Developing Distributed Collaborative Applications with HTML5 under the Couple Objects Paradigm

Nelson Baloian  
(Department of Computer Science, University of Chile, Santiago de Chile, Chile)

Diego Aguirre  
(Department of Computer Science, University of Chile, Santiago de Chile, Chile)

Gustavo Zurita  
(Department of Management Control and Information Systems, Faculty of Economics and Business, University of Chile, Chile)

Abstract: One of the main issues when developing distributed collaborative systems is to support synchronization processes. The Coupled Objects paradigm has emerged as a way to easily support these processes by dynamically coupling arbitrary user interface objects between heterogeneous applications. In this article we present an architecture for developing distributed collaborative applications using HTML5 and we show its usage through the design and implementation of a series of collaborative systems in different scenarios. The experience of developing and using this architecture has shown that it is easy to use, robust and has good performance.

Keywords: Coupled objects paradigm; synchronization; HTML5; mobile collaboration; distributed collaborative systems

Categories: H.3.1, H.3.2, H.3.3, H.3.7, H.5.1

1 Introduction

Collaborative systems are conceived to assist groups in communicating, in collaborating, and in coordinating their activities, either in the same physical place or distributed over many locations (Ellis, Gibbs, and Rein 1991). In order to support collaboration in a distributed scenario, software engineers develop distributed systems, which consist of several autonomous entities with their own local memory (Andrews 1999), communicating with each other by message passing (Ghosh 2007).

For enabling communication and data management in distributed applications, developers typically use middleware by encapsulating the code, implementing the low level TCP/IP communication layer between applications running on different hosts.

Since the early days of the distributed application development the need for a middleware that could simplify the programming of distributed applications was evident since a good portion of the code necessary to implement the communication between applications could be encapsulated thus allowing the developer to concentrate on the application’s logic. One of the first of such frameworks was Sun Microsystem’s RPC (Sirinvasan, 1995) schema developed for simplifying the development of the Networked file system (NFS) for the Solaris architecture. Another of the early runners was the CORBA (Vogel, 1995) architecture, which was
developed with the aim of serving as a communication platform among applications developed for different architectures in different programming languages. However, the architecture was not well specified and various vendors implemented different versions of this architecture which were incompatible among them, this CORBA did not fulfill the expectations.

The aim of these systems is to make distributed applications programming not too different from the programming of stand-alone applications, by encapsulating the communication protocol and offering the programmer a nice interface for executing code in another computer.

Many other platforms and architectures have been developed thereafter (Urnes and Nejabi 1994) for suiting a certain class of applications with common requirements. They differ on the distribution schemes of the shared data, communication mechanisms, and application architecture they support (Lukosch 2002). Rendezvous (Hill, Brinck, Rohall, Patterson, and Wilner 1994) and Suite (Dewan and Choudhary 1992) are groupware platforms, which use a central distribution scheme for the data of collaborative applications. GroupKit (Roseman and Greenberg 1996), DECAF (Strom, Banavar, Miller, Prakash, and Ward 1998) and MASC (Aldunate, Ochoa, Peña-Mora, and Nussbaum 2006) use a replicated distribution scheme. Guerrero and Fuller (Guerrero and Fuller 2001) propose a pattern system to support the design of collaborative applications. Patterson (Patterson 1995), Dewan (Dewan 1995) and Roth (Roth and Unger 2000) propose various taxonomies for groupware platforms, especially with respect to the shared data distribution schema they support. They can be grouped in the following four classes:

- Centralized: A single server maintains the data for all applications. Every application has to contact this server for retrieving the state of the shared work and/or make changes. This may cause bottlenecks if the server is loaded with too much synchronizing work.
- Replicated: Every participant has a copy of all the shared data and the application is exactly replicated at every site. The application has a better performance since it has to access local data only. However, the need to keep these data coherent and synchronized at every site may lead to very complex algorithms, especially to deal with problems such as latecomers (Lukosch 2003).
- Asymmetric: There is no pre-defined central server, but one of the participants takes that role.
- Semi-replicated: there are multiple servers in this schema. Compared to the Centralized schema, this may lead to a shorter response time and is more robust, since the crash of one of the servers does not imply the crash of the system. It has some of the problems of data coherence found in the replicated schema.

Thanks to the rapid development of mobile technologies, wireless networks and the sinking costs of these technologies, today, collaborative systems tend to use mobile devices for enabling and enhancing communication among the members of a particular group. These new hardware capabilities can support ubiquitous group-work in ways that were not conceived before or were impossible to develop with desktop-based technology. However, mobile devices such as laptops, tablet PCs and smartphones usually run over different operating systems, which poses two main issues: (1) in terms of software development, it provides very few possibilities of integration; and (2) in terms of usage, they implement different human-computer
interaction patterns and use different data formats (Feisst, dos Santos, Mitic, and Christ 2005). In order to overcome these problems, we claim the best strategy to follow is to implement platform-independent applications. As of today, the most promising approach seems to be HTML5, which is expected to become a major revision and future standard in the development of Web applications (Hickson 2011).

