# A Web-Based environment for Cluster-Driven High-Resolution Wall Displays

Juliette Fournis d'Albiat Inria

August 20, 2021

This report corresponds to the work realized during my internship within the ILDA research team at Inria Saclay, under the advisory of Emmanuel Pietriga and Olivier Chapuis.

### General Context

As we use more and more data, we need tools to display it. Large, high-resolution clustered displays are one solution to visualise and interact with big data. During my internship, I tried to design a framework that allows to display a web page on a screen wall and interact with it in a collaborative way. There are already several applications that allow collaboration on a web page, for example Google Hangouts and Skype, but there are also frameworks for large, high-resolution screens based on clusters.

### Problem Studied

The problem I considered was how to display and interact with a web page in a way that allows collaborative working. This problem is interesting because web technologies are very popular and the data circulating on the web is constantly increasing. It therefore seems wise to develop tools adapted to this data. For the moment, few solutions have been explored, as web technologies evolve rapidly and technologies become obsolete quickly.

### Proposed Contributions

I claim that web technologies are suitable for developing a framework that allows any application to be displayed on a high resolution screen wall and also provides collaboration tools. I have developed a framework based on web technologies, which allows developers to quickly display their applications on a screen wall. Then I provide a way to interact with this wall by assigning pointers to remote devices.

### Arguments Supporting Their Validity

In order to test my solution, I designed several use cases of my environment. I deployed them on two different distributed infrastructures of the INRIA laboratory. I found a good user experience during my tests. In order to test the robustness of my solution, I also made more theoretical studies on its performance.

### Summary and Future Work

So far, I have built a prototype rather than a complete environment for collaboration on a distributed infrastructure. The results I have obtained show that it is possible to build a complete environment that offers good user experiences. The first steps to achieve this environment are to develop the interaction tools. I only use remote pointing devices, but there are many ways to interact with a wall of screens. Then, for the moment, my solution is limited to the display of a single web page but we can think of a more sophisticated environment for data manipulation. Future questions to be answered are: What environment could improve the user experience? What tools can be used for interaction?

# 1 Backgound

In this section, I recall the general framework of my internship. First, I define the notion of large, high-resolution cluster-based displays. Then, I describe the opportunities they offer and the main challenges they raise.

## 1.1 Cluster-based Large High-Resolution Displays and Opportunities

High-resolution displays are displays with more pixels than usual, allowing hight pixel density information and the creation of sharp, high-quality images. A display is said to be "cluster-based" if it is controlled by several machines. Large High Resolution Displays (LHRDs) have great potential because they enable the visualization of truly massive datasets.Today, there are many application domains that use huge data. For example, technologies such as LHRDs would be useful for command and control centers, geospatial imagery, scientific visualization, collaborative design and public information displays. Today, web technologies are particularly suited to the use of large-scale data. Firstly, because a lot of data is stored on the web, so the data is available directly and does not have to be imported. Secondly, because the web is so popular, there are many tools that have been created to manipulate and interact with the data. Finally, because of the popularity of the web, there is a large audience of people interested in web technologies and therefore willing to use web-based tools.

Figure 1: An example of an a LHRD. The WILDER ultra-wall running multiple asteroid-generated tsunami simulations.

## 1.2 The main challenges of cluster-based Large High-Resolution Displays

To manage large, high-resolution cluster-based display environments, several issues need to be addressed. Although the display environment is distributed, it must appear homogeneous. The display window is spread over several computers and each computer is in charge of a

specific part of it Thus, the display on each computer must be properly managed. Also the rendering of animations must be correctly synchronised. Each display client must receive the same instructions at the same time and must process them in the same way. Secondly, in most web pages there is only one input source, but LHRD environments aim to allow multi-user input events. In addition, the input tools used to interact with LHRDs are different from the traditional keyboard or mouse that are currently used in basic web page applications. For example, a tablet or remote mobile phone could be used to control a pointer. This requires the use of a more general input specification, with more general properties to cover all possible input sources. Finally, the integration of an application must be efficient enough to not have a significant impact on the performance of the main application. If this is achieved, the user will have a good experience.

# 2 Related Work

## 2.1 SAGE2

### 2.1.1 SAGE

SAGE2 [MAN+14] is an application close to what we are aiming for. It has been designed as a sequel to SAGE, a software that allows to display content on cluster-based LHRDs. Although SAGE product has been adopted by over a hundred international sites, SAGE suffers from the fact that it is increasingly difficult to integrate new features. In addition, the original SAGE was not designed to support collaborative workflows, although it has been extended to do so. Using remote gigabit channels, this extension allows participants to share multimedia content between sites and works well for a variety of problems. Nevertheless, this extension has several major drawbacks, firstly, in order to share and interact with the LHRD, users needed to download and install a client application. But, this requirement is binding and it limits client participation. However, this extension has several major drawbacks. Firstly, in order to share and interact with the LHRD, users must download and install a client application. This requirement is burdensome and limits client participation. Secondly, the content of the shared screen remains under the sole control of the document owner, so the user can only interact with their own content. This results in duplication of work and reduced participation. Thirdly, the lack of integration for working with content made it difficult for scientists to use the shared data.

These drawbacks led the SAGE developers to completely redesign SAGE and to exploit web browser technologies for the implementation of SAGE2. Indeed, web browser technologies have major advantages that address the old problems of SAGE. Firstly, web browsers are platform independent and do not require the intervention of a technical expert to install. Thus, the web browser facilitates the installation of the client application. Secondly, there are many relevant tools that enable communication and collaboration. For example, Web Socket and Web RTC communications allow for real-time communication between application and peer-to-peer file sharing. Browsers can also capture events from input devices such as the touch screen, mouse or keyboard. Finally, web browsers support many APIs for interacting with data, such as high-performance visualisation libraries or APIs for storing and loading data via clouds.

