# Group Diffie-Hellman Key Exchange Implementation

## Authors

- Bernardo de Sousa Valverde
- José Luis Bressan Ruas

## Motivation

Secure group communication is a desirable feature in a sensor network. As an example, imagine a network in which there is a sensor in a light switch, a sensor in a lamp and, far from both of them, a gateway. If an user activates the switch, this command should travel to the gateway, who then should send the lamp a turn on command. In this scenario, there would be a noticeable delay between the flick of the light switch and the actual lamp turning on, since the light switch and the gateway have a key for secure communication, while the lamp and the gateway have a different one. The description of the implementation of the Diffie-Hellman in EPOS can be found in (1). Since all the keys are established between each node and the gateway, no node placed between them can understand the messages they exchange.

A secure group communication would be useful in the described configuration so every node involved in the same operation would have one shared key. All of them would then be able to decipher the message from the switch before it reaches the gateway, deciding to turn the lamp on earlier. This would allow for a much quicker reaction time from the system. If the decision was wrongly made, the gateway could send a turn off message to the lamp within a few seconds, still acceptable for most cases.

## Goals

The goal is to implement a generic n-group party Diffie-Helmann key distribution algorithm, thus allowing all the nodes in a network to understand the messages being passed. The best algorithm to be implemented would be the GDH-3 described in (2), as proved in (3).

## Methodology

We plan on studying the proposed algorithm and fully understanding the way it works. The algorithm will be implemented in C++ and tested with an application before being integrated into EPOS's communication protocol, thus assuring it is working correctly. We will then modify the communication protocol in order to incorporate the updated strategy, and finally test that messages are correctly being encrypted and decrypted.

## Tasks

1. Make project plan;
2. Define and configure testing, simulation and coding environment;
3. Study and understand the algorithm;
4. Implement the algorithm; and
5. Update the communication protocol in order to use the new algorithm.

## Deliverables

1. Project plan;
2. Screenshots of the final environment successfully working;
3. Written description of the algorithm;
4. Code and test results; and
5. Code and test results.

## Task 1

| Task | 27/9 | 4/10 | 11/10 | 18/10 | 25/10 | 1/11 | 8/11 | 15/11 | 22/11 | 29/11 |
|------|------|------|-------|-------|-------|------|------|-------|-------|-------|
| Task 1 | D1 | | | | | | | | | |
| Task 2 | | D2 | | | | | | | | |
| Task 3 | | | X | D3 | | | | | | |
| Task 4 | | | | | X | X | D4 | | | |
| Task 5 | | | | | | | | X | X | D5 |

## Task 2

The final objective of this project is to modify EPOS's TSTP (Trustful Space-Time Protocol) to allow the use of a Group Diffie-Hellman algorithm. In order to simulate and test our implementation, we need an appropriate environment.

A TSTP simulation tutorial for EPOS exists here. This provides us with an environment that not only allows the simulation of various nodes, but is also easy to use and test. Another reason to use this simulator is that the code written for it is really similar to the code written for EPOS.

As seen in the tutorial, the OMNeT++ framework and the Castalia simulator are required. The OMNeT++ provides the infrastructure and tools for writing simulations, while the Castalia is a specific simulator for WSNs (Wireless Sensor Networks). The Castalia version provided in the tutorial already contains EPOS's code and communication protocol.

OMNeT++ provides an IDE similar to Eclipse. In the IDE the code may be modified and a new simulation may be configured in an .ini file. This file will define things like the number of nodes, their coordinates, what application they will execute and when they will start its execution. The IDE may execute the simulation and will generate a tracer file where the results may be seen.

The following images are screenshots of an OMNeT++ IDE simulation configuration and the console output, which is different from the tracer file.

**Create, manage, and run configurations**

Allows running an OMNeT++ simulation

Name: geoSync

Main | Environment | Common

Working directory

/castaliaTest/Simulations/geoSync | Browse...