In this article, we propose an architecture for developing distributed collaborative applications using HTML5 and we show its usage through the design and implementation of a series of collaborative systems in different scenarios. As a foundation basis, we extend the Coupled Objects paradigm, initially developed for desktop-based applications under the name of MatchMaker (Zhao and Hoppe 1994). The authors' experience designing and developing collaborative applications (Hoppe, Lingnau, Machado, Paiva, Prada, and Tewissen 2000; Sapateiro, Baloian, Antunes, and Zurita 2011; Zurita, Baloian, and Baytelman 2006) with previous versions of this architecture (Baloian, Pino, and Jansen 2007; Tewissen, Baloian, Hoppe, and Reinberg 2000) show that the coupled objects approach is easy to implement and allows the development of powerful collaborative applications in an easy way.

2 Related Work

The first Web applications were limited to consuming content generated by others in desktop-based services. Today, this trend has evolved to user-generated content and application platforms that are either desktop-based or mobile. The implementation of such Web applications has also evolved from plug-in technologies (such as Adobe Flash) to standard HTML5 development, new browser features, and faster JavaScript engines, among others. The extensive usage of JavaScript in today's Web applications induces the need for frameworks supporting faster development, better reusability and maintainability. As Model-View-Controller (MVC) is a well-known design pattern for server-side application development, it becomes even more important on the client side, leading to several prevalent JavaScript MVC frameworks. Therefore, JavaScript application frameworks will tend to lead the development of future Web-based business applications, including collaborative ones (Grüneberger, Heinrich, and Springer).

Grüneberg presents a series of collaboration extension for JavaScript frameworks, as well as a prototypical implementation of these extensions in SAPUI5 (a UI development toolkit for HTML5 development) and KnockoutJS (a JavaScript library that allows developers to create user interfaces with a clean underlying data model) (Grüneberger, Heinrich, and Springer).

BackBone is a JavaScript library that eases and helps structure the process of developing HTML5-based Web applications. As other popular frameworks, it is based on the model-view-controller (MVC) application design pattern, and as a JavaScript library, it implements a RESTful JSON interface (Osmani 2013).

This library is used for developing single-page Web applications, and for keeping synchronized various parts of Web applications, such as multiple clients and the server. Data is represented as models, which can be created, validated, destroyed, and saved to the server. Whenever an action in the user interface causes an attribute of a model to change, the model triggers a change event; all the views that display the
model’s state can be notified of the change, so that they are able to respond accordingly, re-rendering themselves with new information (Backbone.js).

3 The coupled object Paradigm

When developing distributed collaborative applications, software engineers generally deal with requirements linked to supporting synchronization either in communication, feedback or other kinds of processes. Therefore, there is a need for middleware capable of offering an easy way to develop these features rather than counting on a plain TCP/IP protocol implementation.

3.1 Synchronization Requirements in Distributed Applications

According to Tewissen et al., the typical synchronization requirements to take into account when designing and developing distributed applications for implementing collaborative learning activities in the classroom are (Tewissen, Baloian, Hoppe, and Reimberg 2000):

- **Dynamic synchronization**: it should be possible to start and stop the synchronization process at any point in the application's lifetime. Before and after the coupling phase, applications should continue to exist independently. This is in order to allow a swift switching from collaborative learning activities to individual learning and back.

- **Partial synchronization**: it should be possible to individually synchronize each component of an application's interface with a component of another application's interface, thus allowing the synchronization of applications with completely different interfaces. In fact, in many situations the application used by the teacher will have different functionalities from the application used by the students. In other words, this feature supports the implementation of various roles in a distributed environment.

- **Replicated architecture**: it should be possible to synchronize multiple and potentially different applications already existing in a distributed environment. This is in order to transform many already existing single user applications in collaborative applications. This also eases the implementation of collaborative applications by allowing the developer to concentrate on its functionalities instead of the synchronization.

Although these synchronization requirements were stated for a collaborative learning scenario, it is easy to see they are common to many applications. Therefore, it is good idea to develop a generic solution consisting of an architecture allowing programmers to simply develop distributed applications based on the Coupled Objects paradigm.

3.2 The Coupled Objects Paradigm

The Coupled Objects paradigm states that distributed applications can be synchronized by dynamically coupling and decoupling a set of objects of various applications running in different hosts. When two objects are coupled, the events
generated at one of these objects are automatically propagated to the rest of the coupled objects. This approach differs from more classical ones like sharing a remote single object or having a replication of one object in each application.

This paradigm supports synchronization by combining two architectures for collaborative systems: (1) a centralized server, and (2) a replicated architecture. By combining them, we get the advantages of both solutions and we remove the disadvantages. A centralized server allows a simple and effective implementation of synchronization; a problem of a centralized server is that it is impossible for clients to continue working if it is no longer reachable. By using a replicated architecture, it is possible for an application to keep working in a standalone manner even if the server is unreachable.

3.3 Previous Implementations

In this section of the article, we briefly present the different previous implementations of the Coupled Objects paradigm:

The first version of this middleware was developed in 1993 at the GMD-IPSI institute in Darmstadt, Germany, within the research activities of the COSOFT (Computer Support for Face-to-face Teaching) research group. They intended to develop teaching/learning models and systems for supporting teaching/learning activities in an electronic classroom. In this version, coupling and communication facilities were added to an existing library developed to ease the development of applications involving graphical user interfaces, including functions to couple and decouple new widgets. Communication between applications was based on a client-server model. Applications that were going to be synchronized needed to register first with the server with a distinctive name. Coupling and decoupling of objects was pairwise with objects living in different applications. The naming of objects was hierarchical, where a suffix ‘one’ corresponds to the first component in the hierarchy. Coupling was also hierarchical, which means that by coupling a parent object (container), all contained objects were also coupled. The library also provided a RPC facility: a callback function of a widget of any registered application could be invoked from another registered application (Zhao and Hoppe 1994).