### 2.1.2 SAGE2 architecture

Figure 2 illustrates SAGE2 architecture. It consists of a server, several display clients, an audio client, several interaction clients and an input client. The server is built on top of Node.js [Nod14], which is a platform for creating JavaScript applications. Node.js also provides a package manager for installing applications. This makes installation much easier. Display clients are instances of a web browser that connect to the server by visiting a URL. This URL also contains a unique identifier that identifies each display client and associates them with a specific row and column in the SRSD. Then, each client renders a part of the SRSD and each client display window is defined according to the position of the client on the grid. The SAGE2 server manages the animations by transmitting the pixels of the images via a gigabit data stream at a configurable frequency to all display clients. For synchronization purposes, each client confirms receipt of the frame before the server returns the next frame. To enable multi-user interactions, events from the interaction clients' devices, such as the laptop or mobile phone, are captured and transmitted to the SAGE2 server. The server then

broadcasts these events to each display client.

To extend the types of user interaction, and to enable the use of touch or gyro-mouse devices, SAGE2 uses the Omicron [Omi14] abstraction utility library. Omicron distributes data in a uniform manner that can be interpreted by the SAGE2 application.

### 2.1.3 SAGE2 drawbacks

SAGE2 operates using gigabit channels and updates the client-side display by sending entire frame buffers. These buffers are expected to be large, especially in an LHRD context. Thus, the delay in transferring the buffers can be long and limit the performance of the SAGE2 environment.

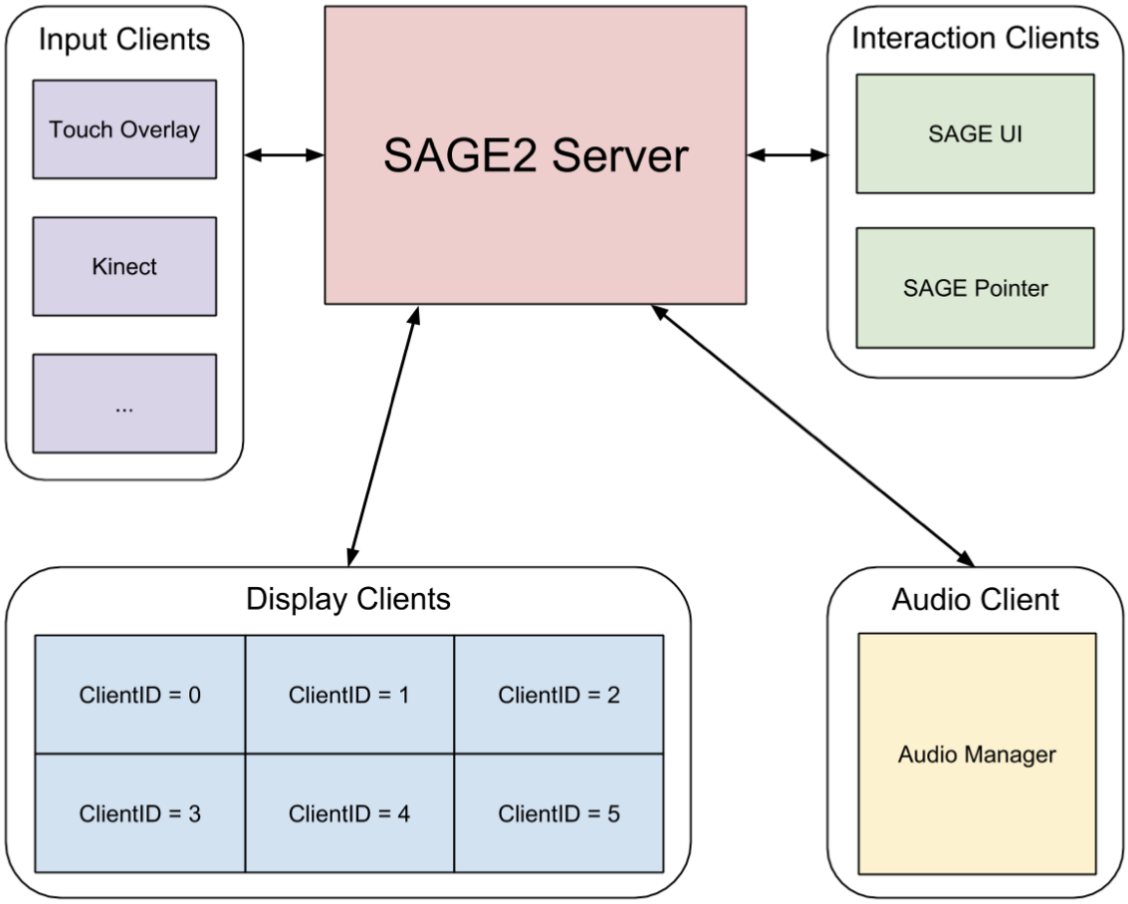


Figure 2: Sage2 infrastructure

## 2.2 Webstrates

The Webstrates system [KEB+15] aims to blur the distinction between documents and applications by enabling real-time collaborative editing of documents. The Webstrates system consists of a server that serves special pages, called Webstrates pages, to an ordinary web browser. Changes

made to Webstrate pages are persistent and are synchronised with all clients that share the page (by connecting to the same URL). A Webstrate page is therefore a collaborative object.

