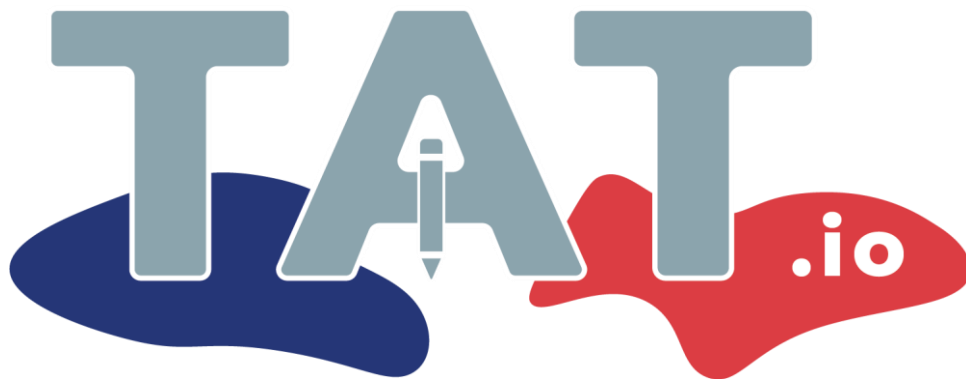


# ELEC-E7320

## Internet Protocol

### **Whiteboarding protocol**



*By Thibaut Queney, Théo Pennerat & Arthur Dumoulin*

## Table of contents

I.	System Architecture .....	3
a.	High-level architecture & its representation.....	3
b.	Protocol stack .....	3
c.	Functional and non-functional requirements .....	5
II.	Design .....	5
a.	Messages definition and format .....	5
i.	Client-side messages .....	6
ii.	Server-side messages .....	7
b.	Sequence charts .....	8
-	<b>CreateMeeting sequence chart</b> .....	8
-	<b>JoinMeeting sequence chart</b> .....	9
-	<b>CreateObject sequence chart</b> .....	10
c.	State diagrams.....	10
d.	A word about ID choice & generation.....	11
III.	Implementation & evaluation .....	12
a.	Technological choices.....	12
b.	Experimental setup .....	13
c.	Security mechanisms.....	14
d.	Network analysis & comparison.....	15
-	<b>LAN environment</b> .....	15
-	<b>Internet environment</b> .....	16
e.	How to launch it .....	17
f.	Additional information .....	18
IV.	Current state of the project.....	18
V.	Possible improvements & future of the application .....	18
VI.	Teamwork.....	19
VII.	Table of figures .....	21
VIII.	Appendix.....	22

## I. System Architecture

In this section, we will first discuss the high-level representation of our architecture. We will then discuss the protocol stack used in order to support our protocol. We will finish by listing the functional and non-functional requirements.

### a. High-level architecture & its representation

The architecture is based on a traditional **client-server architecture**. The idea is that, for each meeting, the server acts as the intermediary for any object modification request from any client. Let's say client A wants to modify an existing object. A request will therefore be sent to the server, and the server will check if client A is authenticated (in a meeting), then look if the object is already in a selected state, and if these requirements are fulfilled, then it will operate the change and send a confirmation, while broadcasting the change to all the other clients. In this situation, we see that no modification of the whiteboard's state can be undertaken without asking the server first and getting its confirmation. In terms of security, this is a weakness in a way: if the server gets compromised, fake change messages can be broadcasted to all the clients in the meeting and therefore a corrupted version of the whiteboard can be created on each client's local copy of it.

The following figure illustrates the mechanism just explained above. The sequence charts shown later in this report will illustrate further how these exchanges take place and how they work.

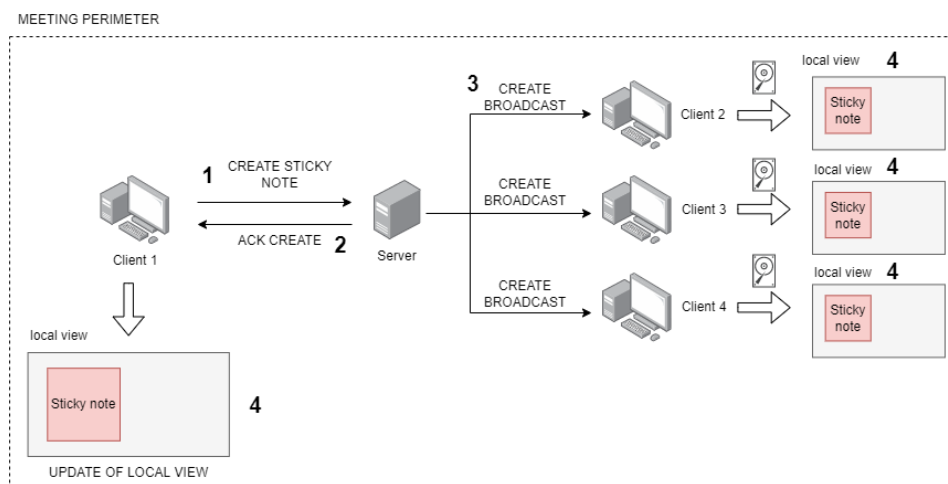


Figure 1: Traditional client/server architecture used in our protocol

### b. Protocol stack

The implementation of the protocol runs on top of a **traditional TCP/IP stack**. **TCP** has been chosen here, even though **UDP** would go faster. The reason for this choice is because we wanted to have reliable messages arriving from the clients to the server, and vice versa. If it was not the case, we would have had to implement some correction mechanisms in the code, which would have taken a lot of time for something that has already been done beautifully with **TCP**. On top of it, we chose **WebSocket** instead traditional sockets to have built-in security through **WebSocket over TLS (WSS)**. With a standard socket, some additional code is required to implement encryption mechanisms (SSL/TLS) but for most of the programming languages used nowadays, **WebSocket** libraries already exist and we just have to use the appropriate functions.

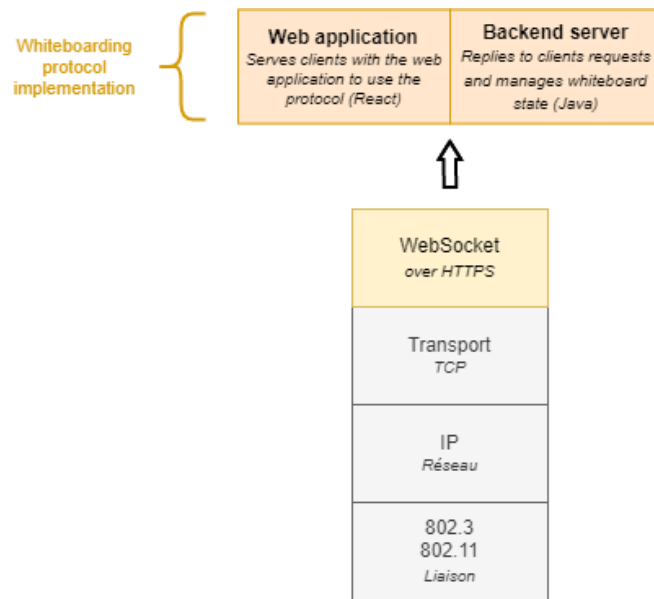


Figure 2: Stack of protocols used for the implementation of our protocol

On the server, two separate functions are provided (as shown on Figure 2):

- **The web application:** this is a React application that is delivered through a regular Apache or Nginx server. The built version of the project generates regular HTML/CSS/JS files that provide the user with a GUI interface to create objects, move them, edit them, upload pictures and implement all the other functionalities. This would be accessible through a regular URL like `https://<server_ip_address>` through HTTPS on port 443.
- **The backend server:** this is another component of the protocol, which is hosted on the same server but runs on a different port (44567). This is accessible via the URL `ws://<server_ip_address>:44567` and clients use it to establish a WebSocket connection in order to send their requests and receives responses.

This behavior is summarized on Figure 3 where a more visual representation is proposed.

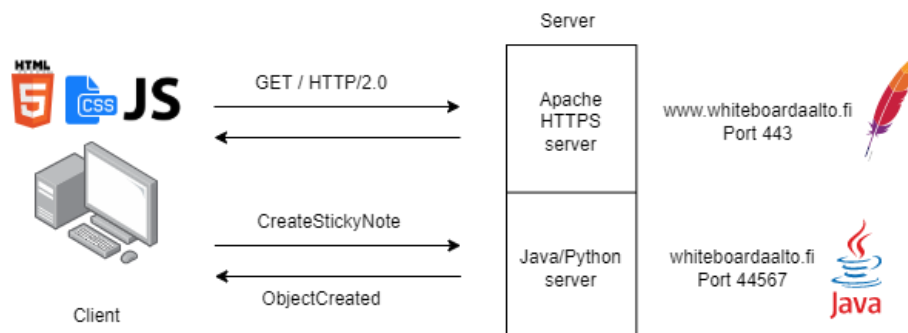


Figure 3: Diagram showing the two components of the protocol implementation running on the server. The DNS names used in this figure are fictional.

