**User Documentation**

**Summary**

[**1.Initialization**](#_heading=h.gjdgxs)

[**2.Assumptions**](#_heading=h.30j0zll)

[**2.Description of the protocols**](#_heading=h.1fob9te)

[**3.Implementation**](#_heading=h.3znysh7)

[**4.Simulation results**](#_heading=h.26in1rg)

# 1.Initialization

Our implementation of the protocols requires at least python 3.10.

These are the following commands needed to run the algorithms implementation on the python environment:

* pip install -r requirements.txt

Also it is required to install docker,rabbitmq from the official sites and running the following in the docker prompt:

* docker start
* docker run -d -p <ip\_address>:5672:5672 -p 15672:15672 rabbitmq:3.8.15-rc.2-management

To run the simulations it is necessary to install the onos controller with:

- sudo docker run -t -d -p 8181:8181 -p 8101:8101 -p 5005:5005 -p 830:830 -- name onos onosproject/onos

We used mininet as a simulation environment so if you want to run the simulation.py script you need to run sudo bash first.

The next step is to run the main\_s.py file to start the server(s) and then run the main.py file with this command line arguments:

* In the case of a non-broadcaster process:

python main.py <message to broadcast> <protocol number> <process type>

* In the case of a broadcaster process:

python main.py <protocol number> <process type>

# 2.Assumptions

# In our implementation we assumed the following: -each process connects to a bootstrap server to gather its unique ID;

-the server(s) address(es) is(are) known to every process in the network

- every process knows the ID of the broadcaster that is settled to one in our implementation;

-the broadcaster sends a message only to processes whose ids have been already received by itself before the broadcast primitive so eventually subsequent receivers that connect to the service after it has broadcasted the message will not receive it;

-the network is fully connected;

-every array that contains messages of a specific type is checked according to an interval of time(e.g every .001 s);

-the message is considered delivered when in the terminal is prompted the "- - - - -Message Delivered - - - - -" string;

-in all the algorithm is used the sha256 cryptographic hash function;

-in the erasure code algorithm has been chosen as an erasure code the shamir secret sharing technique;

-we have modeled a faulty process like a process that discard every message that it receives after the time it fails;

-in the erasure code algorithm, we have considered the "source" field of a message instead of the "from" message field for finality and soundness reasons.

# 3.Description of the protocols

In this section will be stated all the four protocols implemented and considered in the simulations:

* Authenticated Links(AL):

This protocol takes advantage of the underlying authenticated link primitive between each pair of processes to exchange messages for guaranteed authenticity of the processes.The protocol specification is structured in this way:

In The first phase the sender broadcasts a message to all other processes in the network through a SEND message.

In the second phase, when a process receives a SEND message from the broadcaster, it sends to every process an ECHO message. So each process stores these messages to count them.

As soon as a process counts to (N+f)/2 +1 ECHO messages for the same message m, it enters phase three.

In the third phase each process that has received (N+f)/2+1 ECHO messages for a specific message m,broadcasts to all other processes a READY message and as soon as it receives a READY message it stores it. If a process counts to f+1 READY messages for the same message m it re-broadcasts them to all others. If a process counts to 2f+1 READY messages for the same message m and it has not already delivered that message m, it can deliver it.

* Authenticated Messages(AM):

This protocol takes advantage of the digital signature to digitally sign messages sent from a specific node for guaranteed authenticity and non-repudiation. The protocol specification is structured in this way:

1. Propose-Phase. The designated broadcaster 𝐿 with input 𝑣 sends ⟨propose, 𝑣⟩ to all parties.
2. Vote-Phase. When receiving the first proposal ⟨propose, 𝑣⟩ from the broadcaster, send a vote message for 𝑣 to all parties in the form of ⟨vote, 𝑣⟩𝑖 .
3. Commit-Phase. When receiving 𝑛 − 𝑓 signed vote messages for 𝑣, forward these vote messages to all other parties, commit 𝑣 and terminate.

* Hash-Based(HB):

This protocol takes advantage of the cryptographical hashing.In particular, the correctness of the algorithms rely on the collision-resistant property of the hash function used. Cryptographic hash functions are used widely in real-world applications.The protocol specification is structured in this way:

The source node simply sends a MSG message containing its identifier, message content m, and the sequence number h, to all the nodes. Following the convention, we assume that the source also sends the message to itself. Each node may receive five types of messages: ù

-MSG message: this must come directly from the source which contains the message content m. If the source identifier does not match the sender identifier, then the message is discarded.

