ELEC-E7320

Internet Protocol

**Whiteboarding protocol**

*By Thibaut Queney, Théo Pennerat & Arthur Dumoulin*

Table of contents

[I. System Architecture 3](#_Toc98689629)

[II. Design 5](#_Toc98689630)

[a. Messages definition and format 5](#_Toc98689631)

[b. Sequence charts 6](#_Toc98689632)

[c. State diagrams 6](#_Toc98689633)

[III. Implementation & Evaluation 8](#_Toc98689634)

[IV. Team work 8](#_Toc98689635)

[Table of figures 8](#_Toc98689636)

# 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.

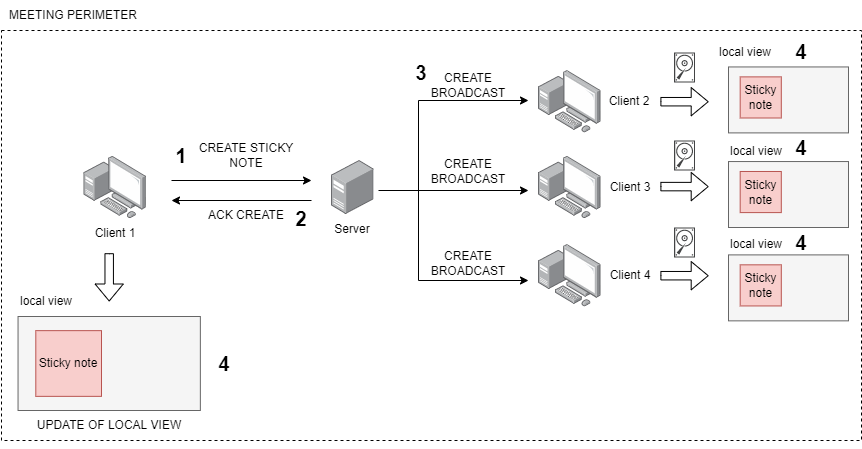


Figure 1: Traditional client/server architecture used in our protocol

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.

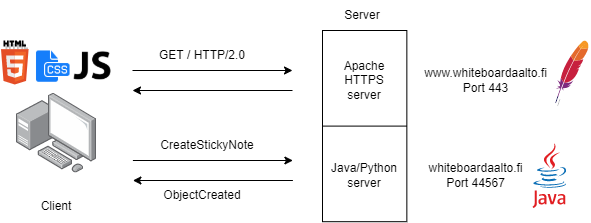


Figure 3: Diagram showing the two components of the protocol implementation running on the server

# 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. 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.

## Sequence charts

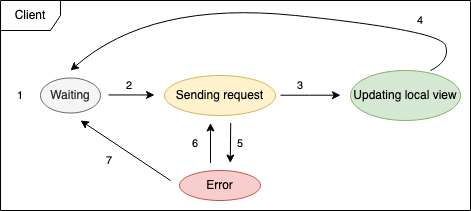
Most of the communications between the client and the server are initiated by the client. The client requests to create a meeting, to join one, to create an object or delete one, etc. Therefore, the server needs to reply to these messages to acknowledge the request. Even though Ethernet tells us that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

# 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)

# 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](#_Toc98687963)

[Figure 2: Stack of protocols used for the implementation of our protocol 4](#_Toc98687964)

[Figure 3: Diagram showing the two components of the protocol implementation running on the server 4](#_Toc98687965)