### c. Functional and non-functional requirements

For the functional requirements, all our use-cases were designed using the template provided in the first class given about protocol design. They can be found on [this Notion blog](#), and follow the same structure (actors, pre-conditions, steps, errors, post-conditions).

Regarding the non-functional requirements, we identified the need for **low latency** (we operate real-time), **scalability** (we want the protocol to be able to take on more load without impacting the performances in a noticeable way), **modularity** (it must be easy to add new features) and **security** (the protocol should not be vulnerable to any kind of attacks, or at least be secured in the way it is implemented).

## II. Design

In this section, we will discuss the definition and the format of the messages exchanged between the communicating entities to make our protocol work. We will also cover the state-machine diagrams and sequence charts to explicit how these messages are exchanged.

### a. Messages definition and format

In protocol design, two different approaches are usually doable: binary-based messages, or text-based messages. Binary-based messages seemed to be quite hard to design, having no previous experience in the matter. We therefore decided at a very early stage of the project to go for a text-based approach, and we found that the option that would offer us the most flexibility would be JSON messages. In many programming languages, JSON parsing libraries already exist and work very well with object-oriented programming. Each message can be mapped to an object in the programming language used for the protocol implementation, offering a very flexible way of selecting, editing and storing objects.

Parsing an object is quite easy, but there is no way of telling what kind of message it is once received on the client or the server - unless some additional fields are added in the message, or some wrappers are used to tell the distant entity how to decode the message. For these reasons, a **combination of these two methods were used to define the messages** exchanged between the clients and the server. A “superclass” is used as a wrapper, having two attributes: the type of message used, and the message itself (see the **SuperMessage** class on the UML diagram below). In the message itself, if it contains a whiteboard object, or some other kind of “sub-object”, an additional attribute is added to tell the computer what kind of message to expect.

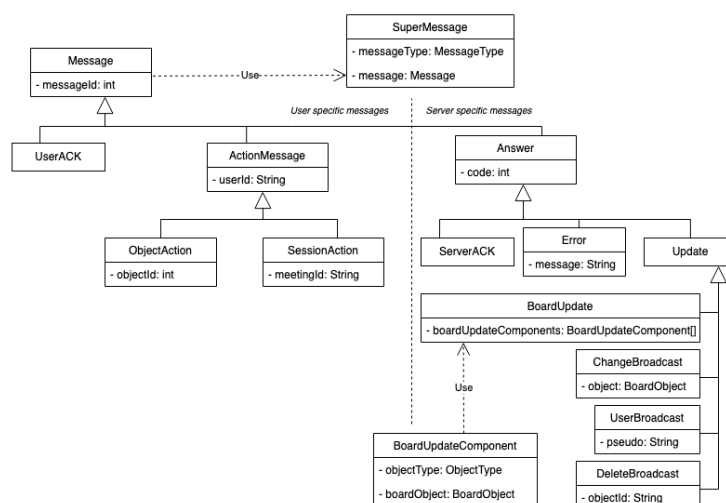


Figure 4: UML representation of the messages

Two main types of messages were defined: **client-side messages** and **server-side messages**. All messages here are represented with their corresponding class (a `LEAVE_MEETING` message received by the server will be mapped, for example, to the `LeaveMeeting` class, and an object of that type will be instantiated on the server). All the messages inherit from a “parent message”: the **Message class**. Whether it is a client or server-side message, they can both be seen - in the object-oriented programming paradigm - as a message of type `Message`. This is helpful to manipulate JSON keys and values, as we don’t need to search for values by looking at each individual JSON fields. Instead, we can use traditional getters and setters, and therefore leverage the built-in functionalities and mechanisms provided by the programming language we use. The same is valid in the other way: replies from the server to the clients are first built creating an object of the appropriate type, and then serialized into a JSON string that will be sent over the **WebSocket**.



*Message examples in JSON can be found by clicking on [this link](#) (Notion blog)*



#### i. Client-side messages

The client-side messages can be divided into 2 categories: **UserACK** messages and **ActionMessage** messages.

- **UserACK:** these messages allow a client to respond to an update coming from the server. For example, if an object has been edited by client A and the change has been approved by the server, a broadcast message containing the change will be sent to all the other clients. These clients have to reply with a **UserACK**, which simply consists of a **messageId** field (an integer) as well as the **SHA256 hash** of the object after the modification. This mechanism is used to detect any variation or deviations from the version that all the clients should have.
- **ActionMessage:** these messages are subdivided into **ObjectAction** (to ask the server to perform an action on an object) and **SessionAction** messages (all the messages to enable the clients to create, join or leave a meeting). They both inherit from the **ActionMessage** class.

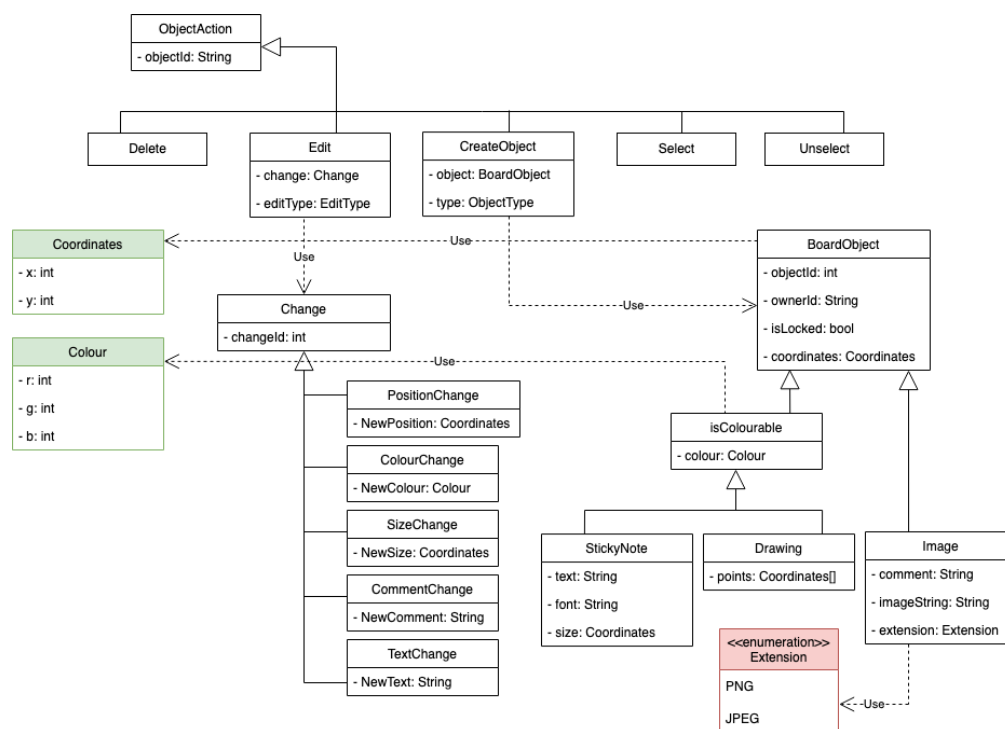


Figure 5: UML diagram representing the `ObjectAction` object

Several kinds of operations have been defined: delete, edit, create, select and unselect an object. They are represented on the top of the UML tree shown on Figure 4, and they reference other types of objects that allow them to perform the action they are required to perform. The **BoardObject** type represents any object on the whiteboard, and the **Change** type represent any change that a user would like to perform on objects. The attributes are all the properties that define the object: if we take the **StickyNote** for example, it has some text (its content), a font and a size (that can also be represented with a **Coordinates** type – x for width, y for height) as well as all the attributes inherited from the parent classes (**isColourable** and **BoardObject**).

The **SessionAction** messages, on the other side, are used to create, join or leave a meeting. Their UML representation can be found in the *Appendix* section of this report.

## ii. Server-side messages

On the server-side, all messages inherit from the **Answer** class. All the answers from the server have a code (an integer) that enables the client to identify quickly what kind of answer it is receiving. The classification shown on Figure 6 illustrates all the different codes that are generated by the server.