– ECHO message: this message propagates information about a message already received by some node. ECHO messages contain the full content m. In the hash based algorithms, we only transmit H(m). This is the main reason that we are able to reduce bit complexity.

– ACC message: this message is used to declare to other nodes when a certain node is ready to accept a message m. Again, instead of sending m with the ACC message, we send H(m).

– REQ messages: In our hash based approach, a node might not know the original message m, even after it has observed enough ACC(m) messages supporting it. Therefore, such a node needs to use a REQ(H(m)) message to fetch the original message content from some non faulty node before accepting it.

-FWD messages: When a node is sent a REQ(H(m)) message, it replies with a FWD(m) message that contains the original message content of m.

* Erasure Code Based(EC):

This protocol takes advantage of the erasure coding technique that makes possible to decompose a message into shares and its reconstructions knowing a sufficient number of shares.The protocol specification is structured in this way:

EC will use [n, k] MDS code. To send a message m with sequence number h, it encodes the message m and then disseminates to each peer. The message is a tuple that contains the tag MSG, the source identifier s, corresponding coded element, and sequence number of the message h. To deal with asynchrony and failures, the algorithm is event-driven. First, upon receiving a coded element from the source, node i forwards an ECHO message along with the coded element. Second, upon receiving an ECHO message, node i decodes the message if it has received enough coded elements. The key design behind how crash-tolerant RB achieves the all-or-nothing property is that each peer needs to (pessimistically) help deliver the message to other peers.

# 4.Implementation

In this section we focus on the implementation choices regarding the four algorithms stated in the previous section.

## 4.1.Authenticated links

The architecture of AL is organized in three modules:

-The server module(Server.py)

-The process module(Process.py)

-The authenticated link module(AuthenticatedLink.py)

### 4.1.1.**Server.py**

In the server module we have the bootstrapping functions required

to assign to every process a unique id. The id is pushed in a rabbitmq together with the ip address of the process receiving that id, so every process receives in its queue the list of the active processes as the form <IP,ID>. The id is created following a simple auto increment policy. Every time that a new process connects to the server to gather its id, the server module creates a new serving thread for this connection and delegates to it the management of the new client request. Subsequently another thread pushes the new pair <IP,ID> inside every active process queue.For efficiency concerns all the queues are deleted after an inactivity timeout: realized in the implementation through the functions "delete\_queues","on\_timeout",start\_counter".

### 4.1.2.**Process.py**

In the process module we have the implementation of the real byzantine reliable broadcast primitives. As a first step the function “connection to server” make the process open a socket to connect to the bootstrap server mentioned above to gather ids information and subsequently starts a thread that has the task to check if for a subset of the messages received and stored some delivering or forwarding condition holds.Every process reads the ids of the active processes through a rabbitmq structure using the “pika” library in the “update function”. After that every process stores the ids and the Ips of all the processes in two lists indexed coherently to the order of the processes' connections. The latter step is to call the “creation links” function that is in charge of instantiating the underlying authenticated links module and subsequently to store the references to these modules in a list. After the creation of the links this function calls for every entry of the authenticated links list, the function “receiver” described in the following section”.

After these initialization steps the broadcaster process can call the broadcast function to send a message to every process in the network. Firstly it calls its own “update” function to update its view of the network gathering new network joining processes ids.

Secondly it uses the underlying module to send to every process a message using the “send” authenticated links function.

Each process has a callback function that allows the underlying module to call it after a message for it has been received by the authenticated link module.

The “deliver\_send”,”deliver\_echo”,”deliver\_ready” functions make the process store the messages with the related flags in a list called “currentMSG '' that lists all the received messages. Every time a process receives a message with a specific flag the related dictionary containing for every key(id) a value(message received) is updated if the message is valid. In the listening thread function, the thread that runs this function counts how many values every message flag dictionary contains, for efficiency concerns this parsing is repeated after a sleeping time :"BREAK\_TIME" in the implementation. If the counters satisfy a different condition a specific action is taken according to the algorithm. So the process can deliver the message if it has counted sufficient numbers of ready messages from the other processes.