The second version, developed at the COLLIDE (Collaborative, Intelligent Distributed Environments) research group in Duisburg, Germany, was implemented in C/C++ over the Windows operative system and was the first one to be called MatchMaker. Still centered on the topic of coupled user interface objects, this new version was successfully used as one basis for software implementations in a computer-integrated classroom scenario (Hoppe et al. 2000).

The Java MatchMaker (JMM) implementation is a complete reimplementation that uses Java specific features such as: RMI, Reflection, Dynamic Class Loading, and common object oriented design principles. The enhancements with respect to the previous version were mainly oriented towards integrating and taking up new standards in the field of distributed computing, thus resulting more compatible and more standardized (Tewissen, Baloian, Hoppe, and Reimberg 2000). An interesting addition of this

A third implementation was developed on 2007 (Baloian, 2007) in order to overcome the problem that arise when more local area networks (LAN) did not allow applications to contact other ones outside the LAN using another protocol than
HTTP for security reasons. Moreover, they also restricted applications running inside the LAN to open server sockets on ports other than the assigned to the HTTP protocol, which is port 80. This made the previous implementation very difficult to be used since it RMI uses other ports. The solution was to re-implement MatchMaker using SOAP over HyperText Transfer Protocol (HTTP) as the transportation layer for communicating clients with the server, and implementing the MatchMaker server as a Web Service. HTTP is normally open to firewalls and it is easily accessible from clients behind proxies or NATs. To accomplish this task, we used Apache Axis as the platform for Web Services and consumer clients in Java. Axis is an Open Source SOAP engine written in Java. At the sever side, Apache Axis works as a servlet, so it needs to be deployed inside a Web Servlet Container such as Jakarta Tomcat, WebLogic or the Web Application Server of J2EE. Apache Axis provides classes allowing clients to connect to a server, communicate with a remote application using the SOAP protocol and invoke Web Service methods with little effort from the developer. This new version of MatchMaker, known as SOAP MatchMaker, works the same way as the current RMI MatchMaker version. It only differs in some aspects, being the communication between clients and server the most notorious one.

In the RMI MatchMaker version, clients act as servers listening for events from the central server. In this new approach, the clients cannot act as servers anymore; they are forced to act in passive mode. This means clients need to contact the server to find out if new events affected the coupled objects. The basic mechanism consists of every client having an event queue at the server side. Clients are constantly requesting the server for new events, and every time a new event is available, it is sent to the client. If no new events are available, the client keeps waiting until a new event is available, but the connection with the server is kept alive.

4 Implementing the Coupled Objects Paradigm with HTML5

Over the last decade we have seen an explosion in the development of new collaborative systems using mobile devices that incorporate and utilize their communication capabilities to support collaborative work in ways that were not conceived before or were impossible to implement with desktop computers. Applications allowing users to collaborate in real time over wireless connected mobile devices have attracted the attention of many authors. Some examples on how mobile technology helps improve collaborative work are:

- Rescue efforts can be more easily coordinated in emergency situations and disaster areas (Monares, Ochoa, Pino, Herskovic, Rodriguez-Covili, and Neyem 2011)
- People attending a meeting can share ideas and data by means of their mobile devices (Zurita, Baloian, and Baytelman 2006).
- A team of construction workers or garden designers on a site without a network infrastructure can share blueprints and schematics (Sapateiro, Baloian, Antunes, and Zurita 2011).
- Educational activities involving students and teachers in collaborative room environments (Pinkwart, Hoppe, Milrad, and Perez 2003).
An important problem with developing collaborative applications for mobile devices is their heterogeneity. They vary from usual laptops, mobile phones, tablet PCs, advanced MP3 players, using different operative systems thus providing very few possibilities of integration. As (Feisst, dos Santos, Mitic, and Christ 2005) already described, these various devices implement different human-computer interaction mechanisms and use different data formats. Additionally, a major challenge in this scenario is providing applications that run on a number of different operating systems for mobile devices. As an answer to this challenge, we argue that the best strategy would be to implement platform independent applications. Currently, the most promising approach seems to build applications based on HTML5 with JavaScript. Until now, the standard way to develop a Web-based application was by implementing multiple views, each one presenting a different functionality to the user. Each interaction with the application would, in most cases, generate a request to the server who would respond with a new page containing the changes resulting from running a piece of code. By using JavaScript and the advanced features of HTML5 like WebDatabase, WebSockets, Canvas, LocalStorage, and LocalFileSystem, it is now possible to develop much more complex applications running inside the browser, which is eventually capable of implementing rich interaction with the user in a single view requiring less interaction with the server.

5 Implementing the Coupled Objects Paradigm With HTML5

Given the current trends on the development of mobile applications, it turns out necessary to count on an implementation of MatchMaker that supports the synchronization of applications running on mobile devices having different operative systems. It is also requires to support the synchronization of applications running on mobile devices with applications running on desktop-based and tablet computers, or even any computer device. Given these requirements we opted for developing a new version for coupling applications developed in HTML5, thus being able to support the conversion of single-user application to multi-user applications for devices running an HTML5 compatible browser, without having to rewrite a particular implementation of the middleware.

