ELEC-E7320

Internet Protocol

**Whiteboarding protocol**

*By Thibaut Queney, Théo Pennerat & Arthur Dumoulin*

Table des matières

[System Architecture 4](#_Toc98328737)

[Design 4](#_Toc98328738)

[Implementation & Evaluation 4](#_Toc98328739)

[Team work 4](#_Toc98328740)

# System Architecture

* Draw a figure that illustrates the high-level architecture of the system. Add a brief description of the figure.
* List the functional requirements. If the architecture includes several layers of protocol,
* You need to explain the functionality of each layer and the interfaces between adjacent layers.
* List non-functional requirements.

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.

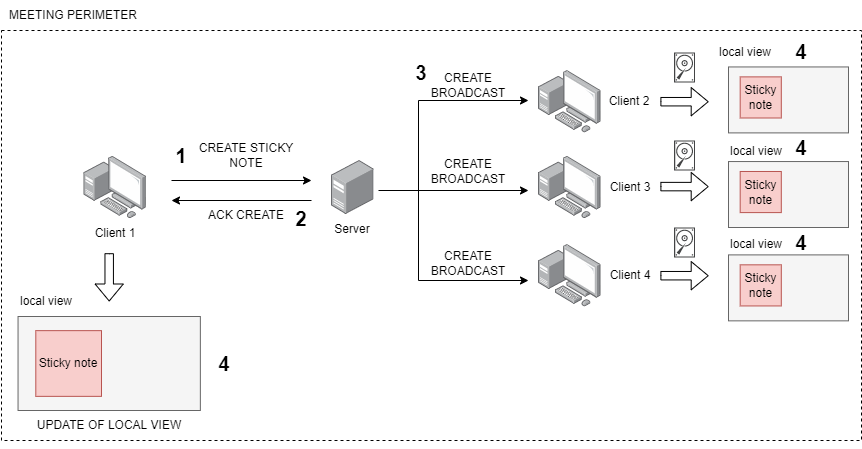


Figure 1: Traditional client/server architecture used in our protocol

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.

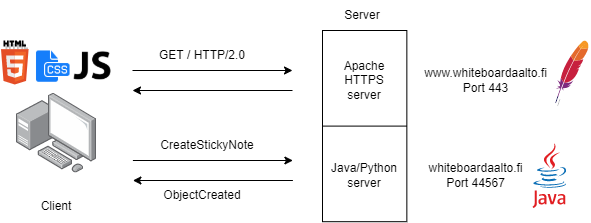


Figure 3: Diagram showing the two components of the protocol implementation running on the server

# Design

* State machines Messages and their formats
* Message sequence charts
* Comparison with previous whiteboarding protocols (focusing on design)

# 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.)