### 2.2.1 Implementation

To synchronise changes between clients, Webstrates uses ShareJS [Sha11] and shares the DOM of the Webstrate page in a MongoDB database. Each operation is then stored in this database. As ShareJS supports plain text and JSON, HTML documents are transformed into JSON files. When a client visits a Webstrate page, the server requests the JSON document and converts it into an HTML page. The Webstrate system uses the DOM MutationObserver API to observe DOM changes, translate the changes into a JSON operation and send them to the database.

### 2.2.2 Webstrates drawbacks

First of all, Webstrates needs to record every change since ShareJS is built on the operational transformation paradigm. Logging changes creates a huge amount of unnecessary data stored by the database for applications that use dynamic animations and involve a lot of changes. Second, although persistence of updates is interesting for collaboration, this difference with the standard behaviour of a web page can be a burden for programmers. Indeed, our goal is to design an environment that allows us to display an application running in our distributed environment as a standard application.

## 2.3 jBricks

jBricks [PHNP11] is a Java framework that integrates a high-quality 2D graphics rendering engine. The framework is essentially composed of two independent modules: one for handling all graphics operations, the other for handling inputs. jBricks adopts a client-server model, with a single instance of the application running on a client node, generating geometry (populating virtual spaces with glyphs) and distributing it to renderers running on the cluster nodes. The virtual spaces and the glyphs they contain are distributed to all nodes in the cluster. They are replicated and kept in sync when glyphs are added, deleted or their properties are changed. I have discover jBricks after the development of my framework. However, jBricks' approach does the same as mine, except that jBrick works on Java objects and I work on HTML objects.

# 3 Animation

## 3.1 Efficiency of the display environment

Unlike an image, a web page is dynamic, it evolves over time. An environment suitable for animations can be defined as an environment in which a large number of images can be executed per unit of time. To display a new image, a browser must perform computational operations to calculate a new image and apply changes to the display. For LHRD, these operations can be more expensive than for a standard display. In a distributed environment, to measure the suitability of a display environment, we also need to consider synchronisation issues. Frames must be updated at the same time and each device must receive the same information.

## 3.2 A model of the overall architecture

In this section, the general model of the display environment is presented.

### 3.2.1 How the Web works

### 3.2.2 Client-server model and dynamic content

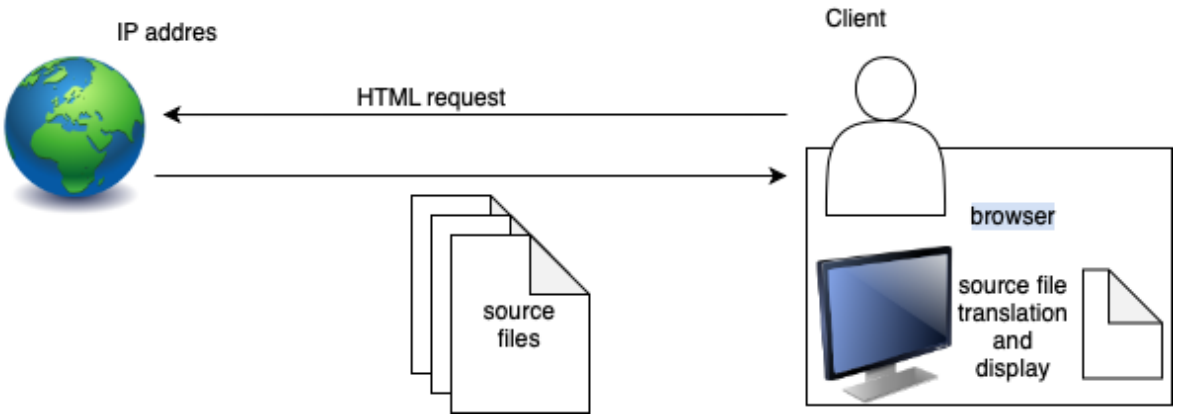


Figure 3: The model client server

I first present the basic knowledge needed to understand the following sections. To understand how distributed display works, it is important to know the overall process that web browsers follow to display a web page. A web client visiting a web page contacts a server. The client is identified by a unique IP address. It sends an HTTP request to obtain the source files of the web page. These source files contain an HTML file, which is the initial content of the web page, a CSS file to style the display and a JavaScript file for the dynamic content of the web page. Then the web browser will be able to translate these files into a single display. Initially, each client will receive the same files and therefore have the same content. But this content will change according to input events such as mouse click or keyboard press. The browser will interpret the input events with the JavaScript file and change the content of the web page.

### 3.2.3 Input Event Management

### 3.2.4 The HTML Document Object Model (DOM)

A web page is a document that can either be displayed in the browser window or as source files. But these representations are the same document. The Document Object Model (DOM) is another representation of a Web page. It is an object-oriented representation. The document is represented as a tree. Each node can be seen as a JavaScript object and has a JavaScript type and methods to be manipulated. The DOM is the representation used for programming.

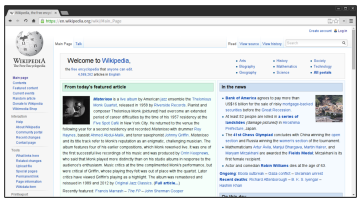

Figure 4: A Webpage displayed by your browser.

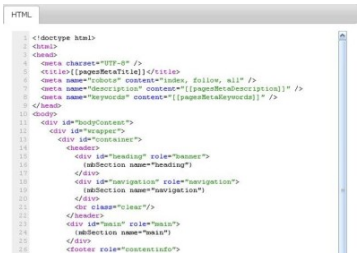

Figure 5: A Webpage as a HTML file.