The HTML5-based MatchMaker implementation shares an important feature with the previous one using Web Services: the client-server communication is implemented using the HTTP protocol. This means the client is responsible for taking the initiative for any communication that is needed. In fact, the implementation strongly lays on the JavaScript function setInterval(func, time), which schedules a call to a function called func every time milliseconds. Although JavaScript does not have a suitable object model that can be used in this scenario, there is a library called JSON (JavaScript Object Notation) defining a very useful one.

5.1 Object Coupling with HTML5

When developing the foundations of the architecture, the following requirements need to be accomplished:
• **Coupling DOM Nodes**: when a browser processes an HTML page, a node in the Document Object Model (DOM) represents each component. These nodes can be accessed and modified through JavaScript, thus having a total control over the page structure. Therefore, coupling the DOM nodes should be considered when coupling HTML-based application interfaces.

• **Event-based Synchronization**: since all the previous implementations of MatchMaker use event-based synchronization, we might keep on this in the new version.

• **Sessions**: in order to keep track of connections, the architecture should propose a session management mechanism, where users could connect to them.

• **WebSockets**: one of the limitations in the SOAP implementation of object coupling is that it turns out necessary to emulate how messages are received from the server through periodical queries. However, HTML5 defines the WebSockets structure, which are bidirectional communication channels allowing the server to send messages to clients without the need to ask for them.

• **Protocol Switching**: switching the main communication protocol from RMI to SOAP in the last implementation of MatchMaker required a considerable effort. Therefore, a new implementation should allow communication protocol switching with the least effort possible.

However, when envisioning an implementation based on the previous list of requirements, two issues arise:

• **Creating new objects to couple**: if we only couple objects through their states, it would eventually turn out difficult to manage the creation of new coupled objects. This happens because the creation of a new object should be implemented as a state change in an object that is already coupled in order to propagate this message to the other clients. That way, we need to use at least an object that has a reference to all the others, and manage as well the creation or deletion of objects as a change on their states.

• **WebSockets is not fully integrated into all the Web browsers**: even though the use of WebSockets will turn out as the best solution to implement the communication protocol, it is not well supported by the majority of Web browsers today.

Therefore, for this HTML5 implementation of MatchMaker, we will enlarge the original list of requirements in order to consider the following features needed to correctly support object coupling:

• **Selective Coupling**: developers should be able to choose between coupling DOM Nodes or JavaScript objects. In the first case, the developer should use event callbacks to couple the nodes. In the other case, the developer should use methods to send the coupling messages.
• **Mixed Synchronization**: the architecture should support both event and state-driven synchronization. That way, there would be no further problems when creating new objects, since this can be modeled as an event rather than a state change.

• **Multiple Protocols**: since HTML5 will eventually become a Web standard, the use of WebSockets should be seen as a strong recommendation. Therefore, communication should be implemented through WebSockets and a backup protocol, in case the Web browser does not properly support the former.

5.2 **Communication Management**

In order to develop applications using the proposed architecture and couple already-implemented applications, it is important to keep a broad separation between the business logic of the intended application and the synchronization mechanisms offered by the architecture. In order to achieve this, we need to use two handlers: the first one deals with JavaScript objects modifying their methods, thus integrating the communication process; the second one enables the communication process as an event-manager (such as the `onClick` event triggered by an HTML button). Therefore, in order to trigger the synchronization between coupled objects, the intended application should follow these simple steps:

- Initializing communication with server;
- Joining a session; and
- Coupling the desired objects.

In order to manage communication processes, we used a client-server architecture. In this architecture, the server publishes two interfaces to be used by the client: one for managing sessions, and other for synchronizing objects.

5.3 **Server-Side Implementation**

With the implementation of the server application, we aim to cover three particular issues in the design of this architecture: (1) to provide a structural basis to develop distributed collaborative systems in an easy way by integrating different communication protocols; (2) to provide support for event-based synchronization; and (3) to provide new alternatives to particular cases of synchronization, where both the client and the server can be accommodated, thus being easily able to transform mono-user applications into multi-user ones.

The proposed architecture manages the different messages ensuring they are validated, persisted and propagated to their correct clients. In order to avoid leaving clients waiting for an answer, this mechanism is implemented in an asynchronous way: therefore, when processing a message, it is queued waiting to be sent, and afterwards it is persisted. The server also monitors concurrency, being in charge of receiving the different calls and deciding when to propagate them.

When the server receives a message, before queuing it, the server can validate the message through a state manager, which can eventually have other validators that will be in charge of deciding if the message is effectively valid or not. If the message is
not valid, it is discarded and this answer is propagated through the client who initially sent the message.

If two messages are simultaneously sent to the server by two different clients who aim to modify the same element, this could turn into inconsistency issues, as what one client sees is actually different from what the other client is seeing. In order to solve this issue, the server evaluates if there is any risk of causing conflict with messages that are not actually being sent. If there is conflict, the server notifies the client, the message is not discarded and it is finally queued as an echoed message. An echoed message is also sent to the client. Therefore, if it is an event, it will be executed only if it is received as an echo, thus respecting the execution order. If it is a change, it cannot be discarded, but this is not an issue since the original state change will not modify the process. Figure 1 shows the server architecture.

