b29f5cf5d4ec90642d4973622be50c5ce55761bb48dabc583                               0.0s
 => => naming to docker.io/library/cs7ns1-18                                                                               0.0s

Setting up simulation...

   A       D
    \     /
 B - X - Y - E
    /     \
   C       F

A is a drone in group 1 with the public key of F
B is an inspector in group 2 with the public key of C
C is a controller in group 1 with the public keys of F and in group 2 with the public keys of B, D and F
D is a drone in group 2 with the public key of C
E is an inspector in group 1 with the public key of F
F is a controller in group 1 with the public keys of A, C and E and in group 2 with the public keys of C
X and Y are nodes


Run each of the following commands in different terminals to run the simulated nodes:
  docker start -a cs7ns1-18-groups-A
  docker start -a cs7ns1-18-groups-B
  docker start -a cs7ns1-18-groups-C
  docker start -a cs7ns1-18-groups-D
  docker start -a cs7ns1-18-groups-E
  docker start -a cs7ns1-18-groups-F
  docker start -a cs7ns1-18-groups-X
  docker start -a cs7ns1-18-groups-Y
Use Control-C to shutdown a node, then restart it again with the same command.
Alternatively, run the following command to start all simulated nodes at once:
  docker start cs7ns1-18-groups-A cs7ns1-18-groups-B cs7ns1-18-groups-C cs7ns1-18-groups-D cs7ns1-18-groups-E cs7ns1-18-groups-F cs7ns1-18-groups-X cs7ns1-18-groups-Y
Then attach to the nodes you care about with a command like:
  docker logs --follow cs7ns1-18-groups-A
Rerun this script with "TCDICN_VERBOSITY=dbug " prepended to get more verbose output from containers.

Press Enter to stop simulation...
```

Fig 5.2.1 - Screenshot of ./simulations/docker/groups.sh setting up virtual environment



```
> docker start -a cs7ns1-18-groups-A
17:01:27.582 [INFO] Starting ICN node...
17:01:27.582 [INFO] Joining groups...
17:01:27.582 [INFO] Waiting for all groups be joined: ['1']
17:01:27.642 [INFO] group 1/F | Joining group...
17:01:27.644 [INFO] Up and listening on :33333
17:01:27.644 [INFO] Targeting :33333 for discovery
17:01:51.790 [INFO] group 1/F | Received new group key
17:01:51.794 [INFO] group 1/F | Joined group
17:01:51.794 [INFO] Starting drone...
17:01:51.795 [INFO] New activity: welding
17:01:54.432 [INFO] group 1/F | Received new group key
```

```
ski@msung 17:01 ~
> docker start -a cs7ns1-18-groups-D
17:01:33.719 [INFO] Starting ICN node...
17:01:33.719 [INFO] Joining groups...
17:01:33.719 [INFO] Waiting for all groups be joined: ['2']
17:01:33.780 [INFO] group 2/C | Joining group...
17:01:33.783 [INFO] Up and listening on :33333
17:01:33.783 [INFO] Targeting :33333 for discovery
```

```
ski@msung 17:01 ~
> docker start -a cs7ns1-18-groups-B
17:01:28.174 [INFO] Drones to be inspected: ['D']
17:01:28.174 [INFO] Starting ICN node...
17:01:28.174 [INFO] Joining groups...
17:01:28.174 [INFO] Waiting for all groups be joined: ['2']
17:01:28.240 [INFO] group 2/C | Joining group...
17:01:28.242 [INFO] Up and listening on :33333
17:01:28.242 [INFO] Targeting :33333 for discovery
17:01:48.375 [INFO] group 2/C | Received new group key
17:01:48.381 [INFO] group 2/C | Joined group
17:01:48.382 [INFO] Starting inspector...
```

```
> docker start -a cs7ns1-18-groups-E
17:01:33.019 [INFO] Drones to be inspected: ['A']
17:01:33.019 [INFO] Starting ICN node...
17:01:33.019 [INFO] Joining groups...
17:01:33.019 [INFO] Waiting for all groups be joined: ['1']
17:01:33.083 [INFO] group 1/F | Joining group...
17:01:33.086 [INFO] Up and listening on :33333
17:01:33.086 [INFO] Targeting :33333 for discovery
17:01:42.198 [INFO] group 1/F | Generated new group key
17:01:42.199 [INFO] group 1/F | Joined group
17:01:42.200 [INFO] Starting inspector...
17:01:50.249 [INFO] group 1/F | Received new group key
```

```
17:01:29.400 [INFO] group 2/B | Joining group...
17:01:29.401 [INFO] group 2/D | Joining group...
17:01:29.402 [INFO] group 2/F | Joining group...
17:01:29.404 [INFO] Up and listening on :33333
17:01:29.404 [INFO] Targeting :33333 for discovery
17:01:44.192 [INFO] group 1/F | Generated new group key
17:01:44.197 [INFO] group 1/F | Joined group
17:01:44.197 [INFO] group 2/D | Generated new group key
17:01:44.202 [INFO] group 2/D | Joined group
17:01:44.203 [INFO] Starting controller...
17:01:53.421 [INFO] group 2/F | Received new group key
17:01:53.424 [INFO] group 2/F | Joined group
```

```
17:01:32.122 [INFO] group 1/E | Joining group...
17:01:32.123 [INFO] group 2/C | Joining group...
17:01:32.125 [INFO] Up and listening on :33333
17:01:32.125 [INFO] Targeting :33333 for discovery
17:01:41.320 [INFO] group 1/E | Generated new group key
17:01:41.324 [INFO] group 1/E | Joined group
17:01:44.799 [INFO] group 1/E | Received new group key
17:01:49.096 [INFO] group 2/C | Generated new group key
17:01:49.097 [INFO] group 2/C | Joined group
17:01:49.097 [INFO] group 1/C | Received new group key
17:01:49.098 [INFO] group 1/C | Joined group
17:01:49.099 [INFO] Starting controller...
```

```
ski@msung 17:01 ~
> docker start -a cs7ns1-18-groups-X
17:01:30.437 [INFO] Starting ICN node...
17:01:30.438 [INFO] Up and listening on :33333
17:01:30.438 [INFO] Targeting :33333 for discovery
```

```
ski@msung 17:01 ~
> docker start -a cs7ns1-18-groups-Y
17:01:31.789 [INFO] Starting ICN node...
17:01:31.790 [INFO] Up and listening on :33333
17:01:31.790 [INFO] Targeting :33333 for discovery
```

Fig 5.2.2 - Screenshot of ./simulations/docker/groups.sh resulting containers running