ELEC-E7320

Internet Protocol

**Whiteboarding protocol**

*By Thibaut Queney, Théo Pennerat & Arthur Dumoulin*

Table of contents

[I. System Architecture 3](#_Toc98703937)

[a. High-level architecture & its representation 3](#_Toc98703938)

[b. Protocol stack 3](#_Toc98703939)

[II. Design 5](#_Toc98703940)

[a. Messages definition and format 5](#_Toc98703941)

[i. Client-side messages 6](#_Toc98703942)

[ii. Server-side messages 7](#_Toc98703943)

[b. Sequence charts 7](#_Toc98703944)

[c. State diagrams 8](#_Toc98703945)

[d. A word about ID choice and generation 9](#_Toc98703946)

[III. Implementation & Evaluation 9](#_Toc98703947)

[IV. Possible improvements & future of the application 10](#_Toc98703948)

[V. Team work 11](#_Toc98703949)

[Table of figures 11](#_Toc98703950)

# 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.

## 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 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.

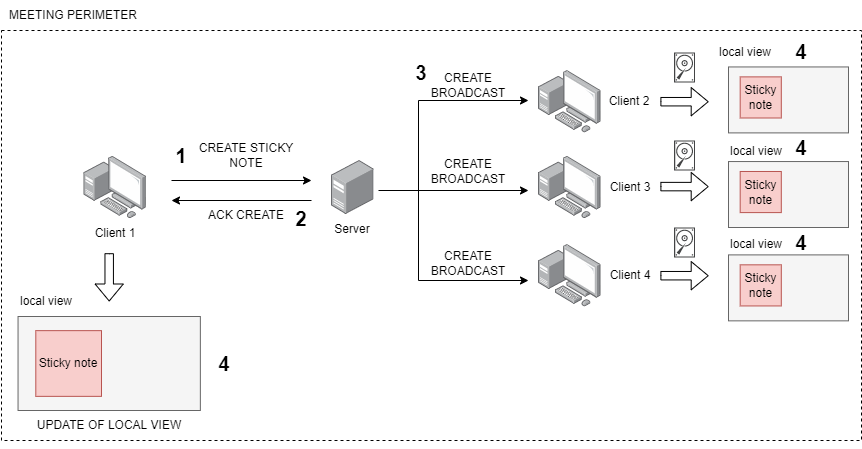


Figure 1: Traditional client/server architecture used in our protocol

## 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.

Une image contenant table

Description générée automatiquement

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://whiteboardaalto.fi* 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 *wss://whiteboardaalto.fi:8888* 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.

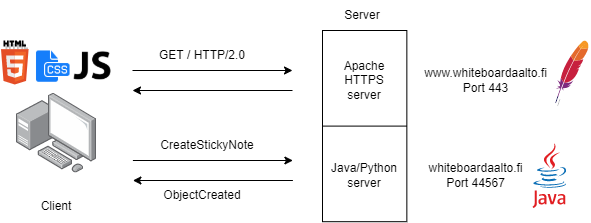


Figure 3: Diagram showing the two components of the protocol implementation running on the server

# 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.

## 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.

Une image contenant texte

Description générée automatiquement

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 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**.

### 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).

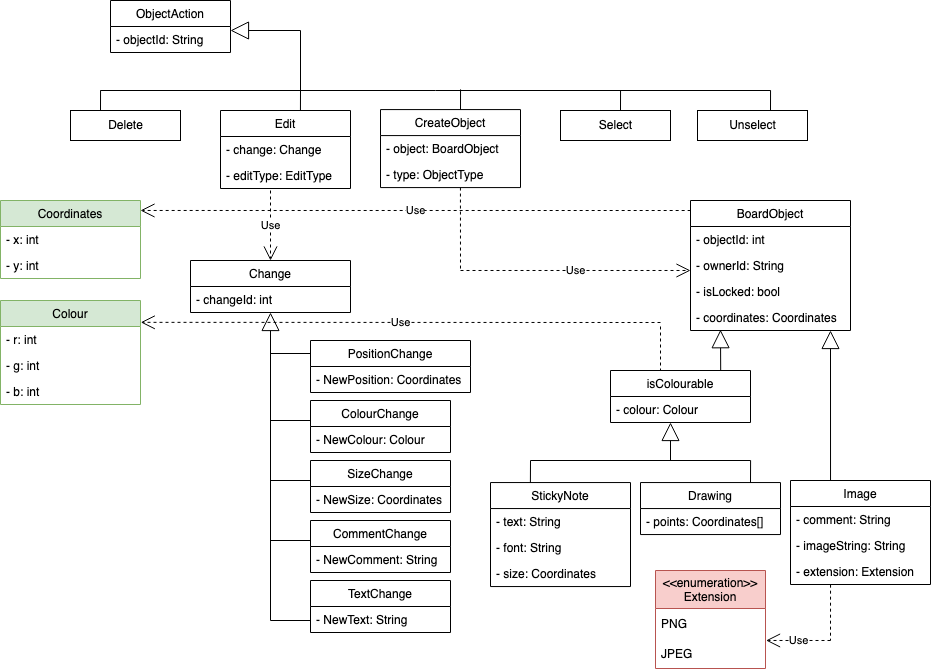


Figure 5: UML diagram representing the ObjectAction object

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, and they reference other types of objects that allow them to perform the action they are required to perform. The **BoardObject** type represent 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**).

### 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 following classification 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 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).

## Sequence charts

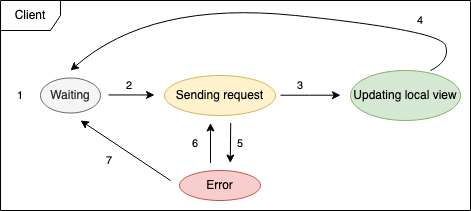
Most of the communications between the client and the server are initiated by the client. The client requests to create a meeting, to join one, to create an object or delete one, etc. Therefore, the server needs to reply to these messages to acknowledge the request. Even though Ethernet tells us that 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. The whole communication process is illustrated on the following figures.

## 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 is identified and in a meeting. 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.



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:

Une image contenant texte, équipement électronique

Description générée automatiquement

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.

## A word about ID choice and generation

# Implementation & Evaluation

* Briefly explain how you implement the app (e.g. SDKs, reusing source code of any existing software)
* Experimental setup
* Test cases (#client, different use cases, different network conditions)
* Result analysis (e.g. latency, data size)

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 or…, 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 though 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 most of the code from the … GiHub repository, removing the components that we would not use and adapting it to our backend logic.

The sever application (a.k.a. the backend) was implemented using Java. 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 wether a new socket is being created, or if a new message just arrived, etc.

# 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 the 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.

# Team work

* Describe how you have worked as a team (e.g. regular meetings, workshops, and etc.)
* State clearly the responsibilities of each team member (e.g. literature survey, programming tasks, network measurement, report writing, etc.)

# Table of figures

[Figure 1: Traditional client/server architecture used in our protocol 3](#_Toc98695840)

[Figure 2: Stack of protocols used for the implementation of our protocol 4](#_Toc98695841)

[Figure 3: Diagram showing the two components of the protocol implementation running on the server 4](#_Toc98695842)

[Figure 4: UML representation of the messages 5](#_Toc98695843)

[Figure 5: UML diagram representing the ObjectAction object 6](#_Toc98695844)