The following list summarizes the steps to implement the server application:

- **Define one or more communication protocols, and the entry points and adapters**: when defining a protocol, it is also necessary to consider adapters and entry points linked to it, which will be in charge of receiving and interpreting the client messages in order to be processed by the server. An adapter is the component that the server uses to receive and send messages from/to clients. The interface provided by an adapter is published through the clients by an entry point. An entry point is a communication interface, in a particular protocol, which publishes the required functions to be used by a client in order to use a particular interface (in this case, the adapter). A client recognizes different entry points according to their URLs. That way, two entry points for two different protocols can coexist in a same server, thus being accessed through different URLs.

- **Define persistence and validation strategies**: since we expect the server to support state and event-based coupling, we need to manage the persistence of a single state for each object in the server. Therefore, the architecture provides a state manager specific for each case. This manager can process each received message and use it for updating the state of pertinent objects, thus generating the necessary messages for updating the state of a particular object when any adapter couples it.

- **Define a session management strategy, its protocol and entry point**: in order to group different clients, we extend the session functionality, introduced by the RMI and SOAP implementations of MatchMaker. Therefore, a client can either create a new session, or join an existing one. Therefore, the messages will only be propagated through clients that belong to a same session.
The coupled object server is in charge of managing the message flow among clients. By clients we mean the applications which should be coupled using this paradigm. In order to provide a solution for a wide range of scenarios we followed an extensible approach in order to support the usage of various different client-server communication protocols. For this, the server is composed by two components: one is a core component called **coupledObjectServer** which implements common functionalities for all scenarios and a secon one is the one which should be extended or modified if the developer wants to change or include a new protocol for client-server communication. The implementation presented in this work uses mainly the http protocol in order to support the usage of HTML5 at the clients’ side.

The main element of this component is the **CouplingManager** class which is in charge of receiving and propagating the messages. The interaction with this class is implemented through the **Adapter** interface. The creation of new CouplingManager instances and the access to them is implemented through the **SessionManager** interface.

The class implementing the **Adapter** interface recieves the messages from the clients and will pass them to the corresponding CouplingManager and viceversa. The HTML5 implementation of this class uses JSON messages from the client. Therefore the class implementing this adapter is called **JSONMessageAdapter** and their functionalities are published (made accessible to http clients) by a J2EE servlet. This adapter has the particular characteristic that instead of sending information directly to the client, it stores the messages in a buffer waiting for the client’s request, thus adjusting the process to the request/response model of HTTP. Figure 2 shows the UML sequence diagram for messages since they are originated at the coupled object until they reach the coupling manager.

The class implementing the **SessionManager** interface offers functionalities for creating new clients, new sessions (each one with an own CouplingManager) and for data management. The HTML5 implementation of this class is called **GenericSessionManager** because it implements these functions in a generic way. The functionalities for joining and leaving a session are published by a J2EE servlet. Creation of clients and sessions is automatically managed. Figure 3 shows the UML sequence diagram of the creation and joining of a session procedure.
Figure 2. Sequence diagram of the message flow since they originate at the client, triggered by a change in a coupled object, until it reaches the

Figure 3. Sequence diagram of the session life-cycle its creation is requested from the client until the client joins it

5.4 Client-Side Implementation

As we see, an application developed with the coupled object paradigm uses a standard, centralized message propagation strategy, which consists in each client generating messages which are sent to the central server which delivers it to the final destination.
However, the coupled objects paradigm and particularly its HTML5 implementation offers a mechanism which allows a developer to define what, how and when a message should be sent to other applications. This is done by transforming normal objects into coupled objects during the application’s runtime.

In the coupled objects paradigm messages are sent and/or received and processed by coupled objects. These objects are normal application objects which are modified in order to give them additional functionalities for sending and receiving messages. In previous implementations, objects had to be defined as such in the program itself in order to allow them to be coupled during runtime. In the HTML5 implementation it is possible to transform any common object during runtime. This transformation is done inside the ClientAdapter class which receives an instance of an object and then modifies its inner code, without altering its interface, in order to grant it the coupling feature. At the same time it builds a Handler inside the object, which will be in charge of implementing the communication between the object and the server. Figure 4 shows this process.

![Figure 4. Schema for converting a normal object into a coupled one.](image)

This strategy allows a very easy conversion of applications which have been developed for a single-user scenario into a multi-user one without modifying its code, thus allowing it to swiftly and flawlessly switch from one scenario to the other. This implementation is also very easy to use from the point of view of the application developer but nevertheless it is also very versatile because it allows a developer to choose in which cases and which type of messages will be generated in order to implement the coupling feature. In first place the developer can choose if the coupling messages will transmit a message describing the event the object received or the state the object has.

**State messages** contain a complete representation of the object characterized by its instance variables. An object receiving a message like this will modify all its instances’ values by those of the message corresponding to the instance variables of the sender object. In order to use this type of messages it is necessary to define two functions: `getState` and `setState`, which can be defined by the code of the object itself (like in the previous implementations of coupled objects) or it can be defined and introduced to the object’s code at the moment when the object is coupled during runtime. This is the coupling type which is recommended if the size of the object is “reasonable small”. By the size of an object we mean the size of its internal representation corresponding to the instance variables, so the size of an object will be the number of bytes necessary to store all the variables. It is difficult to give a concrete number for
“reasonably small” since it depends on several aspects of the application itself and the environment in which it is running. Of course, using fast computers and a fast network connecting them allows handling bigger objects, but it also depends on how frequently needs the application to send and/or receive messages from the others. State message should be preferred in order to avoid some coordination problems because it has fewer problems at the moment of handling latecomers, which is applications which join the collaborative session later will receive the current state of the object. In the case of a collaborative text editor if the latecomer receives the whole state it will receive the last version of the complete text instead of the following changes only.

