Collaboration in Lively
Seminar Web-based Development Environments

Fabian Garagnon and Kai Schlichting

Hasso-Plattner-Institut, Potsdam
`{fabian.garagnon,kai.schlichting}@student.hpi.uni-potsdam.de`

Abstract. Wiki pages and developer journals are the home of dynamic and frequently changing content. Such applications written with Lively Kernel lack of the possibility that several users can edit different parts of a page at the same time.
We introduce a collaboration framework for Lively that allow users to see other users' changes in real-time. This report discusses a command-based approach that addresses this topic and shows one concrete implementation for this problem.

1 Introduction

Typical web activities like editing of a web page by several users at the same time are editing Wiki pages, writing papers, creating presentations and others. Those applications are made possible by Lively Kernel out-of-the-box since it allows changing the interface (and its behavior) of a web page in the browser. When saving a page, the whole DOM structure is saved into a subversion repository whereby changes made by others simultaneously get lost.

Collaboration tools enables several people working on the same thing in real-time. In Lively, such a framework could make possible that two users are writing a text together and a third user is changing the design on the same page. This way, changes can be applied in parallel without having to wait for other users finishing their work. Additionally, ideas can be shared immediately and other users can give feedback directly.

The main goal of this project is to solve frequent synchronization issues in a multi-user environment like the *Lively Kernel Webwerkstatt Wiki*[1]. This synchronization should be in nearly real-time so that every user can see the others working on a Wiki page. Normal activities like adding of new morphs, change position, resizing or editing text is synchronized. There should be no spontaneously changes (e.g. a morphing "jumping" from one place to another), therefore users see other mouse cursors and know in which part of the page the others are working.

Although no direct goal of this project, the resulting framework should be extensible by more advanced collaboration features: Conflict management won't

[1] http://lively-kernel.org/repository/webwerkstatt/webwerkstatt.xhtml

be available in the first step, but the solution should allow to use operational transformations [5] (which is an optimistic approach) to solve conflicts effective. Another neat but not as necessary feature would be a time-slider which gives each user the opportunity to travel back in time and view the Wiki page a few clicks behind (as seen in EtherPad[2] [1]).

The remainder of this paper is structured as follows: The next section introduces our command-based collaboration approach. Section 3 presents the framework design and architecture, while section 4 discusses special implementation considerations. Finally, section 5 concludes and points to future work.

2 Approach

This section explains our approach for implementing basic collaboration features into the Lively kernel. For this purpose a framework is needed that allows synchronizing changes of one user with others.

A first question that have to be discussed is the level on which changes should be observed and recorded. In browser environments, there are at least three potentially suitable levels: (a) Events from input devices (mouse, keyboard), (b) changes in DOM structure or (c) calls to the JavaScript API. Propagating mouse and keyboard events from one users to another (a) has the drawback that button clicks with non-local side effects (e.g. deleting a record on a server) would be called twice. Synchronizing DOM changes like a newly added SVG node (b) would create a common view on the currently opened page, but doesn't execute related JavaScript code: A new button is visible to all users, but no action handlers is assigned to the button since this requires JavaScript. The most promising solution to our problem is to listen to basic Lively API calls (c) (like `TextMorph#setTextString`, `Morph#setPosition` or `Morph#addMorph`) and propagate them to other clients. This way, the JavaScript object state keeps synchronized while the propagated method calls care about updating the underlying DOM.

To record the method calls made to the Lively API, we decided for a command-based interaction (Figure 1): When a method is called, a command is created locally in the client encapsulating all the information needed to execute and undo the method later on other clients. This information includes particularly the arguments of the called function and the current state for undoing (e.g. a call to `Morph#setPosition` saves the new and the old position). After having created the command, it is executed locally to ensure fast feedback to the user (optimistic approach). Then the command object is serialized and send to the server so that other clients get informed about the change. The server assigns an ascending id to the command and puts it at the end of the global command queue which is solely managed on server side. The id of the command defines the order of the command queue, and clients have to assure that commands are executed in the given order. Finally, the command is sent to all clients, who arranges it in their local command queue and execute it according to the command's id. The client

[2] http://etherpad.com/

which created the command in the first place doesn't execute the command a second time, but the assigned id have to be checked with the order in its local command queue. If other commands have been received in the meantime, the command has to be undone and wait its turn.