The **SuperMessage** wrapper is also a way here of identifying the type of message we are receiving. The error will be contained in the **message** attribute, and the type of message will be indicated by the **messageType** attribute – in the case of a **BusyCoordinatesError**, the **messageType** field would contain “**BUSY\_COORDINATES\_ERROR**” (the capital letters here are used because of the naming conventions used in OOP languages for enumerations, like Java for example).

As shown on *Figure 4* in the UML diagram, an answer can be of 3 different types: **ServerACK**, **Error** and **Update**. The term “answer” here is not to be taken literally: it is not always a reply to a previously received message originating from a client. In the case of an **Update** message, it is generated by the server in order to broadcast to all the other users a change that has been made on an object. Concerning the two other types, a **SeverACK** message is used to ACK a client’s request, whereas an **Error** message is used to refuse a request (for whatever reason) and notify the user of this refusal.

ServerACK for object requests	
101	ColourChanged
102	CommentChanged
103	PositionChanged
104	ObjectCreated
105	ObjectDeleted
106	ObjectSelected
107	ObjectUnselected
108	TextChanged

ServerACK for session requests	
201	MeetingCreated
202	MeetingJoined
203	MeetingLeft

Update codes	
301	ChangeBroadcast
302	DeleteBroadcast
303	UserBroadcast
304	UserLeftBroadcast
305	HostBroadcast

Error codes	
401	AlreadyInMeetingError
402	MeetingAlreadyCreated
403	NonExistingMeetingError
411	UserNotAuthError
412	BusyPseudoError
413	UserNotInMeetingError
421	ServerFullError
422	MessageMalformedError
431	BusyObjectError
432	BusyCoordinatesError
433	ObjectNotFoundError
434	ObjectNotOwnedError
435	ObjectNotSelectedError
436	ChangeNotAllowedError

Figure 6: Server codes for ServerACK, Update and Error messages.

Each of the answer types have their own code range: ServerACK messages for **object requests** go from 101 to 199, **ServerACK** messages for **session requests** from 201 to 299, **Update messages** from 301 to 399 and Error messages from 401 to 499; the full classification can be seen on Figure 6.

### b. Sequence charts

Most of the communications between the client and the server are initiated by the client. The client requests to create a meeting, to join one, to create an object or delete one, etc. Therefore, the server needs to reply to these messages to acknowledge the request. Even though Ethernet tells us that no data was corrupted, and TCP ensures that no data was lost on the way, we still need to get some sort of confirmation from the server that the request was processed properly. Some error could happen on the server side, and therefore an error needs to be returned.

In case of no error, we still made the choice to acknowledge the data by incrementing the **messageId** in the client request by one and returning it in the reply. In case the client request was to create, edit or delete a whiteboard object, a checksum of the new state of the object (or, in case of deletion, the checksum of the object before it was deleted) is generated and put inside the reply as well. This way, any inconsistency - for whatever reason - can be detected by comparing the checksum returned by the server with the one the client is expecting. If the state of the object is not the same as it should be, or if an object is missing, the client can request a transfer of all the currently existing objects from the server, and therefore update its local copy of the whiteboard with the correct version.

In some cases, the server still needs to initiate a communication with the clients. This happens when an object is created, edited or deleted, as all the clients (apart from the one that requested the change) need to be updated regarding the new object's state. This is done through a broadcasting system and defined in a specific class of messages: the **BoardUpdate** class.

In the following sections, we will cover a subset of sequence charts to understand the overall logic of the protocol. However, all the other sequence charts **can be found in the Appendix section of this report**. Also, the orange parts in the charts indicate **what hasn't been implemented in the code yet**. This will be discussed in the *Current state of the project* section.

#### - CreateMeeting sequence chart

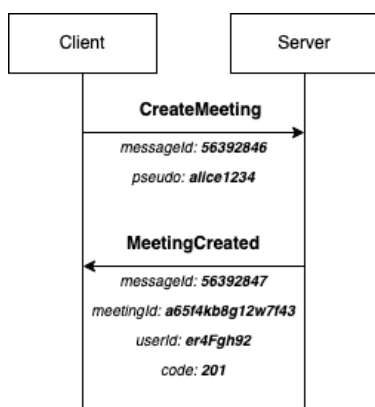


Figure 7: Sequence chart showing the messages exchanged while creating a meeting

The create meeting sequence is one of the two ways of joining a meeting (the other one being joining a meeting by providing a meeting ID).

We can see that the first step is to ask the server to create a meeting (the server will know it is a **CreateMeeting** message because of the **SuperMessage** wrapper that will contain a field **messageType** with the value "CREATE\_MEETING").

To this request, the server replies with a MeetingCreated message by incrementing the **messageId** by 1 and returning the user and meeting IDs to use in the future communications. The last thing to do for the server is to broadcast the new user through a **UserUpdate** message.



## - JoinMeeting sequence chart

This sequence chart is pretty similar to the previous from a logical point of view (client asks, server replies). The only differences are the fact that the number of replies that the server is sending is different, as well as the fields used that slightly differ.

We need here to provide the user that just joined with the current version of the board. Simply sending an ACK with the user ID to use for the next communications would not be enough, as the newly connected client need to see all the objects that have already been created beforehand. The way to do that is to send a second message straight after the first ACK containing all the objects from the board. In order to indicate to the client what kind of objects we're sending, we need to wrap it once more in a parent class that will have an attribute of type **ObjectType** (the **BoardUpdateComponent** class).

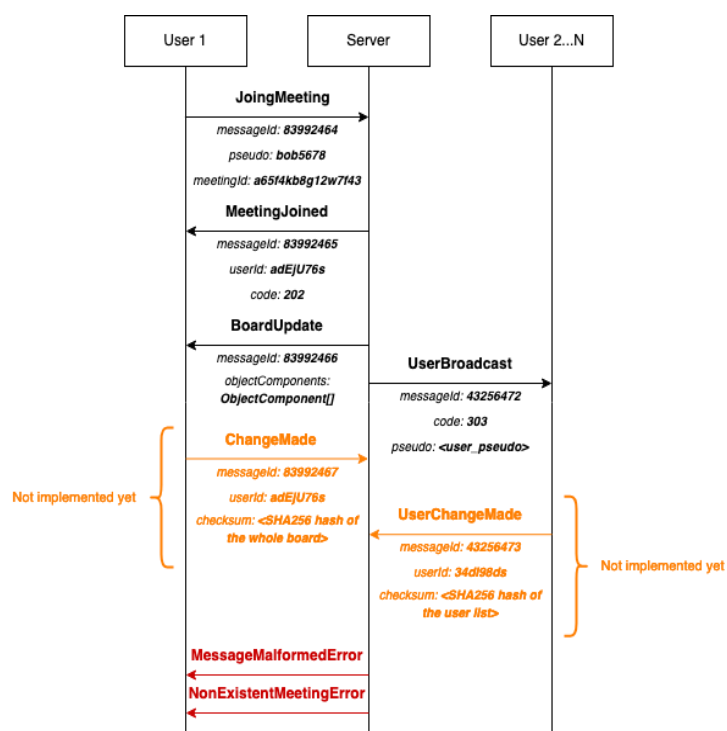
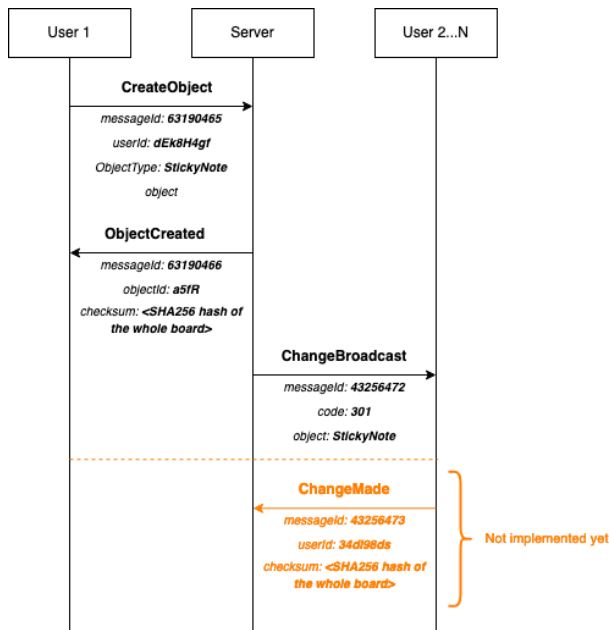


Figure 8: Sequence chart of messages exchanged while a user joins a meeting

The user, like before, replies with his user ID in a **ChangeMade** message and increments the message ID by 1 (taking the last message ID the server transmitted) as well as a checksum of the whole board. If this checksum is not correct, the **BoardUpdate** can take place again. This is a bit of a “redundant” security mechanism in the way that, if the data arrives, it hasn’t been corrupted (the lower layers of the protocol stack offer this service already). However, this can be useful to detect any deviation from the current non-corrupted version that the server holds for later exchanges, or simply to be reused for potential future functional requirements that we could want to add.