### 4.1.3.**AuthenticatedLinks.py**

This module has the task to create a logical authenticated channel between every pair of processes that participates in the network.The message is implemented as a dictionary containing:

1. the message
2. the flag
3. the fingerprint

The messages are serialized and loaded according to the json APIs.

The socket addresses are created according to a combination of the ips and ids of the two processes involved in the exchange.

This module takes as input of the constructor:

1. the reference to the upper module process.py;
2. the id of the receiving process namely the pair <IP,ID>;
3. the address of the sending process namely the pair <IP,ID>.

Thanks to these params the authenticated links module can correctly create a socket pair for the calling process.

In the function "receive" the thread that runs on it,instantiated in the "receiver" function called by the upper module,creates the socket for receiving the messages coming from another process and calls the "\_add\_key"to store the new authentication key for this channel. After storing the new key the function calls the "deliver" function that after checking the authentication of the message it calls the related upper module "deliver<FLAG>" function according to the value of the variable "<FLAG>".

When a process wants to send a message it calls the "send" function that after creating the sending socket, creates the fingerprint of the message and the authentication key if it was not already created. The fingerprint is created with the hashlib.sha256 function instead the authentication key for the channel is created with the "generate\_key" function of the cryptography library.

## 4.2.Authenticated Messages

### 4.2.1.**Server.py**

The server module is analogous to the server.py of the previous algorithm description.

### 4.2.2.**Process.py**

The "connection\_to\_server", "creation\_links","update" and the broadcast function are analogous to the previous algorithm. After setting up a vision of the network with the above described functions, a process broadcasts a message using the broadcast function, but before it has to generate its key pair using the "get\_key\_pair" function and it has to get the keys of the other processes participating in the protocol using the "get\_process\_keys" function, connecting t the KDS.

This is the main module of the implementation; it uses two links primitives following the algorithm specification. In the first phase the broadcaster broadcasts to all the "PROPOSE" flagged messages using the authenticated links primitive without signing the message.

In the second phase every process can send a "VOTE" flagged message using the RSA digital signature and using the Link module.

When a process has to send a "VOTE" flagged message it calls the "make\_signature" function signing the pair <message,flag>;the "TYPE" flag here is used to distinguish between the above mentioned phases of the algorithm. When a "VOTE" message arrives from the lower module through the callback "process\_receive" function, each process has to check the message signature using the related stored key in the "check\_signature" function. Every message that is valid is stored in the "vote\_signed\_mess" through the "check" function and the related structure of "counter\_signed\_messages" is incremented accordingly. The module also uses a "checked" dictionary to verify if a pair <message,related signature> has been already checked to increase performances. The "counter\_signed\_messages" is implemented as a dictionary with as the key attribute the message and as the value the incremental value of the counter; the "vote\_signed\_mess" is implemented as a list of messages(dictionaries). In the final part of the algorithm if a process has to forward its n-f valid "VOTE" messages for a message m it uses a "vote\_messages" dictionary sent using the underlying link module; the "vote\_messages" dictionary contains the list of the valid signed messages for that specific message m.

### 4.2.3**AuthenticatedLinks.py**

The authenticated links module is analogous to the server.py of the previous algorithm description apart from the condition to stop listening to the receiving socket channel where the connection is closed when a message of type "PROPOSE" is received according to the fact that after the first phase of the algorithm when the broadcaster send to all processes a "PROPOSE" message using the authenticated primitive, every process do not reuse the authenticated links to send messages instead every process use the signed messages in combination with the normal perfect point to point links.When a process message is received by the authenticated link module the "process\_receive" function of the above described module is called back.

### 4.2.4.**Link.py**

This module has the logical role of instantiating the lower link\_handler module. It can launch the two threads necessary to manage connections for messages to send or receive from other processes. It also has a "link\_receive" function that has the same role as the above-mentioned protocols: it callbacks the upper process module. It uses two flags to make the threads stop when the protocol ends.

### 4.2.5.**Link\_handler.py**

This module has the trivial function of opening two sockets for receiving/sending messages to the other processes. It also checks if the terminating conditions hold to shutdown the socket pair.

### 4.2.6.**KDS.py**

This module has a similar role of the Server.py module but it distributes the keys necessary to sign the messages and to guarantee authenticity and non-repudial.