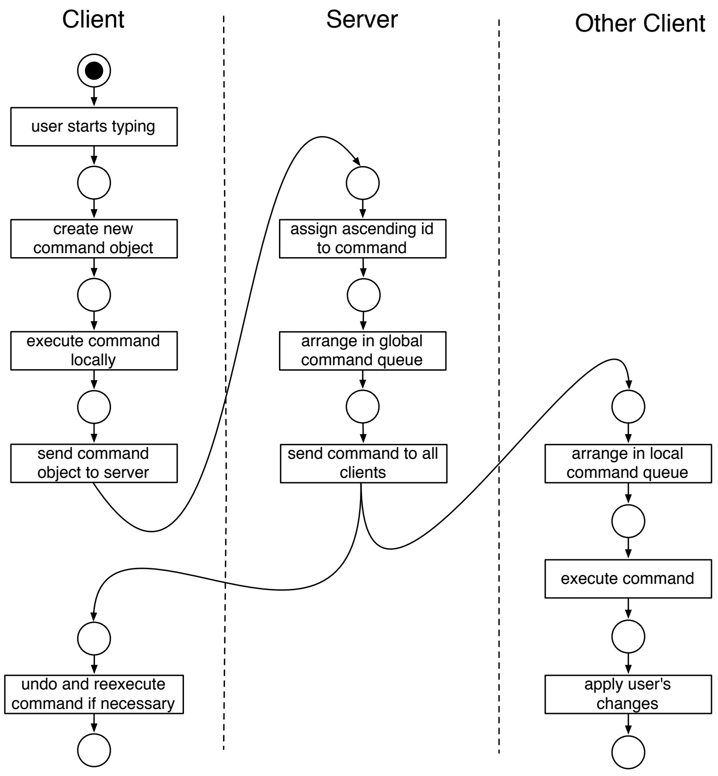
Client
Server
Other Client
user starts typing
create new command object
execute command locally
send command object to server
assign ascending id to command
arrange in global command queue
send command to all clients
arrange in local command queue
execute command
apply user's changes
undo and reexecute command if necessary

Fig. 1: Command life cycle using the example of an user who is typing text

During a collaboration session, all incidental commands are stored on the server side. As soon as a user presses Lively's built-in save button, a new milestone is created (which just references current command's id). When a new user joins an existing session, he receives a unique user id and a list of all commands that have been created since the last milestone. To implement a playback/ time-slider feature as mentioned in the introduction, a user could switch between milestones (and the accordant commands belonging to this milestones) and see all visual changes that have been made in the meantime.

3 Architecture

This chapter describes the architecture of the collaboration framework for the Lively kernel in detail. Figure 2 shows the overall architecture: A Lively application is embedded in the web environment and is therefore a client server architecture. On the client side, a `Command Manager` executes or undoes commands and queues new commands in the `Local Command Queue`. When the client creates a new command, it is serialized and send to the server as explained in section 2. On the server side, a web server is listening for new client connections and commands at which the latter are stored in the `Global Command Queue`.

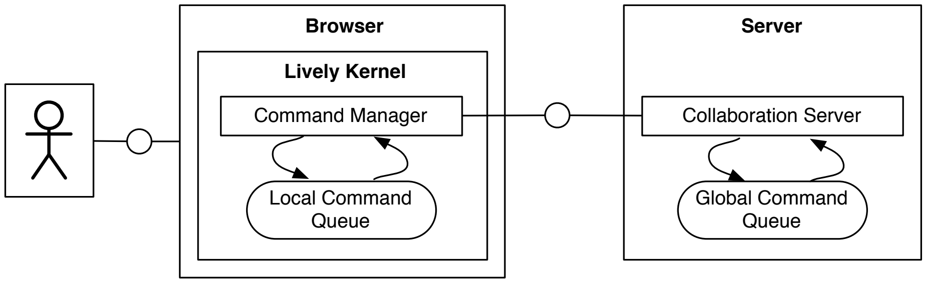

Fig. 2: High-level architecture of collaboration framework

Next subsections will explain the client and server architecture in more detail (see figure 3).

3.1 Client Architecture

The client intercepts some core Lively morph methods with *ContextJS* to record the user's (visual) changes (see table 1). These methods are grouped into three *ContextJS layers*, which can be separately switched on and off. Since some functionality (especially the `onMouseMove`) can produce much synchronization overload, this might be helpful in situations with slow internet connection.. The `Command Manager` queues the commands and sends them via the `Connection` singleton instance to the server.