**Event messages** represent an event which occurred on an object which modified its instance variables. Differently to the state messages, these ones only include information about the event, which in most cases will be fewer that the whole state of the object. In most cases, an object will generate more messages if these are event messages instead of state messages, so its usage is recommended for objects which given their size and nature the definition of state might be not clear. This is especially true in objects which have variable number of sub-objects inside like lists.

The main component in the client-side application is the Client Adapter. This object uses the same interface of the server adapters, and there is only one per client. It relies on: (1) AJAX calls to coupling or decoupling actions, defined at the URL context; and (2) the `EventSource` API provided by the HTML5 specification, that serves to emulate events that are sent by the server through periodic AJAX calls. Figure 5 shows the client architecture.

![Client-side architecture](image)

**Figure 5. Client-side architecture**

It is possible that the server validates a message sent by a client. However, if the server decides that the message is not valid, it will notify the client. Therefore, in order to follow the server validation, the client sends the message, and then it locally executes the event only if the server allows it. With state messages, this is not possible, since any event needs to be executed before generating a new state: then, the validation of state messages is not supported by the client nor the server.

The second stage at implementing the client application is allowing message generation and managing received messages in order to complete the event and state change propagation. In order to achieve this, we used handlers coupled either to a JavaScript object or a DOM element. Handlers are used to: (1) couple to the object they are managing for sending the event or state change messages; and (2) register in the client adapter and execute the required actions when their object receives a message. A same object can have one or more different handlers for controlling more precisely how the coupling is performed. In this architecture we support two kinds of handlers:
• **Object Handlers:** designed to couple JavaScript objects by accessing their methods. When it is linked to an object, it can intervene over its functions in order to send the corresponding message by creating and assigning functions at runtime through JavaScript.

• **Bind Handlers:** designed to manage coupling with callback or binding functions when a particular event is triggered. By using JQuery, a JavaScript library, we can perform this kind of binding either with DOM nodes or with JQuery objects that can define their own events. As opposed to object handlers, a bind handler can only execute callback functions linked to a particular event, and cannot reproduce the event itself.

In order to simplify the use of the client API, we provide a JQuery plugin called **coupler**. This plugin offers a unified interface for coupling objects, encapsulating the creation and registration of handlers. To summarize, Figure 6 shows the global architecture of the communication process.

![Figure 6. The Global architecture](image)

6 **Using the Proposed Architecture**

In this section, we show how to design and develop distributed collaborative applications using a particular implementation of the HTML5-based *MatchMaker* architecture. First we explain how the various components of the architecture were implemented in order to have a simple way to couple objects existing on a browser running an HTML5 page with JavaScript.

The components of the architecture were implemented in the following way:
• **Protocol:** since the majority of current browsers do not fully support the WebSockets structure as a communication channel, we implemented a HTTP-based protocol based on POST and GET queries to send and fetch information to/from the server.

• **Adapter:** we implemented an adapter that will receive the messages in JSON and will internally transform them into messages understandable by the objects.

• **Entry point:** we used a Servlet mapping to three different URIs: coupling, for sending and asking for messages to the server; couple, to couple an object; and decouple, for decoupling an object. We managed errors with HTTP status messages, using the HTML5 current specification.

• **Persistence:** each object is automatically persisted, instead of using a method for achieving this.

• **Sessions and clients:** sessions are shared between many clients, instead of being attached to only one. That way, each time an HTTP session is created, we link a user to it and we manage its lifecycle with the object coupling.

• **Entry point:** since a user can only connect to a client in a new session, or over an existing session, the management of creation and assignation of these is blind for users. In order to achieve this, we implemented a Servlet that used a particular URI: if it gets a GET query, it checks the identifier of the particular model to be accessed. We then register the user, obtained through the HTTP session and respond to the client with the HTML code that implements the client for the application, where we include the relevant user data.

With this implementation, object coupling is achieved in the following way:

• Initializing communication with server:
  
  ```javascript
  var adapter=new ClientAdapter(ServerURL);
  ```

  The **ServerURL** parameter is a string with the URL where the coupling server is located (for example, [http://saduewa.dcc.uchile.cl/HTML5Server](http://saduewa.dcc.uchile.cl/HTML5Server)).

• Join a session:
  
  ```javascript
  adapter.joinSession(ssid);
  ```

  The parameter **ssid** is a string with a name for a session. All objects registered in the same session will be coupled among them.

• Couple the desired object:
  
  For JavaScript (JSON) objects we use:
  ```javascript
  adapter.coupleJSONObject(object, type);
  ```

  The parameter **object** is a reference to the object to be coupled. The **type** parameter specifies if the coupling is by event or by state.

  In the case of DOM Objects with callbacks (like buttons) we use:
  ```javascript
  adapter.coupleDOMObject(object, type);
  ```

  Its parameters are the same as for the previous function.
• Decoupling objects:
  
  adapter.decouple(object);

Next we present different examples of how the Coupled Objects paradigm and the HTML5-based implementation of MatchMaker can be used for developing collaborative applications.