The only step left is to broadcast to all the users that are present in the meeting already that a new user just arrived. This is done through a **UserBroadcast** message to which the clients reply by sending a **UserChangeMade** message.

Here, the server can send back an error after the **JoinMeeting** request if the pseudo has already been taken by another user (BUSY\_PSEUDO\_ERROR) or if the meeting doesn’t exist (NON\_EXISTENT\_MEETING\_ERROR).



### - CreateObject sequence chart

To create an object, the user that generates the request must provide the object type, its user ID as well as the object (JSON format). The server can reply with different types of errors (if the coordinates are already taken for example) but if everything goes well, an object ID is sent back as well as the corresponding ACK code and the checksum of the whole whiteboard.

The change then is broadcasted to all the other users. For this purpose, a **ChangeBroadcast** message that contains the board object is sent. The client replies with a **ChangeMade** message (as seen before) to confirm it has received the update.

### c. State diagrams

As the server and the clients operate in two completely different manners, we need to define their states independently. Their actions depend on each other, but they don't perform the same ones. In this report, we will focus on the states in which the client and the server can be once the client has successfully joined a meeting and is interacting with the whiteboard. Other states can be defined – in especially when a client is requesting to join a meeting, or when a client wants to create one – but this represents a very small part of the overall traffic, and these types of exchanges are discussed further in the sequence charts.

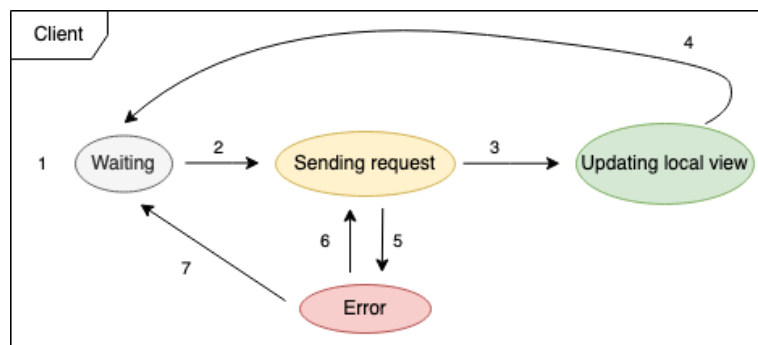


Figure 9: Machine-state diagram showing the state of the client during a meeting

On the figure above, we can see that 4 main states have been identified. We can describe the transitions between these states as followed:

1. The client waits for an action to be performed by the user.
2. Once an action has been performed, the client sends a request to the server with the appropriate action message.
3. If the server ACKs the request, the client updates its local copy of the whiteboard.
4. Once the update has been performed, the client goes back into **waiting state**.

5. After the client sent a request, the server can reply with an error and therefore the client can be in **error state**.
6. If the client is required to perform additional requests to the server to fix the error (if the checksum of an object is not the one expected for example), it goes back into **sending request state**.
7. If the error doesn't require the client to perform any additional actions, it goes back into **waiting state**.

On the server side, we can identify 6 different states. The transitions between these states can be described as followed:

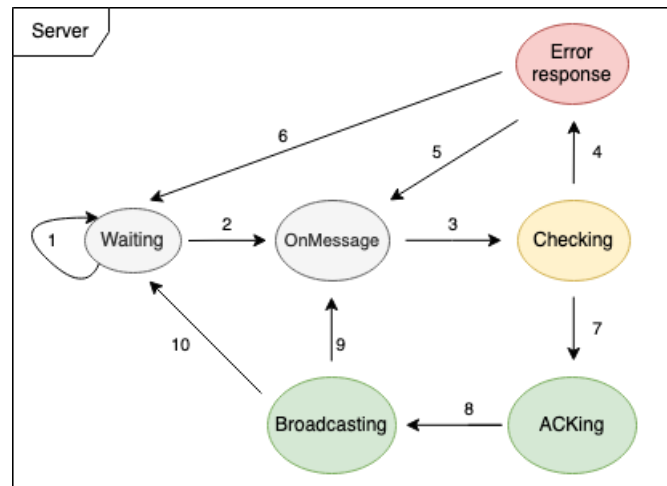


Figure 10: Machine-state diagram showing the states of the server during a meeting

1. The server is waiting for the queue to get a message by one of the clients.
2. The server gets a message and starts processing it.
3. The server checks the request and performs verifications in order to ACK the request or generate an error.
4. If the checking led to an error, the server sends it to the client.
5. If another message is available straight away, the server goes back to the **OnMessage** state where it starts processing the incoming message.
6. If no message is available, the server goes back into **waiting state**.
7. After the checking the request, if it is approved, the server ACKs the request by sending a confirmation to the client.
8. The server broadcasts the change to all the other clients so that they can update their local copy of the whiteboard.
9. Same as step 5, but after having broadcasted a message.
10. Same as step 6, but after having broadcasted a message.

#### d. A word about ID choice & generation

In this protocol, there are 4 different kinds of IDs. They are all alphanumerical strings that are ASCII-encoded, apart from the message ID that needs to be numerical only (integer) so that it can be incremented. We have:

- The **meeting ID**: 16-chars long **string** (128 bits, 16 bytes). It is used to uniquely identify a meeting. For the strings used here, we can use the characters from A to Z, from a to z and from 0 to 9, which gives us 52 different characters for letters and 10 different chars for numbers.

Altogether, it is 62 different characters that we can use to generate this ID, and it is 16-chars long (1 byte per character). The number of possible combinations is 62 to the power 16, which is too long of a number to be written here. This way, we ensure that the meeting ID is unique.

- The **user ID**: 8-chars long **string** (64 bits, 8 bytes). It is used to uniquely identify a user. This offers 62 to the power 8 different combinations, which gives more than  $10^{14}$  different combinations. This is a bit of an overkill for a user ID, as its relevance is only true in the scope of a single meeting, but at least we're sure we have something unique.
- The **object ID**: 4-chars long **string** (32 bits, 4 bytes long). It is used to uniquely identify an object. 62 to the power 4 is 14 776 336, which is almost 18 million different object IDs for a single meeting. This ensures object uniqueness on the scale of a meeting.
- The **message ID**: it is an **integer**. In Java, an integer is 4-bytes long, which is the same as the length of the user ID (can be 1 of 4 294 967 296 different possibilities). The point here is just to generate a random number that will be incremented whether by the server or by the client, depending on who sends the first message. This is definitely enough.

Another ID is also present in the code: a **change ID** (returned by the server after it confirmed a change). For the moment, this is only a random integer that is generated on the flight, just for the sake of filling the field. The idea of the whole project was to be able to implement new functionalities without changing the whole code. Even if there is no use for this field now, it can be used later if we decide to implement versioning of objects: this ID could uniquely identify a change that was made and this way, we could go back to a previous version easier or simply keep tracks of who did what (which is not a requirement at the moment).

### III. Implementation & evaluation

In this section, we will begin by explaining the technological choices that we made regarding the programming languages used to implement the protocol. Second, we will **discuss the experimental setup** used to develop the GUI and the backend server. Third, we will cover the **different security mechanisms** that we implemented in the code. Last, we will talk about some **network analysis** that we performed through packet captures in the different environments that we set up before giving a **small tutorial** on how to launch the project as well as some **additional information** regarding the code.

#### a. Technological choices

The user application (a.k.a. the frontend) was implemented using React. React is a framework that uses TypeScript (an enhanced version of JavaScript) and HTML to generate in the end some HTML/CSS/JavaScript files that can be interpreted by any web browsers. The reason for this choice was to have more flexibility in the user interface design (graphical interface frameworks or libraries, like JavaFX for instance, can be quite hard and tricky to use them without prior experience). The other reason is that the application can be delivered to any entity that understands these formats - basically all the browsers on the market. This increases the portability of the application as it can be delivered anywhere at any time through a web server.

As the user interface is just here to provide some methods to interact with the server and visualize the data exchanged, we did not want to spend a lot of time developing something really advanced. However, we did not want to make it too simple nor impossible to use. For these reasons, we reused the code from the **whiteboard** GitHub repository of **cracker0dks**, removing the components that we would not use and adapting it to our backend logic.

The server application (a.k.a. the backend) was implemented using **Java 17**. The reason for this choice was the powerful built-in data structures and methods proposed by Oracle (such as **HashSets**,

Lists, Maps, and all the methods allowing to manipulate them), as well as some previous experience with that language.