Simulation

Executable: ○ opp_run ● Other: /castaliaTest/castaliaTest | Browse...

Ini file(s): omnetpp.ini | Browse...

Config name: TSTPNOMF

Run number: | Processes to run in parallel: 1

Options

User interface: ○ Default ● Command line ○ Tcl/Tk ○ Other: 

Record eventlog: ● Default ○ Yes ○ No

Debug on errors: ● Default ○ Yes ○ No

More >>

Apply | Revert

Filter matched 5 of 5 items

type filter text

□ C/C++ Application
▶ Launch Group
▼ OMNeT++ Simulation
    geoSync
    LISHA

Close | Run

---

```
<terminated> geoSync [OMNeT++ Simulation] castaliaTest (04/10/17 12:02 - run #0)

Running simulation...
** Event #1    T=0    Elapsed: 0.000s (0m 00s)  0% completed    ev/sec=0
** Event #17309   T=360.005067999   Elapsed: 0.055s (0m 00s)  100% completed   ev/sec=314691

<!> Simulation time limit reached -- simulation stopped at event #17309, t=360.005067999.


Calling finish() at end of Run #0...
Castalia|       module:SN.node[0].ResourceManager
Castalia|             simple output name:Consumed Energy
Castalia|                   24.4743
Castalia|             simple output name:Estimated network lifetime (days)
Castalia|                   3.18706
Castalia|             simple output name:Remaining Energy
Castalia|                   18695.5
Castalia|       module:SN.node[0].Communication.Radio
Castalia|             simple output name:RX pkt breakdown
Castalia|                   120 Failed with NO interference
Castalia|                   8 Failed with interference
Castalia|                   20 Failed, below sensitivity
Castalia|                   5 Failed, non RX state
Castalia|                   101 Received with NO interference
Castalia|             simple output name:TXed pkts
Castalia|                   94 TX pkts
Castalia|       module:SN.node[1].ResourceManager
Castalia|             cimnlo output namo:Concumod Enorgu
```

# Task 3

The Group Diffie-Hellman algorithm will be implemented using ECC (Elliptic-curve Cryptography) because

of the improved security it offers for a smaller key size, as seen in the following table.

| Symmetric Key Size (bits) | RSA and Diffie–Hellman Key Size (bits) | Elliptic Curve Key Size (bits) |
|---|---|---|
| 80 | 1024 | 160 |
| 112 | 2048 | 224 |
| 128 | 3072 | 256 |
| 192 | 7680 | 384 |
| 256 | 15360 | 521 |

Table 1: NIST Recommended Key Sizes

Some basic understanding on how ECC works can be seen in (4). An example of its use in a peer-to-peer Diffie-Hellman operation can be seen in (5).

The following paragraphs will describe how the GDH-3 algorithm will be implemented using ECC. The elliptic curve to be used will be chosen from (6). The following notation will be used:

| | |
|---|---|
| n | group size |
| $M_i$ | i-th group member; i ∈ [1, n] |
| $k_i$ | random secret number generated by group member $M_i$ |
| G | generator point of the chosen elliptic curve |
| o | order of the chosen elliptic curve |
| $K_1$ | $k_1 G$ |
| $K_i$ | $k_i K_{i-1}$; i ∈ [2, n] |

The first step in a GDH-3 operation with n nodes is for $M_1$ to calculate $K_1$ by using elliptic curve point multiplication. This operation was described in (4) and (5). $K_1$ will then be sent to $M_2$, which will in turn calculate $k_2 K_1$, resulting in $K_2$. This process will repeat itself until $K_{n-1}$ is calculated by $M_{n-1}$. $M_{n-1}$ will not send $K_{n-1}$ directly to $M_n$, but will instead broadcast this value. Thus, all the nodes will know $K_{n-1}$.
The next step is for each node $M_i$ to remove its own $k_i$ from $K_{n-1}$. Since the elliptic curve point multiplication operation is commutative, as shown in (7), we can do this by finding $k_i^{-1}$, where $k_i k_i^{-1}$ mod o = 1, as shown in (8), and then calculate $k_i^{-1} K_{n-1}$. All the calculated values will be sent to $M_n$.
The last step is for $M_n$ to calculate $k_i^{-1} K_n$ for each value received and broadcast it. Each node $M_i$ will then be able to calculate $K_n$ over the received $k_i^{-1} K_n$. Since $M_n$ also received $K_{n-1}$ when $M_{n-1}$ broadcast it, all the nodes will be able to calculate $K_n$, which is the final group key.