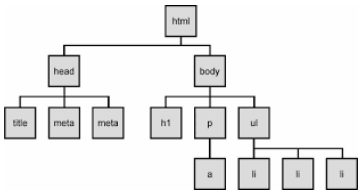


Figure 6: A Webpage, source code(HTML)

## 3.3 A model for distributed architecture

For the distributed architecture, there are several clients that must function as one. They must therefore receive the same content over time. Clients first contact the server and receive the source files of the web page. Then they perform updates to their window so that all client screens appear as one. The remote devices send input events to the server to interact with the screen. There can be very diverse remote devices. Then the display client's web browsers must update their display based on the input events and the dynamic content of the web page. In contrast to the single client model, display clients are different from interaction clients. This is because most of the time, input events do not come from the machine displaying the web page. See figure 7 for a representation of the overall distributed infrastructure.

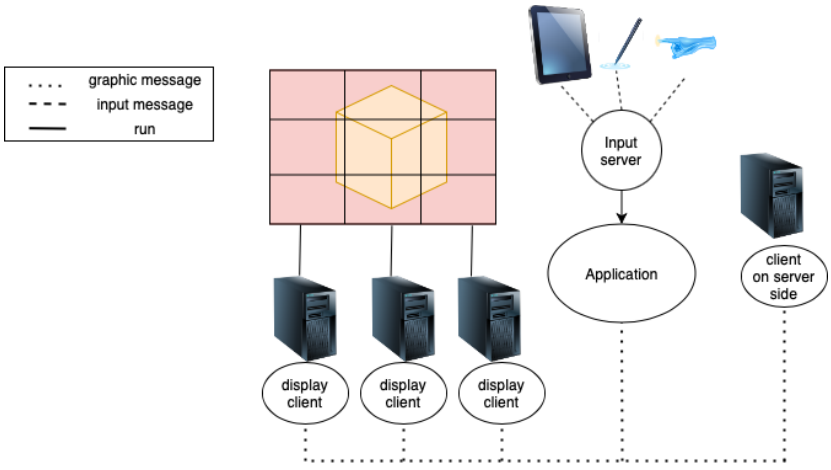


Figure 7: A distributed infrastructure

### 3.4 Viewport management

In order to manage the viewport of each display client, I attached url parameters identifying each client, its position on the layout, it dimension and the size of the complete display. See figure 8 for an example,Then the css file of each display client is dynamically modified according to these arguments.


Figure 8: An url with arguments for managing the display.

### 3.5 Input event handling

For a single web page, input events are interpreted by the web browser on the client side. Thus, a first possibility for handling input events is to send these events to the display clients and let them update their content accordingly. The main disadvantages of this approach are the loss of computational power as the client-side web browsers perform the same operation, but also the fact that the input events may generate non-deterministic changes, for example changing the colour of some text to a random colour. And so this non-determinism will lead to synchronisation problems. A second possibility is to create a server-side client that handles the input events and sends updates to the display clients. The performance of this approach will be limited by the time required to send updates from the server to the display clients. Nevertheless, this approach offers strong synchronisation guarantees as the server-side client always has a valid and up-to-date version of the web page. In addition, the calculations required by the web page content updates are only performed once, on the client side. So I decided to adopt this approach.

### 3.6 Propagation of changes from server side to client side

With changes made on the server side, updates must be passed to the display clients. A classic approach to transmitting information is to send a frame buffer corresponding to the updated display at a constant rate. However, this approach has limitations because, firstly, the display clients will not have access to the content of the web page but only to its representation. The server side should always have a pixel representation of the web page, even if it is not used for display. As we are working with LHRD technologies, the number of pixels to be processed is larger than usual thus frame buffer must take a long time to be transmitted and rendering the entire graphics scene on a single computer would also take a lot of time. So I decided to use the DOM representation of the web page and send the updates that occur in the DOM.

The DOM representation is suitable for propagating changes to the web page because it is a succinct but complete representation of the web page and is designed to be easily manipulated.

### 3.7 DOM structure and listening for DOM changes

As described below, the DOM is a tree made of different types of node.

- Element nodes. These are the inner nodes of the DOM. They are identified by their type which defines the methods and attributes they support.
- Text nodes. The text of the web page is divided into nodes to control the layout of the text.
- Attribute nodes. These nodes allow you to associate properties with elements, such as font colour or presentation size.

With this representation, a Web page is a tree whose leaves are of two types, attribute nodes and text, and whose inner nodes are HTML elements. Since it is not possible to change the type of element nodes, all DOM updates can be defined as follows

- a modification of a leaf.
- a insertion or a deletion of a subtree.

There exists an API that listens to node updates and returns a reference to the node when the update occurs. With this API, we designed the following procedure to propagate DOM changes from the server to the display clients. I present this procedure below. On the server side.

- Adds identifiers to each node of the DOM.
- Send the DOM with identifiers to the display clients. The display clients should send a validation message when they have received all the information.
- When the display clients have responded, enable animation. Run the initialisation scripts and listen for input events.
- Each time a node is modified, if the modification is to an attribute node, send the updated value and node ID to the display clients, if it is a subtree insertion, create the internal node first and assign it an ID, send this ID, the parent ID and the previous sibling ID (if any) and the node type to the display clients, and continue on to the children of the node by recursively applying the procedure. If this is a deletion, send the updated node ID to the display clients.

On the display clients' side.

- Wait for the initial DOM content with the identifiers.
- Recreate the DOM and send a validation message of the initial message from the incoming server.
- When a message has arrived, if it corresponds to a node modification, apply the changes to the DOM.