The `Connection` singleton holds a persistent connection to the server which is established as soon as a client opens a collaboration-enabled Lively page. This connection is realized through the emerging WebSocket [3] technology, which allows bidirectional communication with a smaller footprint. Thus, pushing messages (in our case primarily commands) from server to clients and the other way round is easily possible without much overhead - it's much like using sockets in desktop programming.

Layer	Class	Method
`CollabMorphLayer`	`Morph`	`rotateBy`
		`translateBy`
		`setPosition`
		...
`CollabMouseMove`	`Morph`	`onMouseMove`
`CollabTextMorphLayer`	`TextMorph`	`setTextString`

Table 1: Overview of intercepted Lively methods

3.2 Server Architecture

The architecture of the server is shown on the right side of figure 3. The server is split into two main parts, the `Collaboration Server` acting as the web server and the `Redis Server` representing the database. The `Collaboration Server` is written in Node.js[3], an event-driven I/O framework that enables writing server-side JavaScript. Redis[4] is an in-memory key-value store that allows advanced data types as values (e.g. strings, lists, sets, ...) and can persist its data to the disk in configurable intervals. It perfectly fits in the collaboration use case since it provides fast access to the data while being semi-persistent so that most of the data can be restored when the server crashes.

The `Collaboration Server` accepts *WebSocket* connections and commands from clients. Every command is saved into the Redis database to enable the playback of commands and inform new clients about the stored commands. Table 2 gives an example snapshot of the Redis database to show which data is stored. Every key is scoped to the user's page URL so that commands of different pages aren't merged.

The `Collaboration Server` uses the *publish/ subscribe* mechanism (based on channels) of Redis to send new commands to every client. Three channels are exposed by the `Collaboration Server`: *commands*, *milestones* and *mouse*. All channels, except for the mouse cursor, are saved semi-persistent into the Redis key-value store. Mouse cursor events are directly propagated to all clients since they are only of interest in a short time slice. Thus, the Server subscribes to the *command* channel in Redis with a callback, which is called every time the server publishes a new command from a client. This callback function sends then the published command to every client.

[3] http://nodejs.org/
[4] http://code.google.com/p/Redis/

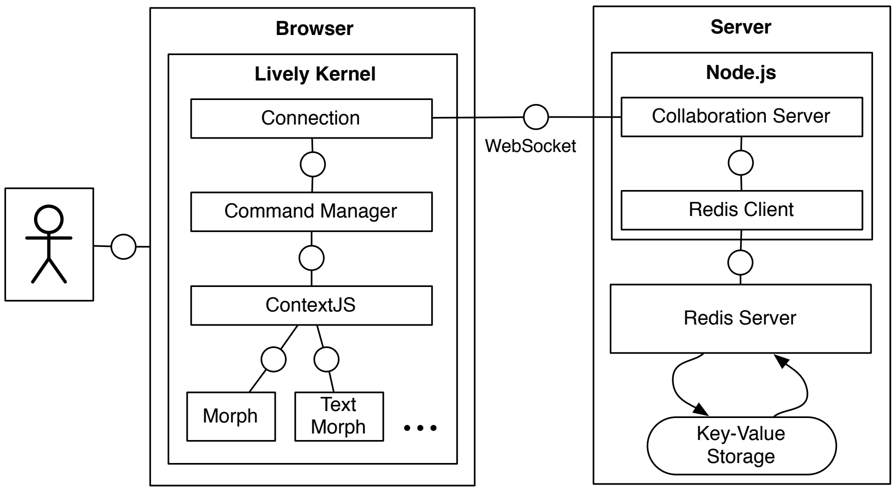

Fig. 3: Detailed client and server architecture of collaboration framework

4 Implementation Considerations

4.1 Globally unique IDs

When a user updates a morph on his page, the changes have to be sent to the accordant morph on the other users' pages. Therefore, globally unique IDs are assigned to all new morphs so that a morph can be unambiguously identified on each page. We extended the ID generation in Lively to avoid conflicts (i.e. same ID generated on different clients for different morphs) by scoping all IDs to the current user: Instead of generating IDs like `1234:Morph`, `lively.data.Wrapper#setId` add the user ID resulting in `user541234:Morph`[5].