## Task 4

Since we wanted to implement and test the key calculation before using it to cipher and decipher messages, all the modifications described here were made over the basic EPOS code.

The already implemented Diffie-Hellman protocol was used as a reference for our work. The first thing done was to remove the Elliptic Curve Point class defined inside the Diffie Hellman class and move it to a file of its own. This made it easier to be used by other classes.
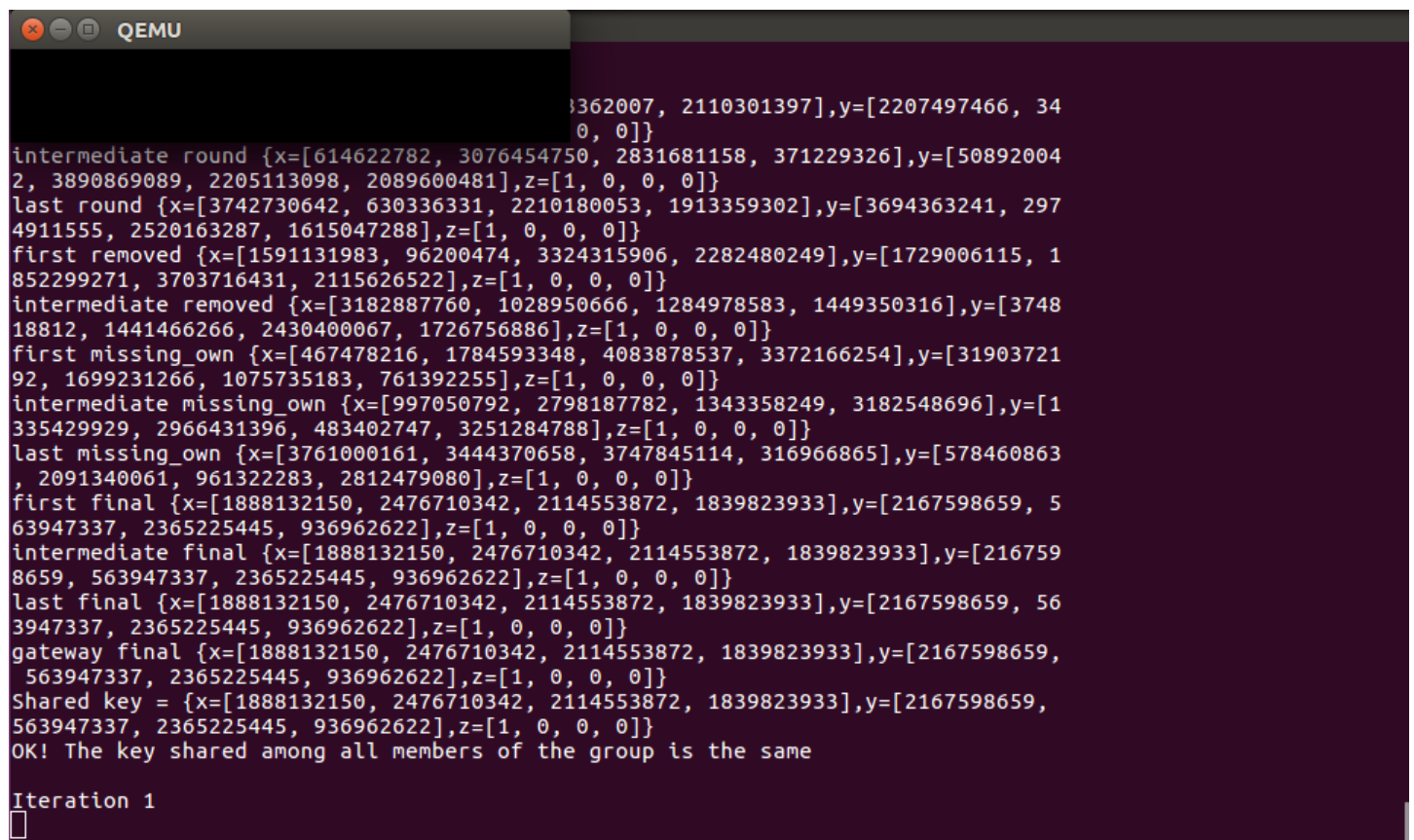
The newly created Group Diffie-Hellman class was implemented with two methods that constitute the bulk of the algorithm: insert key and remove key. The first one adds a node's private key to an Elliptic Curve Point (either the base point or a different one, received via an attribute), and the second removes it. Both of these methods use the already implemented mathematical operations of the Elliptic Curve Point class. In order to remove a key from a point, the private secret's multiplicative inverse is needed. For it to be obtainable, some modifications were made to the Bignum class.