This module simply act as an always-on key distribution server that attends connections on a specific port and parses the messages received from the process participating in the network. When a process wants to sign a message it has to generate its private/public key pair so it calls the "handle\_read" function of the KDS server. This function parses the message received as a dictionary and scans its type to check if the current process has to generate a new key pair or it is asking for the public key of another process in the network to verify a signature.

## 

## 4.3.Hash Based

### 4.3.1.**Server.py**

The server module is analogous to the server.py of the previous algorithm description.

### 4.3.2.**Process.py**

The "connection\_to\_server", "creation\_links","update" and the broadcast function are analogous to the previous algorithm. After setting up a vision of the network with the above described functions, a process broadcasts a message using the broadcast function; The message is firstly packed as a dictionary containing the flag,the message and the sequence number. Like in the previous algorithm the are separated receiving functions depending on the value of the flag of the message parsed by the underlying authenticated links module. Each function is called by the underlying module according to the flag parsed. In this module the "MSGset" is implemented as a dictionary composed of <tuple(message\_source,message\_sequence\_number);list(actual messages received with that key attributes)>pairs.

The algorithm uses lists containing "echo","acc","req","fwd" flagged messages as structures to verify if a certain message with key attributes has been already received.

The algorithm also uses two dictionaries to store the counters for messages received related by the key attributes fields specified in the algorithm specification. The "check" function is called by the receiving function when certain conditions specified in the algorithm implementation occur. To state if a message with key attributes<message,flag,sender> has been already received the implementation uses a "first '' function that simply verifies if a certain key <message,flag,sender> exists already in one of the above mentioned lists.

### 4.3.3.**AuthenticatedLinks.py**

The authenticated links module is analogous to the server.py of the previous algorithm description apart from the condition to stop listening to the receiving socket channel where the connection is closed when a message of type "ACC" is received according to the algorithm specification.

## 4.5.**Erasure Code Based**

### 4.5.1.**Server.py**

The server module is analogous to the server.py of the previous algorithm description.

### 4.5.2.**Process.py**

As the previous algorithm we have implemented the message as a dictionary containing the relevant fields like: the flag field, the message field,codicing element field and the Hash field described also in the algorithm specification. For this algorithm we have designed these data structures to implement the different functions required to develop the algorithm:

1. The "msgset" containing all the different messages received by a process is a dictionary with as a key attribute a tuple composed by the message source and the message sequence number and as the value: the list of the messages received with those key attributes.
2. The "codeset" contains all the different coding elements received by a sender with a certain source of the message,hash of the message and sequence number of the message received. The structure designed to implement this is a dictionary with as key attributes the above described identification values and as the value the list of the coding elements concerning those key attributes.
3. The counters ,for the different types of messages received, are implemented as a dictionary that contains incremental counters for messages received by a sender with a certain flag of the message,source of the message,hash of the message and sequence number of the message received. The structure designed to implement this is a dictionary with as key attributes the above described identification values and as the value an incremental integer concerning those key attributes.
4. To state if a specific flag-labeled message has been already sent/received by the process we have designed dictionaries to contain this information. Each dictionary is indexed according to the key attributes stated in the algorithm specification.

The "connection\_to\_server", "creation\_links","update" and the broadcast function are analogous to the previous algorithm. After setting up a vision of the network with the above described functions, a process broadcasts a message using the broadcast function; The message is firstly packed as a dictionary containing the flag,the message and the sequence number. Like in the previous algorithm the are separated receiving functions depending on the value of the flag of the message parsed by the underlying authenticated links module. The underlying module takes as reference for the above process module, the "process\_receive" function that in turn redirects the flow to the actual receiving process function to do related actions following the algorithm specification.#TODO the function enc dec with get powerset

The last function is the "check" function and it is implemented following the specification of the algorithm; to check if a message with a certain hash has been already received by a process, we have implemented a flag, namely "there\_is", that is setted to true when for a certain index "j" exists in the "msgset" dictionary a message "j" that has the same hash value of the received message hash previously computed. For listing all the possible subsets of a set we have used the "get\_powerset" function that follows a trivial recursion algorithm.

### 4.5.3.**AuthenticatedLinks.py**

The authenticated links module is analogous to the server.py of the previous algorithm. When a process message is received by the authenticated link module the "process\_receive" function of the above described module is called back.

# 5.Simulation