### 3.8 Problems with concurrent DOM changes

Since we are listening to DOM changes and input events in parallel, we may encounter concurrent conflicts. This is because input events result in DOM modifications, so DOM reads and writes occur in parallel. In order to give a clear idea of the simultaneous conflicts that can occur, I describe a case that I was confronted with during my internship. Suppose

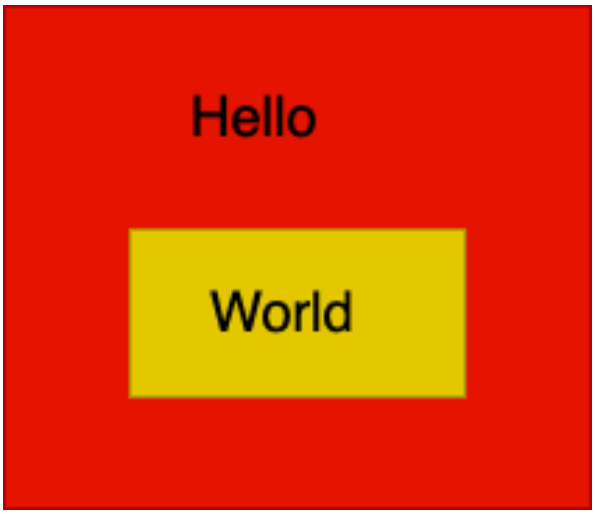


Figure 9: Hello World Display

you want to display Hello World as shown in the figure 9 in your HTML page. Then you need to create node A and node B given in figure 10, and insert node A and then node B. Since you want to forward the changes, you listen for changes in the DOM and get references to the nodes when they are changed. In this example, you would get two references, first a reference to node A and then to node B, which would allow you to recreate these nodes. The problem is that, since node B is inserted into node A, your reference to node A may already contain node B when you try to recreate node A (you will get the subtree corresponding to figure 12 instead of figure 10. Therefore, you will recreate node B twice, when recreating node A and when you are informed that node B has been created.

### 3.9 Managing concurrent DOM modifications

As explained earlier, there are only two possible DOM changes, a leave change or a subtree insertion/deletion. The modification of a leave does not create a conflict since we can apply a modification as many times as we want, the final result will be unchanged. So only insertion and deletion can cause conflicts. Therefore, I decided to insert a new field on the nodes indicating if the node has been inserted/deleted in the DOM. Then I use this field to identify duplicate insertions/deletions and avoid them.

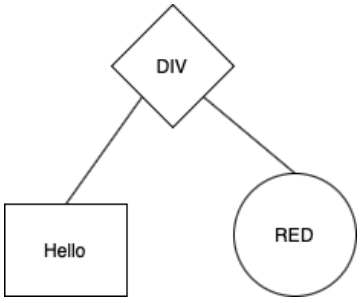


Figure 10: Node A

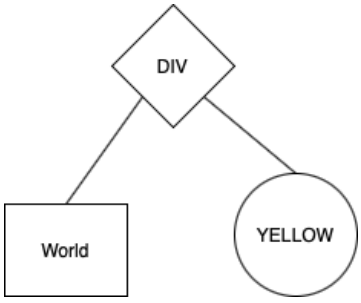


Figure 11: Node B

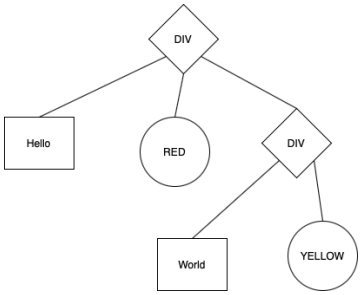


Figure 12: Node A after Node B insertion

# 4 Interaction

To interact with a web page, we usually use the mouse and keyboard of our computer. When faced with a screen wall or other distributed display, we need other interaction devices. Indeed, today it is possible to use touch frame, motion capture devices and remote touch devices. For example, it is possible to use a touch device such as our mobile phone or a tablet or a gyro mouse to control a cursor. We then need to provide a more general type of interaction, broad enough to cover various pointing devices.

## 4.1 Mouse and keyboard fonctionnement

n JavaScript, the mouse and keyboard are represented by event streams that are sent to each DOM element. For example, the mouse is attached to coordinates that indicate its position and generate click events. Thus, in JavaScript, the mouse is an infinite sequence of move, click, up, down and other events. The targets of these events are the DOM nodes located at the same coordinates as the mouse. The determination of these elements is called mouse picking. The programmers can attach event listeners to DOM nodes to create interaction with the mouse. For example, they can add a button that changes colour when it receives a click event.

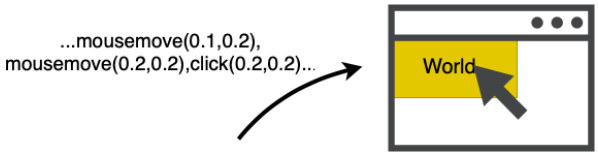


Figure 13: The events generated by the mouse

## 4.2 Enabling remote devices

For distributed displays, remote devices are often used for interacting with the display. In order to use them as input device, we should cast the stream that they generate into a stream of input events for DOM elements. These events are aimed at being used by a programmer thus they should be properly defined. Moreover, we have few information about what does the remote device look like. Thus, our interface should be generic enough to be used a variety of input devices such as a gyro-mouse, a styllus or a mobile-phone.