The Bignum class, as defined in (9), represents an integer larger than the normal integer size that exists in a finite field p. Thus, every operation's result is already calculated mod p. The main issue here was that p was defined by the user before compilation and it was impossible to define another Bignum of the same size with a different p. Since we need to find the secret's multiplicative inverse over the elliptic curve's order, and not over p, some changes were made. A new static attribute was added to represent the curve's order and a method was created to change p and the order. Thus, by calling this method before inverting the secret, we can find the correct value. By calling it again after the inversion, the next operations are realized mod the correct p once more.

One final addition was made in the Elliptic Curve Point class. Since the final result of the GDH algorithm is a point, but the secret used in cryptography algorithms is an integer, we added a method called numerize() to transform the point into a number. This is done by calculating xy, x and y being the point's coordinates.

In order to verify that the implementation was working correctly, a test application was created. In it, four nodes participate in the GDH algorithm and, in the end, compare the results. The final version of the modified code, as well as the test class can be found here.

Here is a screenshot of the test result:

# Task 5

In order to be able to simulate the final result using OMNeT++, the base code available at the TSTP Simulation tutorial was modified. The first thing to do was add the modifications made in the last task to the base code. Thus, the files bignum.h, bignum.cc, diffie_hellman.h and diffie_hellman.cc were updated and the files group_diffie_hellman.h, group_diffie_hellman.cc, elliptic_curve_point.h and elliptic_curve_point.cc were created.

Before the communication between nodes may be started in a simulation, they must be initialized. In the initialization of each involved node's TSTP, all of its components execute a bootstrap method. In the TSTP Security's bootstrap, it originally established a key between each node and the gateway using the Diffie Hellman protocol. Now, we verify how many nodes are involved. If there are only 2, they keep using the usual Diffie Hellman protocol. If there are 3 or more, the gateway starts the Group Diffie Hellman protocol, identifying the first node and sending a startup message.

All the messages used in this implementation are control messages. Each kind of message used had a corresponding class implemented in the tstp.h file and an enum added to the control message subtypes defined in tstp_common.h. The already existing control messages were used as reference for the new implementations. The new types of message are GDH_Startup, GDH_Round, GDH_Broadcast and GDH_Response. GDH_Startup works only as a trigger for the protocol to be started, while the others send the partially calculated keys from one node to another. Enums to differentiate the node types and their current state were also created. The types of message, types of node and state of the nodes used in this work are loosely based on (3).