6.1 Converting a complex single-user application in multi-user: Mobiz

Business Process Modelling (BPM) allows organizations to support the abstraction of business processes from technology infrastructures. It initially focused on the automation of business process, the coordination of tasks and the management of data and other resources, such as human-driven processes in which human interaction takes place in series or parallel with the use of technology (Sheth, Georgakopoulos, Joosten, et al. 1996). Authors have identified the lack of flexibility as one recurrent problem for process elicitation. In order to support an agile process elicitation procedure we developed a mobile collaborative application named Mobiz. (Baloian, Pino, Reveco, Zurita, 2013) This application will help to leverage business process models on the way using mobile devices thus allowing the construction and edition of process models in a collaborative way. Figure 7 shows the main user interface of Mobiz.

![Figure 7. Mobiz user interface](image)

The look and feel of the desktop interface was partially based on the one proposed by the BIZAGI process developer [http://www.bizagi.com/], mainly because it is a well known product in the business processing environment. Thus, process engineers will rapidly learn to use the new tool. Furthermore, the interaction principle for creating new elements of the model starting from a contextual popup menu of existing elements works very well in both, the desktop and the mobile environments. One of
The most interesting features of MOBIZ partially inherited from BIZAGI is the use of a context menu when interacting with an artifact on the workspace. We added some additional features in order to take further advantage of this interaction element. This contextual menu does not display actions to perform but a list of possible types of artifact that may follow the current one on the process being constructed. This application follows the MVC (Model-Controller-View) architecture: it has a model object containing all the BPMN graph elements which are created by the user using a controller object. A view object is in charge of displaying these elements in a canvas.

In order to turn this application into a collaborative one using the implemented architecture, we just had to couple the model object of the application. In this way, all changes made to the graph by one user of one application will be distributed to all applications that joined the session. The view module of each application is in charge of displaying all changes occurring on the model. Figure 8 shows the class diagram of the most important parts of the Mobiz application.

```javascript
//Initialize adapter
var adapter=new ClientAdapter(serverUrl);
```

The BPMNDiagram class manages the creation (and destruction) of each element shown in the figure and maintains the relation with each created instance. Also each instance of a class has an associated DOM node. This node receives the events generated when a change in the object has to be performed when a coupled object changed. In particular, an "<element>Created" event will be triggered and passed to a node which represents the BPMN diagram on the interface each time a new element of the diagram is created, where <element> is the type of element which has been created. Also, when an element is modified a "change" event is triggered and passed to the node representing it.

The single-user HTML5 Mobiz application is a considerable complex application with more than 1,500 program lines. However, despite its complexity it was very easy to convert it to multi-user application using the coupled object implementation described in this work by just adding the following piece of code:
adapter.joinSession(sessionName);

adapter.coupleObject('diagram', bpmnDiagram.element, {messageType: 'EVENT'});

jQuery(bpmDiagram.element).bind('taskCreated relationshipCreated eventCreated gatewayCreated',
    function(event, object) {
        // All created elements of the diagram are // coupled at creation time
        adapter.coupleBinding(object.id, object.element, {
            messageType: 'STATE',
            bindings: ['change']
        });
    });

6.2 Using multiple protocols and asymmetric coupling (Microblogging)

Students of a business degree program have to attend a course in which they are asked to identify situations in real contexts, for which, the introduction of solutions based on Information Technology might be an opportunity to improve the life or solve a problem for citizens that frequently circulate around that area, live or work there. They have to geographically identify spots with troubles and/or opportunities and collaboratively propose, discuss and select the most appropriate solutions.

Figure 9. Two screenshots of the application, the small one taken from a Smartphone, and the big one from a desktop computer screen.
Figure 9 shows the main view of the proposed prototype as seen on a screen of a Smartphone and a desktop computer. This application support students performing following activities:

1. **Information gathering and brainstorming the ideas in a divergent mode.** Students can geo-reference concrete physical locations where the proposed idea is going to be put into reality. This will include a title, a textual description and maybe pictures in order to better contextualize the proposal means it has been made already public. In the Smartphone view of Figure 4 the proposal of the idea is being created.

2. **Find alternatives, information processing and comments of alternatives using divergence and convergence modes.** In the upper part of the desktop interface view shown in Figure 9 we can see the searching tool which searches for ideas by author’s name, associated comments words or date of creation. The result of the search will be shown as a list under the search tool and the corresponding location of all proposals matching the search criteria will be shown on the map.

3. **Make choices in a convergent mode.** Each comment to a proposed idea has a ranking mechanism associated which allows students to assign points in favor or against the proposal. This mechanism can be seen in the desktop view of the interface in Figure 9, where the last comment shown at the bottom right gives two points in favor for this idea.

An interesting characteristic of this application is that it takes data from and exports data to very well known social networks like Tweeter or Facebook, since the it involves users posting comments and others responding them. In this way, users can interact with this application while they are connected to their preferred social network posting new comments or reading others’. At the client side, this application has the same structure then the previously described one, thus the way of coupling and decoupling objects is the same. However, for this application we used the extensibility feature of the server which was used to create an independent application (in Java) with the only purpose of posting and receiving information to and from the social networks.