4.2 Object (de)serialization

As described in section 3.1, we are intercepting relevant morph methods to listen to changes. Arguments are saved in command objects that have to be serialized when sending to the server. Especially when it comes to serializing whole morphs, i.e. for `Morph#addMorph` synchronization, a smarter serialization approach is necessary to handle circular references and already existing objects. [2] provides a Relaxer and Restorer class for (de)serialize Lively morphs that could be easily adopted for our use case: When adding a new morph (which might contain sub morphs, for example), all references to other morphs are only saved by their IDs so that a quite flat JSON structure is generated. Other clients can then deserialize this structure and restore all previous relationships.

[5] a colon as separator couldn't be used here since some existing code relied on the class name after the first colon

Key	Example Value
`<url>//milestones`	`["140", "234"]`
`<url>//commands-current-id`	`"336"`
`<url>//commands`	`[{ id:336", commandType:"SetPositionCommand", morphId:"user_0366079:Morph", val:{x:10, y:10}, oldVal:{x:5, y:5}, timestamp:"1280353890466", userId:"user_0" }]`
`<url>//users`	`["user_1", "user_2"]`

Table 2: Exemplary snapshot of Redis key-value database

4.3 WebSockets

Websockets are a fully bidirectional and slim protocol from the upcoming *HTML5* standard. The decision for *Websockets* as the protocol between the client and the server was taken, because of the need for nearly realtime bidirectional communication. There is a japanese website[6] on which everyone can test the speed difference between *Websockets* and XML HTTP requests (*Ajax*). With the latest *Google Chrome* browser the benchmark of the website shows that *WebSockets* are nearly 40 times faster than the *Ajax* requests.

5 Summary & Outlook

Some special events are intercepted by the use of *ContextJS*. These events are transformed into a command and send through *WebSockets* to the server, which saves the commands in the key/value store *Redis*. Via this mechanism nearly every Wiki page can be made collaborative.

At this stage of the project users can collaboratively work on a wikipage and add new morphs or drag them around, or even complex actions like adding new journal entries. All these actions are synchronized between the server and all users. Each user can even see the mouse cursors of the other users.

General Outlook In the future there will be a client side time slider, which enables each user to travel back in time, via undoing the saved commands. The user name next to each users mouse will be the login name and not an anonymously generated one. Near the time slider there can be a button or checkbox to easily switch the collaboration on and off. These two user interaction elements can be grouped to a preference pane. If one user saves the Wiki page, this could be

[6] `http://bloga.jp/ws/jq/wakachi/mecab/wakachi.html`

intercepted and every command saved on the server before this event can be deleted.

In the future there have to be also a feature to suppress the synchronization of some events. At the moment the menus from every user are synchronized to every other user, which is not the preferred behavior.

Conflict Management The conflict management could be extended to Operational Transformations, because in our solution it depends on the action taken by the users, which user will win. For example, if two users are positioning the same morph, the last command that arrives at the server will win. This is because the position of a morph is absolute and so the last value will be used. An example in which the first user wins is, if one user deletes a morph before another can change the size of that morph. The deletion will invalidate the resizing command of that morph.

Consecutive projects which will implement a better conflict management are welcome.

Clock synchronization Events which are not originated by a user action, like the tick of a `ClockMorph` are produced by every client nearly at the same time. These multiple events could also collide with each other, if the users are in different time zones. There are multiple solutions to solve these kind of conflicts, one solution is that only one client produces these special events. But the problem with different time zones is not considered with this solution. Not synchronizing these kind of events would be another result to this problem.

References

1. Etherpad time-slider. http://www.youtube.com/watch?v=Endvb81oz80 (2009)
2. Dannert, J.: WebCards - Entwurf und Implementierung eines kollaborativen, graphischen Web-Entwicklungssystems für Endanwender (2009)
3. Hickson, I.: The websocket api. W3C Working Draft 22 December 2009 (2010)
4. Nichols, D.A., Curtis, P., Dixon, M., Lamping, J.: High-latency, low-bandwidth windowing in the jupiter collaboration system. In: UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology (1995)
5. Wang, D., Mah, A., Lassen, S.: Google wave operational transformation. http://wave-protocol.googlecode.com/hg/whitepapers/operational-transform/operational-transform.html (2010)