Ways to treat these new kinds of messages had to be implemented. The first modifications were made at the destination(Buffer * buf) method in the TSTP class. This method identifies where the message is headed. The second ones were at the TSTP's update() method, where the received message is simply identified. The final modifications were made at the security component's update() method, where the messages are properly treated.

The first message sent, as was said before, is the GDH_Startup message, sent from the gateway to the first node to trigger the key exchange protocol.
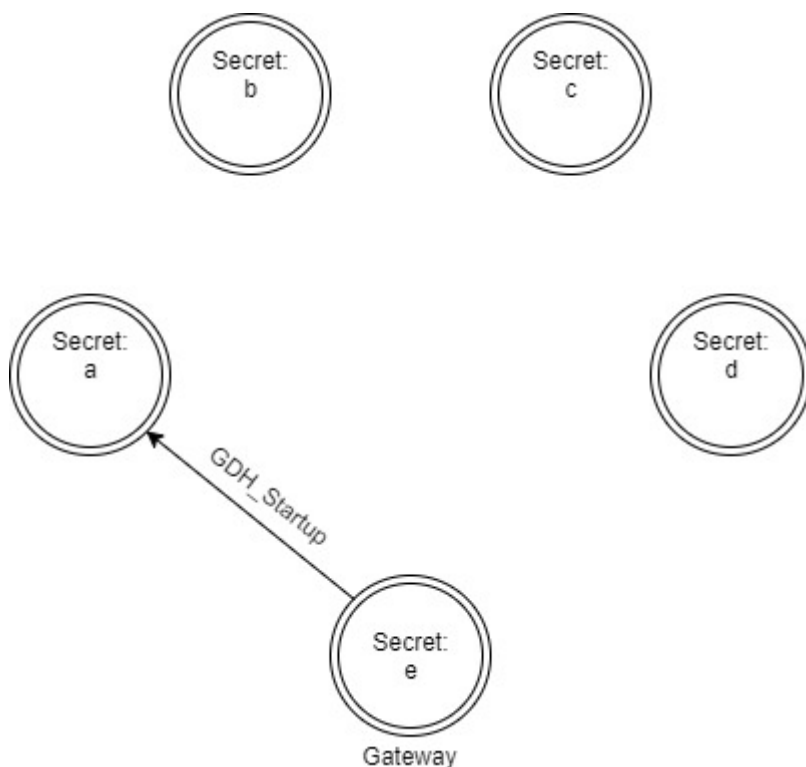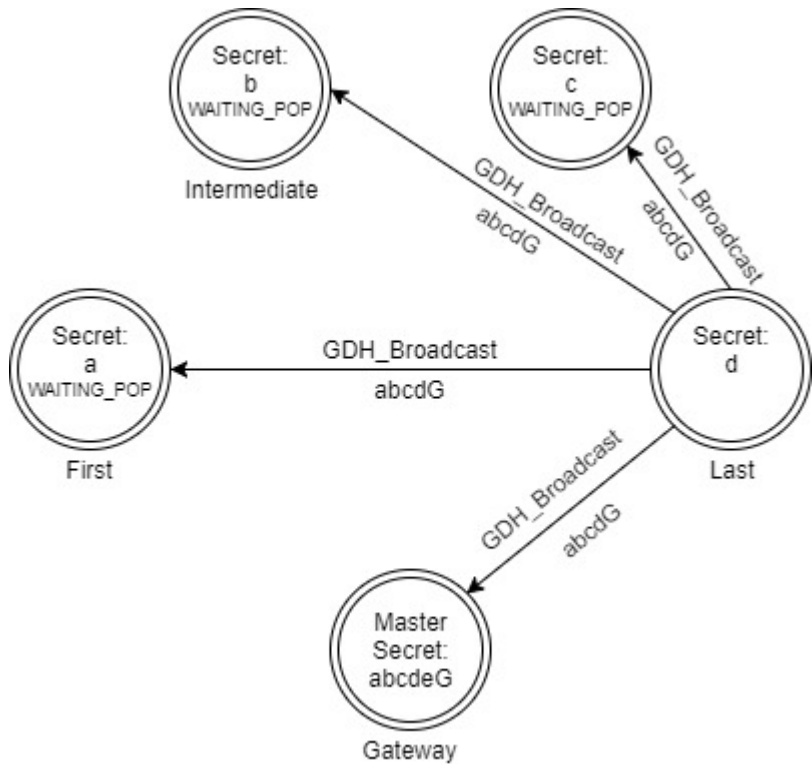