## 4.3 Pointer interface

We have used the notion of pointer in order to cast input events. Pointer is a generic notion and it covers a lot of kind of interaction. We generate pointer events such as move, up or down. These events also have fields describing the pointer such as coordinates, dimension or source identifier.

## 4.4 Experiences with a touch device

We experienced pointer events definition with a touch device. Our touch device used a TUIO protocol to send information about touches positions. TUIO messages are UDP messages arriving at a constant rate containing a frame sequence ID (an increasing number which is the

same for all packets containing data acquired at the same time) and the list of identifiers of all currently alive tangible objects. These objects, which represent fingers in my experiments, are described by the position in the screen of the touching device and a session ID which is increasing number, incremented at each new touch. In order to cast this sequence into pointer events we cache every touch objects. We decide to allow only one pointer for a device although it can be several touches and to work with relative coordinates. We update coordinates of pointer by calculating position of the center of gravity of the touches available on the touch device. We also use fields such as size or orientation to reflect relative displacement of touches.

### 4.5 Picking issues

Since an event is attached to a target, each time an event is generated, we should be able to determine the DOM nodes which are under the cursor. A restriction of browsers is that they can have access only to the DOM nodes which are currently displayed. Input events are send to the server and it is the server which translates them into pointing events thus the server should determine the targets of these events, nevertheless the server may not display the Web page There are two solutions, displaying the scene on the server side or sending messages to the display clients to determined elements under cursor. This choice depends of the complexity of the scene to display and the network performance. We have decided to implement both possibilities so to let the programmer decide regarding to it infrastructure what is the better choice.

# 5 Experiments

I tested my application on several use cases, I decided to present here one of them, the design of an interactive dashboard about coronavirus figures created with the D3 library [BOH11].

## 5.1 Settings of the Experiments

My experiments were done on two different infrastructures provided by the Digiscope network. I used Wilder which is a cluster of 10 computers, each with an Intel Xeon quad core PC at 3.7GHz and an NVIDIA Quadro K5000 graphics card. Wilder has a resolution of 14,400 x 4,800 pixels, and its dimensions are 5.90m x 2m. I also tested my results on Wild, a cluster of 16 computers, each with an Apple Mac Pro quad core at 3.2 GHz and two NVIDIA 8800GT graphics cards. Wild has a resolution of 20,480 x 6,400 pixels and its dimensions are 5.50m x 1.80m.

## 5.2 Wilder description

During my internship, I mainly worked on Wilder [Dig] which is an ultra-high resolution screen wall. Wilder is the collection of ten screens, see figure 15 for a representation of the

Figure 14: Geolocation and real-time display of rail traffic throughout France on Wilder platform

architecture. Each screen is controlled by a remote computer. It is possible to communicate with this cluster of computers, for instance I usually used an ssh connection to control the display of Wilder.

### 5.2.1 Testing performance of DOM changes propagation

In order to test the performance of the DOM change propagation and to see how well it adapts to large screens such as Wild or Wilder, I created tests such as creating a configurable number of pointers that move in the vertical or orthogonal direction. Then I was able to check that the switch from one screen to another was smooth and that all movements were done correctly. I observed that the number of pixels becomes too large to be processed properly by the screen before reaching a bottleneck due to the flow of information from the server to the clients. Indeed the browser display is refreshed typically 60 times per second. If a change to a DOM node can be sent with a 500 byte buffer (this is the order of magnitude of the packet

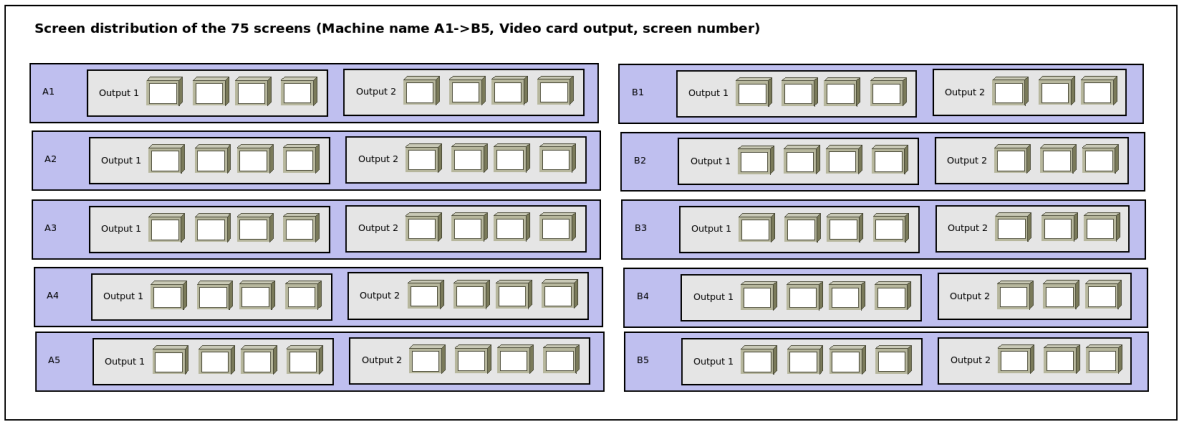


Figure 15: The layout of the Wild's screen

I have experienced for updating or creating objects) and we need to update 20 objects we will need to send a buffer of 1MB. So we should have a bandwidth of 10Mbps for a network, which is reasonable. I measured in my environment on Wilder the time to update the colour of one hundred squares of size 8x8 pixels by capturing the dates of transmission and reception of the information packets on the server and client side. I repeated the experiment 8 times and reported the results in Figure 16. On average, although I update many objects (100) the packet takes 40ms to be sent. This is less than the delay between two frame refreshments. So, with the distributed environment, we get the same performance as if everything was done locally by a single very fast machine.