For the backend, the server was coded from scratch. We didn't reuse any previously written code that could have served for some whiteboard application. However, we used third-part libraries such as Jackson (JSON parser) and WebSocket (for the WebSocket communications). Our main class - the **WhiteboardServer** class - extends the **WebSocket** class, which requires to override four functions: **onStart**, **onMessage**, **onError** and **onClose**. These methods are used to handle different kind of situations, depending on whether a new socket is being created, or if a new message just arrived, etc.

### b. Experimental setup

The experimental setup consisted, for the backend, of a very simple Python client that connects to the WebSocket on the loopback interface (127.0.0.1), sends a message and waits for the reply – all of that in an infinite loop. This only takes a few lines of code in Python and enables real-time debugging to help the development of the server. On the same machine, the Java IDE was opened and running the dev-version of the server. It is not really advanced, but it is enough for basic testing and for demo purposes.

For the frontend, the first “functional” version of the backend was distributed and used for development and testing purposes. The next steps consisted of trying out different network configurations. The first other configuration that we tried was from a host to another host within the same LAN. The only additional step here was to get the host IP address through a simple **ifconfig** command. Once more, we had no surprise, and this was working perfectly. The second and last type of configuration that we tried was on a distant server (virtual machine in the cloud) that had a static IP address.

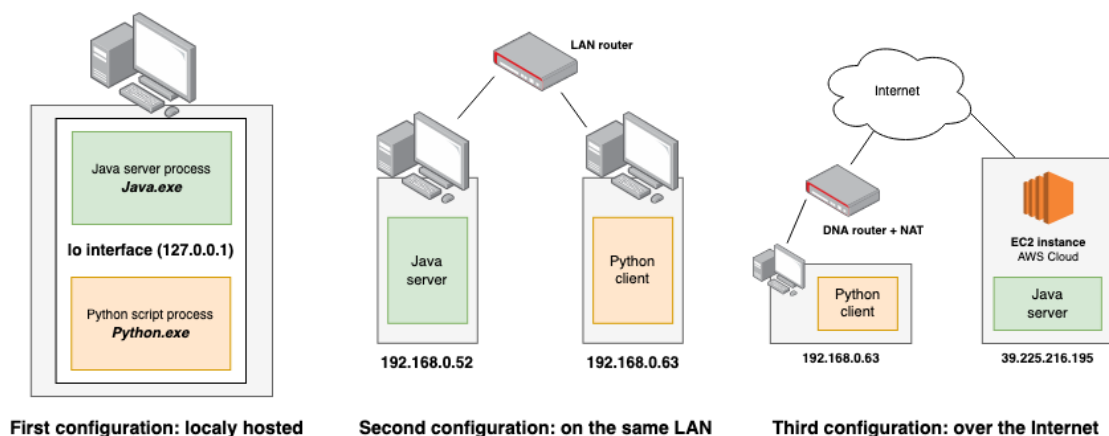
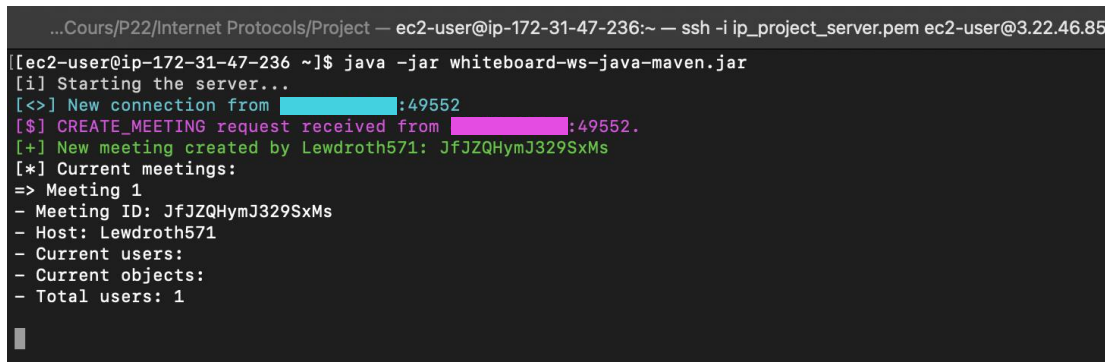


Figure 11: The 3 network configurations that we tried out for our protocol

This configuration is the closest to what we could have if the application were to go on production, as the server is hosted externally (not in the same LAN as the client), which allows anyone to connect. The network performances of these configurations as well as their comparison will be discussed in the next sections. In the following figure, we deployed it on an EC2 instance (further discussed in the *Possible improvements & future of the application* section) in the **eu-east-2 region** (Ohio datacenter). We assigned to it a fixed IP address (an “Elastic IP” in the AWS jargon) and defined a security group that allows port 22 (SSH & SCP) and port 44567 (the Java server). It worked perfectly, even though the

latency was higher than if the server was in Europe. However, for the *Network analysis & comparison* section, we decided to move the instance as close to Finland as possible (Stockholm) in order to have the best testing conditions possible.



```

...Cours/P22/Internet Protocols/Project — ec2-user@ip-172-31-47-236:~ — ssh -i ip_project_server.pem ec2-user@3.22.46.85
[ec2-user@ip-172-31-47-236 ~]$ java -jar whiteboard-ws-java-maven.jar
[i] Starting the server...
[<>] New connection from 3.22.46.85:49552
[$] CREATE_MEETING request received from 3.22.46.85:49552.
[+] New meeting created by Lewdroth571: JfJZQHymJ329SxMs
[*] Current meetings:
=> Meeting 1
- Meeting ID: JfJZQHymJ329SxMs
- Host: Lewdroth571
- Current users:
- Current objects:
- Total users: 1

```

Figure 12: SSH session with the whiteboard Java server. The IP displayed next to the SSH user is the one assigned in the private subnet on the AWS side.

### c. Security mechanisms

When designing a GUI, it is (almost) easy to lead the user to do exactly what we want, by controlling the input and giving access only to the functionalities we want through predefined action buttons for example. However, this is a protocol and like all the protocols, it can be abused. Nothing prevents an attacker from connecting to the server's WebSocket and sending specially crafted messages. If this attacker decides to pass another user ID than its own in its message to the server, it could lead to the deletion of an object for which it wasn't supposed to have any control. For this reason, the first security mechanism that was implanted was a function that tests, each time a message is received, if the user is authenticated. Each time a user joins a meeting – whether by creating or by joining one – its WebSocket object (the **conn** variable in the Java code) is mapped to the User object that represents the actual user. In other words, if the server gets a request with Bob's user ID while using a different port and IP address than the ones that Bob uses to communicate (since the creation of the **WebSocket**), the server will send back an error (**USER\_NOTH\_AUTH\_ERROR**). This is the first step towards preventing uncontrolled actions to take place.

The other security mechanisms could be counted as simple logic; they are more to be seen like a way of preventing errors from happening. Under this category, we can find the following main ones:

- The **verification of the coordinates** (checking if before creating or editing an object, the requested coordinates are indeed free and ready to host a new object);
- The **reelection of a new host** when the current host leaves the meeting (if the host decides to leave the meeting). This way, there is always a host in a given meeting and, later, if we decide to implement a privileges system (a host would have more rights than regular users, like the control of certain parameters, the exclusive ownership of certain objects, or even the power to expulse users from the meeting), the host status has already been defined and can be used.
- The **deletion of a meeting once there is no user anymore**. This prevents the server from creating meeting again and again and wasting RAM's space. If there is still a reference of a meeting in which there are no users anymore in the list of all the meetings hosted by the server, the garbage collector will not delete the meeting and this behavior will waste the server's resources. We therefore need to perform some deletion once all the users have left a meeting.
- The **check performed on a pseudo** when a user wants to join a meeting. If that pseudo already exists, it won't be accepted by the server. As we only display the pseudo in the meeting room,



even though a user is uniquely identified by its ID, two users could appear as the same one if they are given the same pseudo.

All these verifications contribute (as well as others) to a better and safer experience.

#### d. Network analysis & comparison

For network analysis, we used a tool called Wireshark that helped us to visualize the packets that were going through the network. We applied a filter on the WebSocket protocol to focus on the communications between the server and the client. To have conditions as close to reality as possible, we connected 3 clients (other than the host) to the meeting in order to see the broadcasting requests as well. Captures were made on both the client side and the server side in a LAN environment between two hosts, and only on the client side over the Internet (could not run Wireshark on the EC2 instance).