When this message is received, the node adds its own randomized private key to the base point of the chosen elliptic curve. It also sets its type to GDH_FIRST and its state to GDH_WAITING_POP, since it is

now waiting to receive a key from which it has to remove its secret. The next node is identified and a GDH_Round message with the partial key is sent to it. Since the TSTP class has a reference to all the existent nodes, it identifies the next one simply by identifying what the next non-gateway node is.
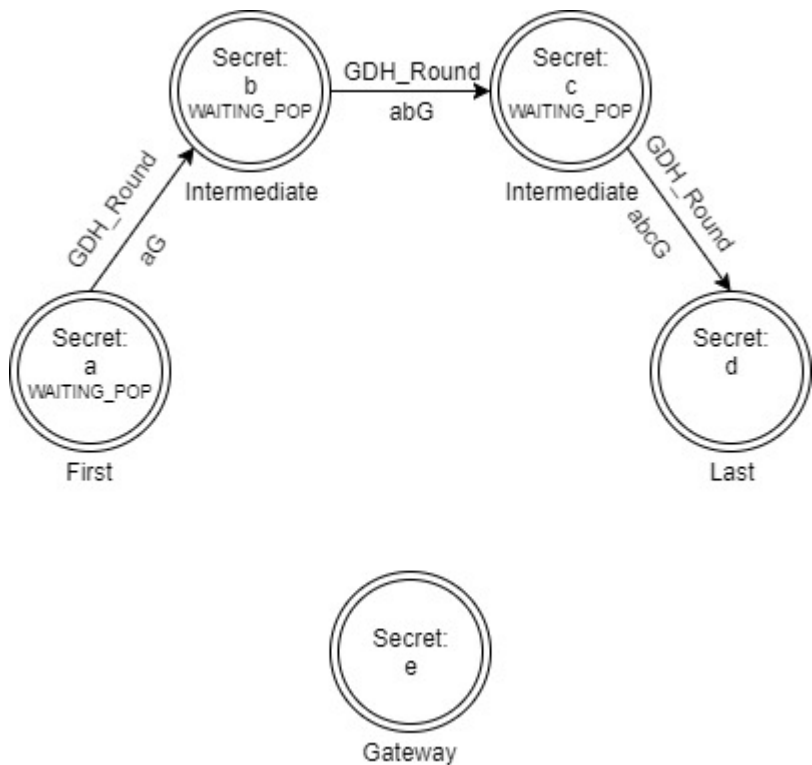


When a GDH_Round message is received, the first thing to be done is to identify if the current node is an intermediate node or the last node. The current node is the last one if it is the last referenced node in the TSTP class, or if the only node identified after it is the gateway. Otherwise, it is an intermediate node. When the type is identified, it is correctly set.

Each intermediate node that receives the GDH_Round message adds its own secret key to the received one and forwards the new value to the next node. The next node is identified in the same way it was identified by the first one. After doing this, the intermediate node's state is also set to GDH_WAITING_POP.