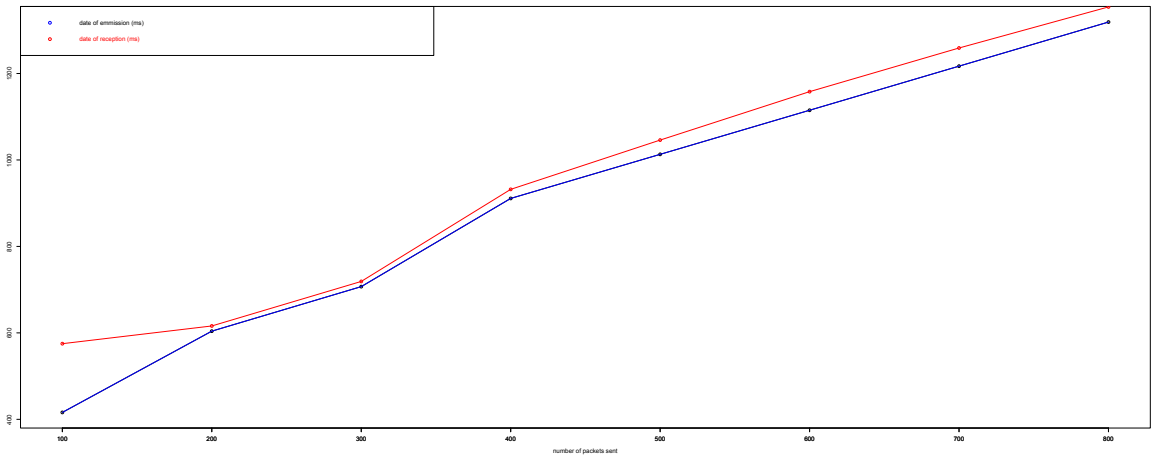


Figure 16: Date of transmission and reception of a packet for the update of 100 objects. The origin of the ordinate axis is taken arbitrarily.

## 5.3 Designing an interactive and collaborative dashboard

To create an interactive dashboard, I first collected a large database of coronavirus diseases. Then I designed an interactive map, where the user can select a country, zoom in or out. Then

the user can generate graphs about the selected countries. For example, if only one country is selected, he can create line graphs on the number of cases or the number of vaccinations against the coronavirus during a given period. It could also display other interesting figures such as the country's cardiovascular mortality rate or the prevalence of diabetes. If multiple countries are selected, a date must also be selected, and bar or pie charts can be created to compare the figures for the selected countries at the give date. All graphs are resizable and moveable, so their layout can be manipulated by the user. It is possible to close a chart or to drag the display area. (see figure 19).

The choice of possible interactions was made in order to validate the choices that were made to transform the input streams into pointer-type events. To do this, I listed all the common touch gestures (see figure 17) and used them on the dashboard application to interact. Then, since it is possible to handle all touch gestures with few lines of code, this validates the fact that the pointer event type is large enough to recognise complex gestures like touch gestures and also accurate enough (since touch gestures are easily handled with few lines of code).

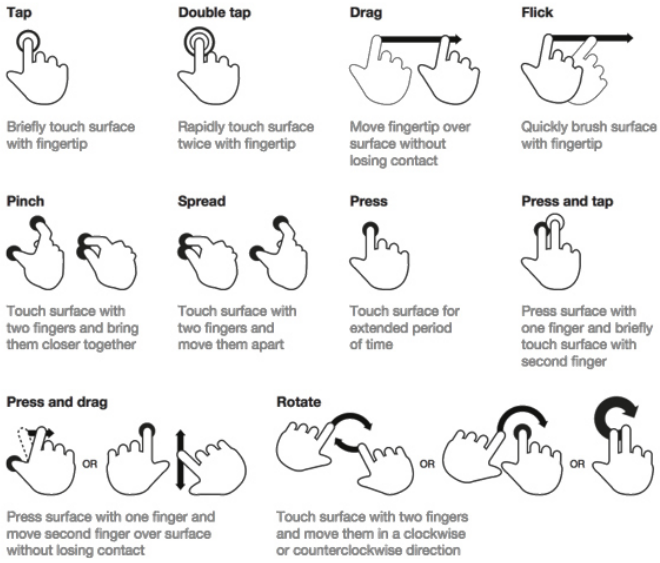


Figure 17: The common touch gestures

The creation of the dashboard uses a lot of data and all data can be drawn with the application. This allows to check the performance of the environment. I got satisfactory results, the time of transmission of information is not perceived in the display and the movements are fluid.

My application can be run locally, on a single machine. The display clients and the server client are then different windows of the same browser. A user can use their mobile phone or tablet to interact with the display. My application can also simulate a wall with two computers, one for the server client and one display client and the other for the second display client (see figure 18).