##### - LAN environment

For these tests, we launched four tabs in the same browser and we recorder the packets going through the **eth0** interface (Ethernet). Therefore, as the interface is shared between the four instances of the frontend client, the packet capture is showing the traffic of all of them.

No.	Time	Source	Destination	Protocol	Length	Info
4904	115.082907	192.168.1.100	192.168.1.55	WebSocket	60	WebSocket Ping [FIN]
4905	115.082907	192.168.1.100	192.168.1.55	WebSocket	60	WebSocket Ping [FIN]
4906	115.082907	192.168.1.100	192.168.1.55	WebSocket	60	WebSocket Ping [FIN]
4907	115.083101	192.168.1.55	192.168.1.100	WebSocket	60	WebSocket Pong [FIN] [MASKED]
4908	115.083151	192.168.1.55	192.168.1.100	WebSocket	60	WebSocket Pong [FIN] [MASKED]
4909	115.083181	192.168.1.55	192.168.1.100	WebSocket	60	WebSocket Pong [FIN] [MASKED]
4910	115.083210	192.168.1.55	192.168.1.100	WebSocket	60	WebSocket Pong [FIN] [MASKED]
4946	117.197476	192.168.1.55	192.168.1.100	WebSocket	368	WebSocket Text [FIN] [MASKED]
4948	117.215483	192.168.1.100	192.168.1.55	WebSocket	327	WebSocket Text [FIN]
4949	117.215483	192.168.1.100	192.168.1.55	WebSocket	327	WebSocket Text [FIN]
4950	117.215483	192.168.1.100	192.168.1.55	WebSocket	223	WebSocket Text [FIN]
4951	117.215483	192.168.1.100	192.168.1.55	WebSocket	327	WebSocket Text [FIN]

Frame 4951: 327 bytes on wire (2616 bits), 327 bytes captured (2616 bits)	
Ethernet II, Src: f8:4d:89:84:da:35 (f8:4d:89:84:da:35), Dst: AsixElec_48:20:4d (00:0e:c6:48:20:4d)	
Internet Protocol Version 4, Src: 192.168.1.100, Dst: 192.168.1.55	
Transmission Control Protocol, Src Port: 44567, Dst Port: 54862, Seq: 837516, Ack: 932, Len: 273	
WebSocket	
Line-based text data (1 lines)	

0000	00 0e c6 48 20 4d f8 4d 89 84 da 35 08 00 45 00	...H M M ...S .E
0010	01 39 00 00 40 00 40 06 b5 d3 c0 a8 01 64 c0 a8	9 - @ @ - - - - d -
0020	01 37 ae 17 d6 4e 8d b0 79 5d 38 02 77 2a 50 18	7 - - N - - y]8 w P
0030	13 81 eb f0 00 00 81 7e 01 0d 7b 22 6d 65 73 73	...- - - - { "mess
0040	61 67 65 54 79 70 65 22 3a 22 43 48 41 4e 47 45	ageType" : "CHANGE
0050	5f 42 52 4f 41 44 43 41 53 54 22 2c 22 6d 65 73	_BROADCAST", "mes
0060	73 61 67 65 22 3a 7b 22 6d 65 73 73 61 67 65 49	sage": { "messageI
0070	64 22 3a 33 32 35 39 33 38 35 2c 22 62 6f 61 72	d": 32593 85, "boar
0080	64 4f 62 6a 65 63 74 22 3a 7b 22 6f 62 6a 65 63	dObject" : { "objec
0090	74 49 64 22 3a 22 39 59 70 7a 22 2c 22 6f 77 6e	Id": "9Y p2", "own
00a0	65 72 49 64 22 3a 22 6b 4f 52 71 31 51 68 33 22	erId": "k ORq1q3"
00b0	2c 22 69 73 4c 6f 63 6b 65 64 22 3a 74 72 75 65	, "isLock ed": true
00c0	2c 22 63 6f 6f 72 64 69 6e 61 74 65 73 22 3a 7b	, "coordi nates": {
00d0	22 78 22 3a 37 31 37 2c 22 79 22 3a 35 31 36 7d	"x": 717, "y": 516}
00e0	2c 22 74 65 78 74 22 3a 22 22 2c 22 66 6f 6e 74	, "text": "", "font
00f0	22 3a 22 43 6f 6d 69 63 20 53 61 6e 73 20 4d 53	": "Comic Sans MS
0100	22 2c 22 73 69 7a 65 22 3a 7b 22 78 22 3a 31 2c	, "size" : { "x": 1,
0110	22 79 22 3a 31 7d 2c 22 63 6f 6e 6f 75 72 22 3a	"y": 1}, " colour":
0120	7b 22 72 22 3a 32 35 35 2c 22 67 22 3a 31 32 36	{ "r": 255, "g": 126
0130	2c 22 62 22 3a 31 38 35 7d 7d 2c 22 63 6f 64 65	, "b": 185 }, "code
0140	22 3a 33 30 31 7d 7d	" : 301}}

Figure 13: Screenshot of the Wireshark capture in a LAN environment showing all the exchanges between the clients and the server

The clients are all on the same computer, which means they share the same IP address but have a different client port. A TCP stream is identified by its IP address and its TCP port. If we want to focus on one client, we can simply add in the statistics tool of Wireshark the filter **websocket and (tcp.stream eq 17)**. After selecting the right data, we obtain the following results:

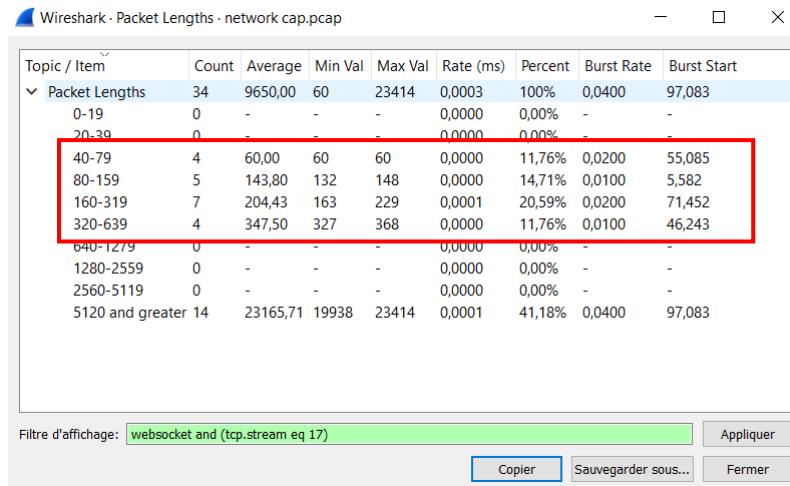


Figure 14: Statistics generated by Wireshark for a test in a LAN environment showing the average packet length

We see that **most of the data (59%) have an average length that is between 40 and 639 bytes**. The other 41% are the other data transmitted by the **WebProtocol** itself to maintain the connection. For example, the longest possible packet used in this protocol is the CREATE\_OBJECT one, and if we capture this packet we see that its length (on the wire, at the physical layer) is 368 bytes, which corresponds to the max value of the 320-639 range.

Regarding the latency, it never exceeds 60ms from the server to the clients. If we take a look at the 4 streams (corresponding to the 4 clients), the maximum latency is always between 50 and 60ms. However, in the other way (from the client to the server), the latency is much higher, going from 125ms to 225ms maximum.

#### - Internet environment

The same test was run from a host in our LAN to the EC2 instance in AWS cloud (hosted in Stockholm). In terms of packet lengths, this won't change as it is the same protocol and the same types of messages exchanged. However, it can be interesting to look at the RTT. From the server from to one of the clients, we have a RTT that just exceeds 50ms.