When the last node receives the GDH_Round message, it adds its own secret to the partial secret received. The calculated key will have accumulated all the secrets, except for the gateway's. The key is broadcast to all the existing nodes. This way, the gateway may calculate the final secret and all the other nodes may remove their own key from the partial one.
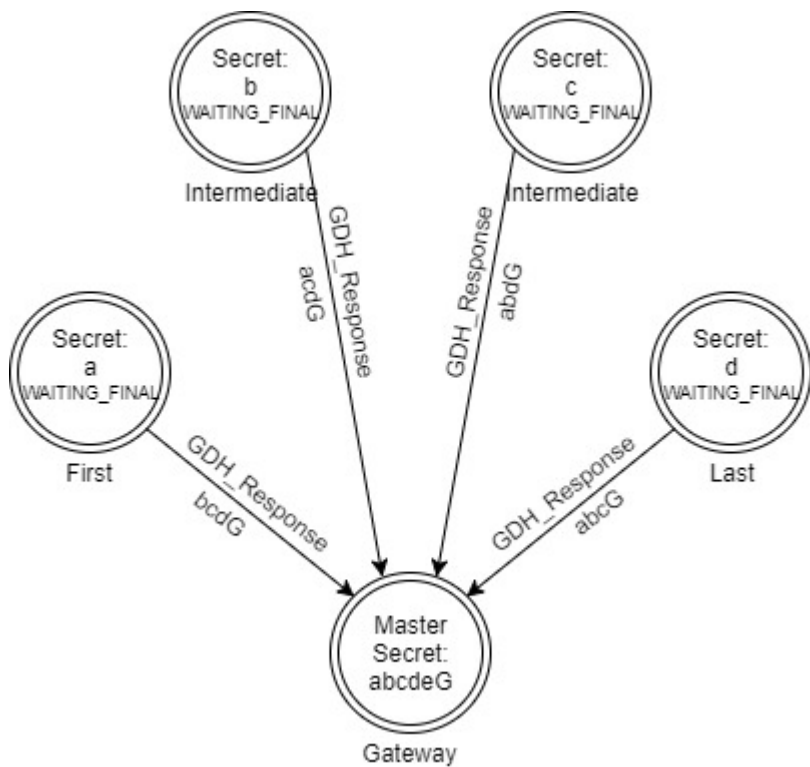


After removing their own secret from the partial key, all the nodes send the newly calculated value to the gateway through a GDH_Response message. Their state is also changed to GDH_WAITING_FINAL, since they're waiting for the final partial key in order to calculate the final secret.

Each GDH_Response message received contains a partial key to which the gateway adds its own secret. A GDH_Broadcast message is sent to the author with the newly calculated value. This way, they will have a partial key that contains all the secrets except for their own. By adding their own secret, they are able to calculate the final key.
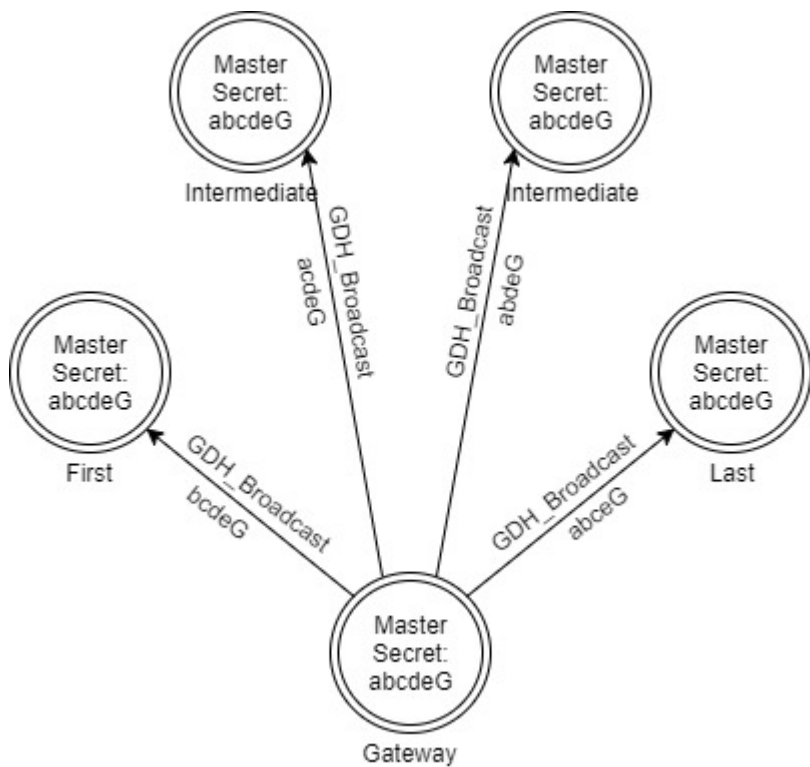
After each node calculates the final key, they have to identify the other ones as trusted peers. This is done by creating a Peer object corresponding to each one of the other nodes and then setting the master secret of the Peer as the calculated secret. Since the master secret is a number and the key is a point, the numerize() method is used. It is really important to note that EPOS doesn't use only the master secret to encrypt and decrypt messages. The peer's id is also used. Thus, in order for all the nodes to be able to successfully encrypt and decrypt messages, they create the peers using the gateway's id. After the peers are created the protocol is complete and messages may be exchanged between every node.