![Diagram](image.png)

*Figure 10. The figure shows the original structure of the CoupledObjectWebServer class and the additions to implement the new protocol.*
In order to implement this, a second communication protocol called Hessian was developed which basically emulates a remote procedure call protocol (Like Java’s RMI) but over the HTTP protocol. Figure 10 shows the structure of the classes implementing the CoupledObjectWebServer. The blue boxes show the classes provided by the generic server. The orange classes show those which were added in order to implement the interface with the social networks. We can see that since Hessian implements a HTTP protocol it can only be initiated (executed) by a client’s request. This means the JSONMessageAdapter, which was used for the generic solution, can also be used in combination with this protocol, and that is the reason why there was no need to implement a special new SessionManager. Figure 11 shows the architecture of the whole application.

![Figure 11. The figure shows the architecture of the whole application including the interface to twitter and facebook](image)

### 6.3 CollabCode: A Collaborative Learning Scenario

This application allows a teacher to monitor student work online while the students are engaged in which consists in developing a piece of JavaScript code. It also allows him/her to give feedback to the students log in a web application which they can use to write a piece of code in JavaScript, they can run it and see its output on the same page (see Figure 5). The student writes JavaScript code in the text area on the right side of the figure. By pressing the button labeled “Run code”, the code is executed and the output is shown in the text area on the right side of the figure. By pressing the button labeled “Get code”, the student gets the modifications made by the teacher.
On the other hand, the teacher’s interface shows a set of buttons, one for every student that has logged, labeled with the student’s ID. The teacher can display/hide code being developed by the student by pressing the respective button. She can also write/correct the code with the student by pressing the “Save” button (see Figure 6). The teacher’s interface shows at the left the buttons for displaying/hiding the code for each student. At the right side it shows the students’ code labeled with their respective ID.

In order to develop the students’ application with MatchMaker, it is necessary first to provide a login page where each participating student provides his ID. This ID will be later used to identify each student’s application. Then, an HTML5 page is loaded which renders the interface shown in Figure 5. This page has a JavaScript piece of code which declares a variable named after the student’s ID. It then adds a String attribute, which will be synchronized with the student’s text area where the code is being written. This variable will be registered with the server as a coupled object. Each time the “Run code” button is pressed the current code is submitted to the server. By pressing the “Get code” button the code currently stored will be retrieved and displayed. This can be used to “download” the code with modifications added by the teacher.
In order to make these two applications collaborative, each time a new student starts an application the teacher’s application gets a message in order to create the necessary objects to show the new student’s workspace. Then the student’s and the teacher’s applications join a new session in order to couple their respective buttons and text areas.

6.4 Collaborative Sketching Scenario

There are many mobile computing scenarios where collaborative sketching might help accomplish the common goal (see Zurita, Baloian, and Baytelman 2006) for an example). With this application, users can utilize the touch sensitive screen of their mobile devices to collaboratively draw sketches with a stylus or the finger. Such an application may generate a lot of traffic from the clients to the server and back. Therefore, this application was also developed to test the robustness of the HTML5-based MatchMaker under such conditions.

The application consists of a single webpage which is downloaded by all participants. The JavaScript code captures all `mousedown`, `mousemove` and `mouseup` events registering their coordinates and drawing lines among these points in order to render the sketch. In order to make this application collaborative, a JSON object containing an array is created where all points corresponding to a single stroke are recorded. This object is coupled with the server. Every time a sketch is completed (`mouseup` event) the object state is updated at the server. At the same time, and every 200ms, the application retrieves the new version of the object and draws the new stroke on the Canvas element, adding this one to all previously drawn. Since the key of the JSON array object is the same for all application instances, each user will retrieve the new stroke, no matter from whom it originated. In order to distinguish the user who generated the stroke, the JSON object also contains the user ID. In this way, the color of the stroke will be chosen according to this information. For collaborative applications where the interaction between users in synchronous and a lot of traffic is generated, the response time is critical for their usability. In order to analyze the delay for updating the sketch status we conducted an experiment measuring the round trip time between client and server for 120 JSON objects containing strokes of various sizes, sending a stroke every 2 milliseconds. The size in kb of the JSON objects representing the strokes varied from 1.0 to 1.5. The average response time was of 55 milliseconds with a standard deviation of 30 milliseconds, which can be considered good for human users.

7 Conclusions and Future Work

This article reports an architecture to support the design and development of distributed collaborative applications using HTML5. The motivation for this development was the need for a suitable middleware, which could simplify the development of this kind of application in mobile scenarios. Nowadays available mobile technologies are vast and heterogeneous, which requires that such a middleware to be as portable as possible. The strategy was to implement the Coupled Objects approach, which has shown to be simple, flexible and powerful, this time using HTML5, which allows this implementation to be highly portable. This
particular feature of HTML5 allowed us to develop a multi-platform mobile application based on the proposed architecture. We claim that HTML5 is a promising technology for developing compatible collaborative applications for mobile scenarios. However, it has been not fully adopted by all browsers yet. Google Chrome, Opera and Firefox are among the ones, which include most of the features. Even with these browsers there are some important features still not implemented, like the WebSockets, and this is the reason why they were not used in this implementation of MatchMaker. WebSockets allow a bidirectional communication between server and client (browser) thus liberating the client of periodically polling the server in order to retrieve the latest state of coupled objects.

References


Baloian, Neeson, José A. Pino, Carlos Reveco, Gustavo Zurita: Mobile Collaboration for Business Process Elicitation from an Agile Development Methodology Viewpoint. ICEBE 2013: 306-311