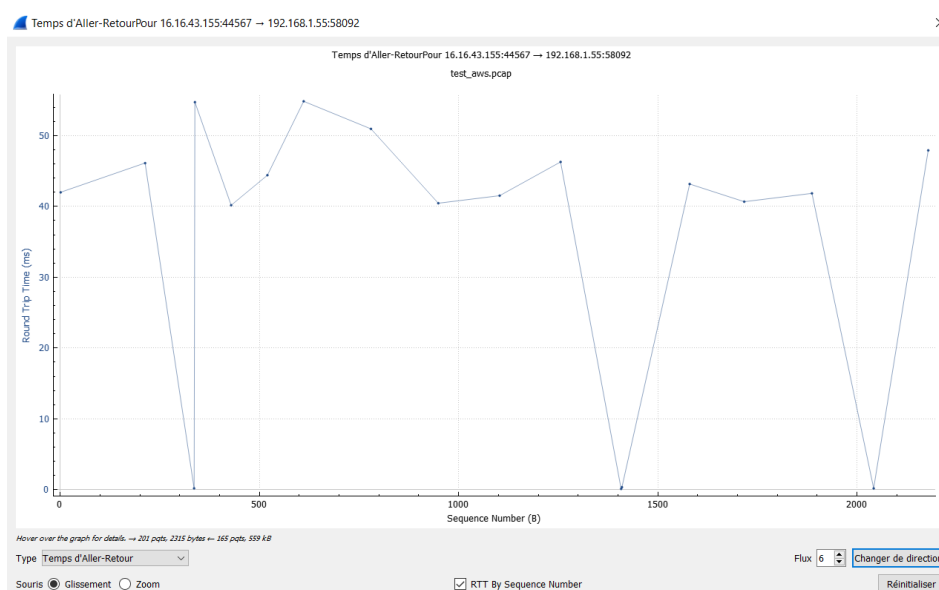


Figure 15: Screenshot showing the RTT from the server to the client in an Internet configuration



The measures of the RTT from the client to the server is much lower than in the LAN: only 90ms max on the following screenshots, and by looking at the other streams, we reach even 25ms max.

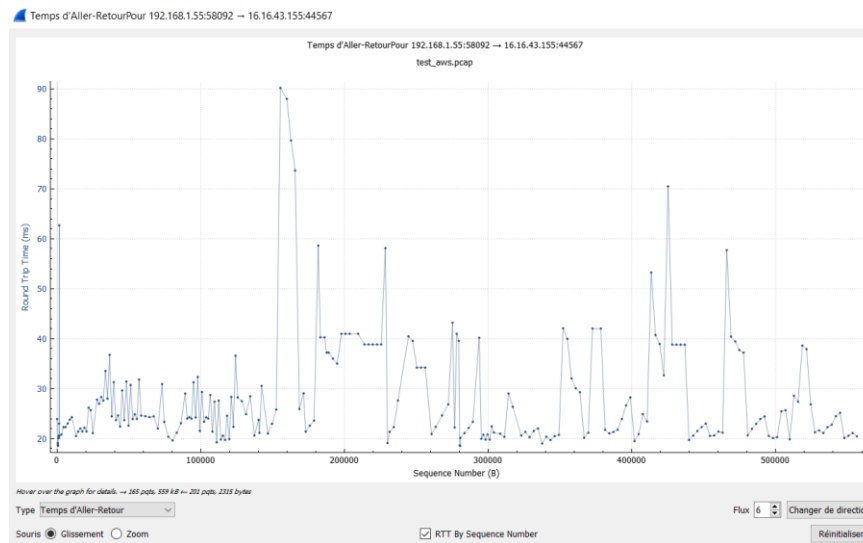


Figure 16: Screenshot showing the RTT from the client to the server in an Internet configuration

Regarding other protocols, the only reference we have is the whiteboard repository of *fwcd* (can be found at [this link](#)) which also uses JSON messages, but the length of the exchanges seems to be smaller. Also, the application uses only Java (both for the server and the client) and communicates over a traditional TCP socket (not a WebSocket). Using the WebSocket enables built-in data encryption, but it adds some HTTPS payloads and additional messages that consume bandwidth and can also add latency. However, for a whiteboard application with only a few users, this should not impact the user experience so much.

#### e. How to launch it

To launch the project, two different approaches can be considered. The first approach consists of using the **React client**. For that, simply go in the root folder of the project and go into the **frontend** folder. Under this directory should be a **dist** folder, in which an **index.html** exists – this page can be opened with any browser, and it allows the user to connect to the server. To launch the server, go to the backend folder and at its root should be a file named **server.jar**. To launch it, simply execute the following command in a terminal:

```
java -jar server.jar
```

It is important to mention here that the server needs to be launched first so that the browser can connect to it in a second time. The second approach is to use the Python client instead. This is to try out hand-crafted messages and to debug individual behaviors of the server. For that, simply go to the frontend folder and execute the client.py by typing the following command in a terminal:

```
python3 client.py
```

The script uses requires the installation of two additional Python libraries: *websocket*, *websocket-client* and *colored* (this one is just to make the responses of the server more colorful). They can be installed with the following commands:

```
pip3 install websocket
pip3 install websocket-client
pip3 install colored
```

Once connected, the command line will ask for a message to send to the server. Message examples in JSON can be found at [this link](#).

**N.B.:** for the Python script, it is important to edit the **create\_connection** function to put the right IP address and port inside (the one used in your own setup). You can also change the port of the server by going into the source code of the Main class in the Java backend folder.

#### f. Additional information

Additional information regarding the backend code structure can be found on the GitHub repository in the backend folder (README.md file). To get a local copy of the code for further analysis, simply issue the following command:

```
git clone https://github.com/TPennerat/whiteboarding\_protocol.git
```

Then, open the **whiteboarding\_protocol** directory in your favorite IDE and navigate to the **backend** folder.

## IV. Current state of the project

Unfortunately, we haven't been able to implement all the functionalities that we designed. The reasons for this are both due to an excess of technicality and to a lack of time. Among these functionalities, there are:

- The **ACK sent by the client to the server after it has sent updates**. This would require implementing some sort of counting mechanism that waits for all the users to reply with a **ChangeMade** message (ACK from a client). Unfortunately, we didn't have the time to finish this feature on time.
- The **freehand drawing**. The drawing class exist and is implemented on the server. The way it is managed is by storing a list of points (**Coordinates** type) in the **Drawing class** as an attribute. From a server point of view, it's not that complicated and it actually works. The problem was on the GUI in order to link all the points together for forms that are not linear (like handwriting for example). The underlying mechanism of storing the points is present, but not its representation.
- The **encrypted version of the WebSocket (Secure WebSocker, or WSS) is not in use**. At the beginning of the coding process, communicating over HTTP was easier in order to analyse the packets exchanged with Wireshark. Also, it was easier to develop a small simple Python client that was simply sending and receiving the JSON strings that we input. We made the mistake of thinking that it was just a matter of changing the URL from ws://something to wss://something but it is not the case. This would require to re-write a good bit of code and to use some encryption library to generate TLS certificates.

## V. Possible improvements & future of the application

The current version of the implementation of our protocol doesn't provide much scalability. If a second server were to be added to host more meetings or more clients, the whole logic of the code would have to be changed. The data is for now stored into the server's RAM, which has two problems:

- If the server reboots for some reasons, all the data is lost.

- It is hard to maintain a consistent version of the data between two servers. It would have to be replicated synchronously, wasting more bandwidth and requiring complex code to simply add more computation power.

A different architecture was envisaged at the beginning, consisting of using two additional components: a queue and a key-value database. Amazon Web Services offer both these options and provide an amazing environment to integrate them together. The idea was to define an endpoint for all the incoming messages, which would be stored in a FIFO queue. Each server would then be polling the messages from the queue, creating and deleting objects from the database on the flight depending on the client's requests. DynamoDB would have been a perfect solution for that, and for the queue, AWS offer a solution called SQS Queue. This would have created a producer-consumer model in which the clients are the producers (the messages are what they produce) and the server are the consumers (they process the messages and send back a reply). Two main advantages can be gained out of this configuration:

- **Leveraging AWS infrastructure's possibilities**, a metric can be added to the EC2 instances (the servers that would be hosting the backend Java application) to watch their CPU utilisation, or directly on the SQS queue to watch how full the queue gets. Based on this metric, new EC2 instances can be automatically launched through an Auto Scaling Group that orders AWS to dynamically allocate more resources. This way, the application never suffers from resources shortage and the end-user is not impacted.
- Using a **key-value database** is particularly fit to JSON messages. Instead of storing objects directly in the server RAM, we can now decouple the whole architecture, separating the computing matter from the data storing aspect. DynamoDB provides scalable storage capacities and is fault-tolerant: the data can be shared between several instances that replicate the data as soon as any change is made to them. It can also share that storage between different geographical areas, preventing any data loss in case one of Amazon's datacenter goes down. This is an extreme situation, but it is important to mention it to show how much better this kind of configuration would be.

The problem that we encountered with that kind of architecture is that it requires some previous knowledge of how to use the AWS API. Even though it is well documented, it is not that easy to learn it quickly, especially when we don't have any prior knowledge of protocol design. Most of our time was spent designing the protocol and defining the messages that would be exchanged between entities. The second problem is that the client would communicate directly with the SQS queue. However, the server would reply from a different endpoint (different identity from a networking point of view, as the IP address and the port used are not the same). Also, SQS doesn't use any WebSocket to transmit data between the clients and the queue. This would have required to create use two different communication channels: one for sending data to the queue, and one to communicate with the servers.