In the simulation, whenever a message is sent an id is created using its coordinates and the current time. Since in the simulated environment time does not pass while a method is executing, two messages created in the same method have the same id and are considered the same, resulting in one of them being discarded. This would make the last node unable to send a GDH_Response message right after sending the GDH_Broadcast. Since the gateway doesn't have to send any messages after it receives the GDH_Broadcast from the last node, it now resends the received message to the last node. It then proceeds in the same way as the other nodes when they receive the GDH_Broadcast message while in the GDH_WAITING_POP state.

The resulting code may be found in https://github.com/bsvalverde/GroupDH/tree/alternate in the Castalia folder. The simulation used was securityTest.ini, and the resulting output may be found here. After t = 30s packets start being sent from the nodes to the gateway. Before it the group key is established, and after that time it is possible to verify that the values sent in the packets are correctly decrypted.

## Bibliography

1. FRÖHLICH, Antônio Augusto. RESNER, Davi. Key Establishment and Trustful Communication for the Internet of Things, In: Proceedings of the 4th International Conference on Sensor Networks (SENSORNETS 2015), pages 197-206. Angers, France. 2015.
2. STEINER, Michael. TSUDIK, Gene. WEIDNER, Michael. Diffie-Hellman key distribution extended to group communication, In: Proceedings of the 3rd ACM conference on Computer and communications security (CCS '96), pages 31-37. New York, USA. 1996.
3. ARAÚJO, André. PFEIFER, Nícolas. SASSE, Evandro. Multy-party Diffie-Hellman Key Exchange. < https://epos.lisha.ufsc.br/Multi-party+Diffie-Hellman+Key+Exchange > as seen in 9/24/17.
4. DEVCENTRAL. Elliptic Curve Cryptography Overview. < https://www.youtube.com/watch?v=dCvB-mhkT0w > as seen in 11/27/2017.
5. PIERCE, Robert. Elliptic Curve Diffie Hellman. < https://www.youtube.com/watch?v=F3zzNa42-tQ > as seen in 11/27/2017.
6. CERTICOM. SEC 2: Recommended Elliptic Curve Domain Parameters. September, 2000. Available at < http://www.secg.org/SEC2-Ver-1.0.pdf >.
7. MATHEW, Ahkil. Elliptic Curves. December, 2008. Available at < https://amathew.files.wordpress.com/2009/12/ellipticcurves.pdf >.
8. CRYPTOGRAPHY STACK EXCHANGE. Inverse problem about scalar multiplication on Elliptic Curve. Available at < https://crypto.stackexchange.com/questions/8925/inverse-problem-about-scalar-multiplication-on-elliptic-curve >.
9. RESNER, Davi. Estabelecimento de chaves e comunicação segura para internet das coisas. 2014. Available at < http://www.lisha.ufsc.br/pub/Resner_BSC_2014.pdf >.