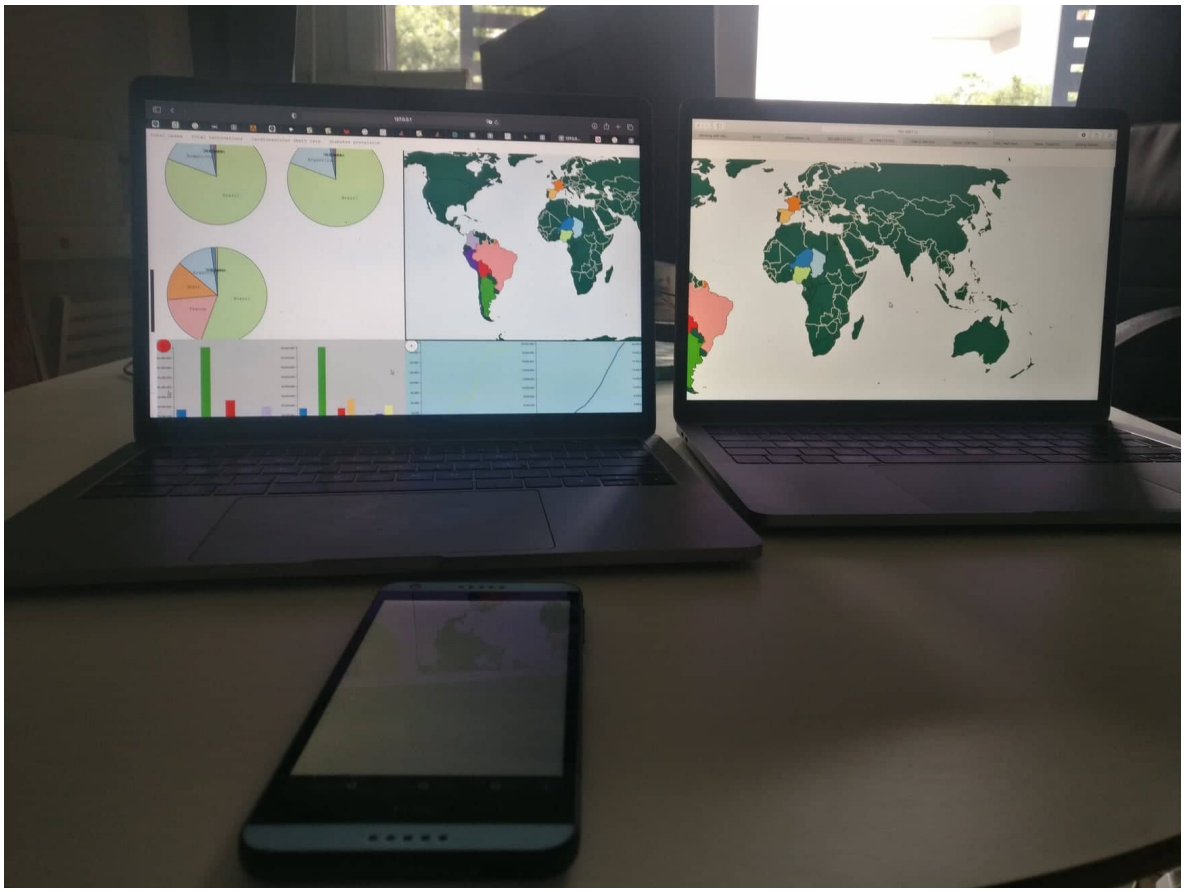

Figure 18: The application COVID-19 running on two computer for testing.


Figure 19: The application COVID-19 running the Wilder platform.

# 6 Conclusion

## 6.1 Summary

During this internship, I am building a prototype that takes advantage of web browser technologies to display a standard JavaScript application with a SRSD. I also allow to manage the interaction remotely. I chose to use web technologies because they are very portable, popular and efficient. My approach is close to that of the SAGE2 developers but I handle the communication for the client-side display in a different way. I decided to send DOM changes directly instead of sending pixel buffers of the full image representing the html page. As LHRD technologies involve a large amount of pixels, this solution will perform better. Indeed, the amount of information to be transferred to update the DOM must be considerably smaller than a pixel buffer. Finally, I carried out tests on my environment by taking advantage of the access to real infrastructures provided by the INRIA institute. This allowed me to demonstrate that my environment works properly and that it can be used for real-life applications.

## 6.2 Future Work

For now, my work is a prototype that shows how to create a faster environment for distributed web page display. It could be integrated into a distributed environment or extended for wider use. Indeed, it lacks integration such as audio management. Moreover, for the moment, the programmer can only display one web page but we can imagine an environment where it is possible to display several web pages juxtaposed at the same time and interact with them.

## References


[BOH11] M. Bostock, V. Ogievetsky, and J. Heer. $D^3$ data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[Dig] Wilder, digicospe project. `http://digiscope.fr/fr/platforms/wilder`.

[KEB+15] Clemens Nylandsted Klokmose, James R. Eagan, Siemen Baader, Wendy E. Mackay, and Michel Beaudouin-Lafon. *Webstrates*: Shareable dynamic media. In Celine Latulipe, Bjoern Hartmann, and Tovi Grossman, editors, *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST 2015, Charlotte, NC, USA, November 8-11, 2015*, pages 280–290. ACM, 2015.

[MAN+14] Thomas Marrinan, Jillian Aurisano, Arthur Nishimoto, Krishna Bharadwaj, Victor Mateevitsi, Luc Renambot, Lance Long, Andrew Johnson, and Jason Leigh. Sage2: A new approach for data intensive collaboration using scalable resolution shared displays. In *10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 177–186. IEEE, 2014.

[Nod14] Node.js. `http://nodejs.org/`, 2014.

[Omi14] Omicron. `http://github.com/uic-evl/omicron/`, 2014.

[PHNP11] Emmanuel Pietriga, Stéphane Huot, Mathieu Nancel, and Romain Primet. Rapid development of user interfaces on cluster-driven wall displays with jbricks. In Fabio Paternò, Kris Luyten, and Frank Maurer, editors, *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing System, EICS 2011, Pisa, Italy, June 13-16, 2011*, pages 185–190. ACM, 2011.

[Sha11] Share.js. `https://sharejs.org`, 2011.