Some alternatives might exist, but we didn't have the technical skills nor the time to investigate further. However, this is a very interesting topic that we would like to keep on researching for any potential further development of this project.

## VI. Teamwork

The teamwork was organized through a software that is called **Notion**. It was used as a central hub to put what we found (articles, examples, bits of code from other people) as well as our ideas. It was

also used as a meeting planner on which we created a calendar with all the meetings that we assisted to. The frequency of the meetings was pretty variable, depending on if we needed to consult each other or not. On this very same calendar, we put all the deadlines of the different phases (taking the different steps indicated in the subject), which allowed us to have a clear vision at any moment of what needed to be done and when. The tasks for each of these phases were also clearly stated with to-do boxes that we could tick.

For the first phase, which consisted in the use-cases as well as the non-functional requirements definition, we mainly used brainstorm as our working method. This way, we were able to put together all the ideas that we had, and it led us to an efficient identification of all the possible requirements. Then we assigned equally to each of us a certain number of use-cases to write, and once that was done, we reviewed them altogether in order to agree on a common basis on which we started to build.

The definition & design of the messages (UML) exchanged between the client and the server were done by Thibaut, as well as the development of the Java server (backend) and the writing of the present report. The development of the frontend (React client) was shared between Théo (focusing more on the communications with the server) and Arthur (focusing more on the user interface). We also set up a GitHub repository to manage the versioning of our code, which was done by Théo.

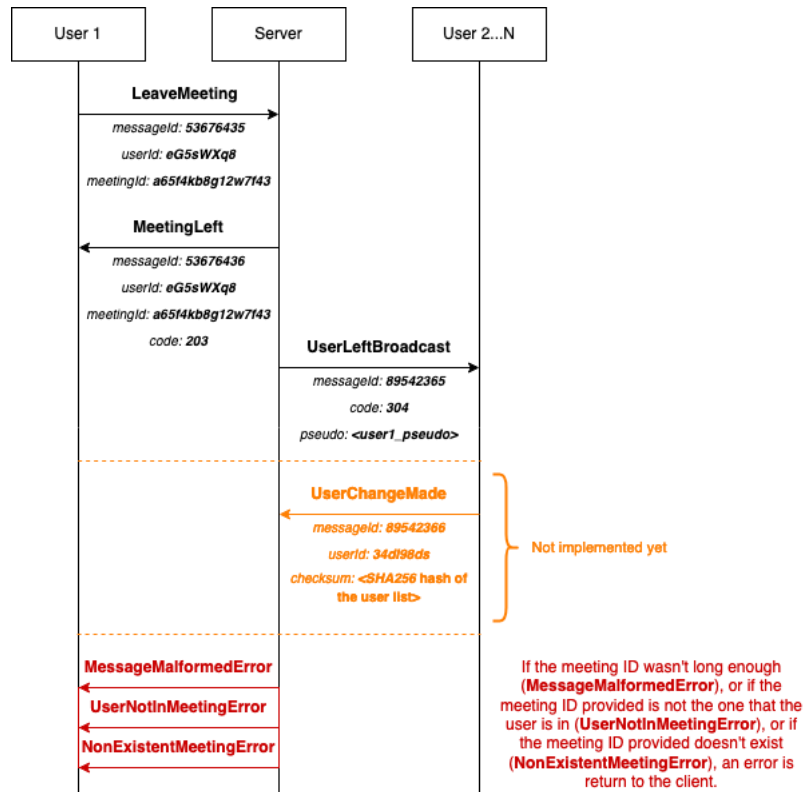


## VII. Table of figures

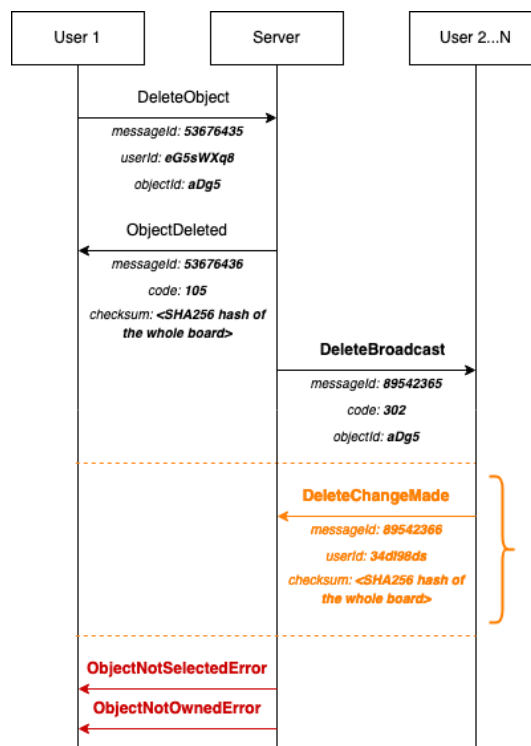
Figure 1: Traditional client/server architecture used in our protocol.....	3
Figure 2: Stack of protocols used for the implementation of our protocol .....	4
Figure 3: Diagram showing the two components of the protocol implementation running on the server. The DNS names used in this figure are fictional.....	4
Figure 4: UML representation of the messages .....	5
Figure 5: UML diagram representing the ObjectAction object .....	6
Figure 6: Server codes for ServerACK, Update and Error messages. ....	7
Figure 7: Sequence chart showing the messages exchanged while creating a meeting .....	8
Figure 8: Sequence chart of messages exchanged while a user joins a meeting.....	9
Figure 9: Machine-state diagram showing the state of the client during a meeting.....	10
Figure 10: Machine-state diagram showing the states of the server during a meeting .....	11
Figure 11: The 3 network configurations that we tried out for our protocol .....	13
Figure 12: SSH session with the whiteboard Java server. The IP displayed next to the SSH user is the one assigned in the private subnet on the AWS side.....	14
Figure 13: Screenshot of the Wireshark capture in a LAN environment showing all the exchanges between the clients and the server.....	15
Figure 14: Statistics generated by Wireshark for a test in a LAN environment showing the average packet length.....	16
Figure 15: Screenshot showing the RTT from the server to the client in an Internet configuration ....	16
Figure 16: Screenshot showing the RTT from the client to the server in an Internet configuration ....	17

## VIII. Appendix

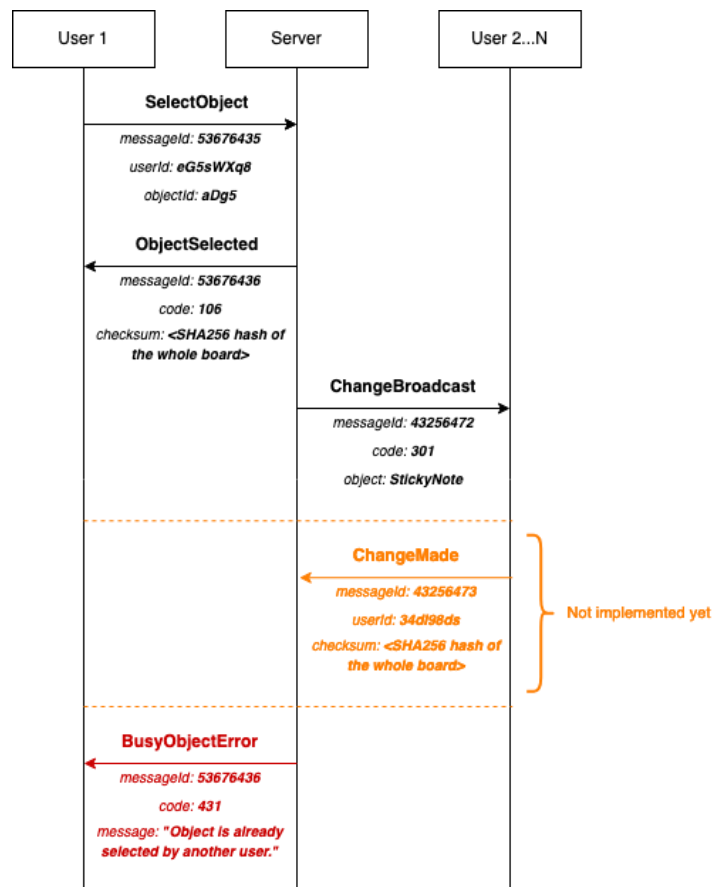
## - Appendix n°1: LeaveMeeting sequence chart



## - Appendix n°2: DeleteObject sequence chart



- Appendix n°5: SelectObject sequence chart



- Appendix n°6: EditObject sequence chart



- **Apendix n°7: UML diagram representing the SessionAction**

