BACHELOR'S THESIS

Design and Implementation of Shared Workspaces

Author:
Conrad CALMEZ

Supervisor:
Jens LINCKE

29 .06 .2012

Declaration of Autonomy

With these words, I assure, that this bachelor's thesis was written autonomously. No other resources except the specified were used and quotations were labeled explicitly.

Potsdam, 29 .06 .2012
Conrad Calmez

Design and Implementation of Shared Workspaces in a Mobile and Desktop Environment

Shared Workspaces for Lively Kernel

Conrad Calmez

Hasso-Plattner-Institut
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany

Supervision:
Software Architechture Group
Prof. Dr. Robert Hirschfeld, Jens Lincke

Project Partner:
SAP Research
Dan Ingalls, Marko Röder

conrad.calmez@student.hpi.uni-potsdam.de

Abstract. Synchronous, distributed collaboration systems simplify the process of cooperative work for groups of people not being at the same place together. This thesis describes the design and implementation of a system that supports synchronous, collaborative work in Lively Kernel. We show how we used optimistic as well as pessimistic synchronization approaches for diffent types of content. The system is implemented as a shared workspace in addition to a private work area. Further we illustrate what benefits synchronous work has over a asynchronous way of working together.

Keywords: Javascript, Lively Kernel, collaboration, CSCW, shared workspaces

Table of Contents

1 Introduction

Wiki systems offer an easy and accessible way for people to work together. Usually users can create and save pages that can afterwards be edited by other users of the wiki system. With such systems, work can be distributed over long distances. Lively Kernel [6] offers such functionality via two system built-in mechanisms.

First, the system is split into worlds which are part of the Morphic Framework [17] implementation that is a core component of the system. Those worlds form a so called Webwerkstatt [7] which collects the knowledge produced by the system's users just as other wiki systems do with pages.

Second, the PartsBin [8], as a way to publish written programs to the system, offers an identical benefit on the level of applications.

Working in wikis surely enables users to work together, but the style of collaboration is rather asynchronous since only one user can save the document at the same time. For the ability to work together at the same time[1], there need to be additional mechanisms.

As changes in a wiki can not be seen until someone saves them, duplication of work can happen if the work to be done is not pre-coordinated. But such coordination creates overhead on the process of working and disrupts the workflow of synchronous collaboration. However, the synchronization of content is an important task that has to be done in near real time to create a notion of synchronous work. Consequently, a system for synchronous work should either be on one location[2], so that synchronization does not have to happen, or it has to synchronize the content in a way that creates the least overhead on the actual process.

As Lively Kernel does not offer such functionality, we approached to implement such a system. This bachelor thesis describes how we augmented Lively Kernel's collaboration facilities by creating a new application that enables its users to collaborate at the same time no matter of what place they are. We compared our system with other collaboration systems in the collaboration matrix in figure 1[3]. Since synchronous collaboration is missing in Lively Kernel, most of the focus lays on even this style of collaboration. In general, the intention of our application is to support collaboration for people working on a common goal. That is why the users should not have to take care of the asynchronous collaboration style as well. Therefore, our system has asynchronous aspects as well.

When users come together to work on a common goal, that is called a session. Those sessions can be started if one user opens a new shared workspace within

[1] synchronous collaboration
[2] meant is actually one machine / computer
[3] The graphic is based upon the authors personal estimation and is not backed with measured numbers. It only serves for an approximated comparison of systems.

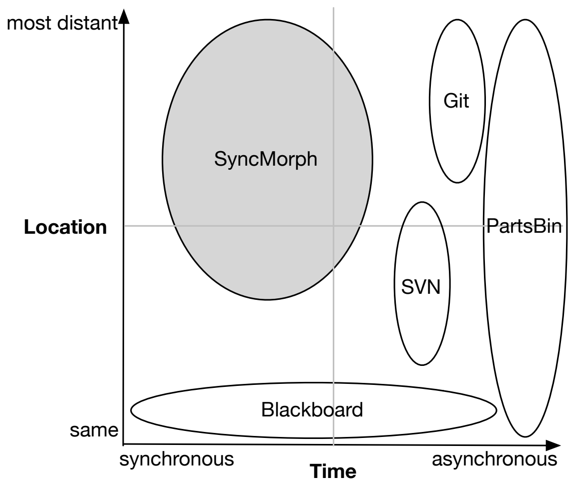


Fig. 1. Collaboration Matrix; The SyncMorph is the system of interest in this thesis. The PartsBin is part of Lively Kernel.

the system. In order not to have to wait for all participants to join a session, the system needs to support asynchronous collaboration styles. Users joining later need the content that has been produced at the time they joined, as well as the updates that happens after joining. As a consequence, our system does not only support synchronous, but also asynchronous collaboration styles.

Collaboration systems are very diverse in terms of how they use computers to support the work of a team. One natural approach is to physically share one space. This has the side effect that work can not be distributed over a distance. Since Lively Kernel is a web application that inherently has users all over the world, it should be possible to distribute work all over the world as well. Hence, an important key assumption of our system is that each user is working on his or her own machine. Conclusively, our system embraces distributed as well as co-located work.

As our system is based on the usage of individual computers it needs a synchronization mechanism. To synchronize the content, one needs to identify the content's benefit.

As the Morphic Framework is a core functionality in Lively Kernel it is understood that one wants to distribute morphs via our system that we called SyncMorph. To make the workflow easy, we decided to use interaction that a user of Lively Kernel is used to. In addition, our system is implemented as a morph. This suggests to use morph interaction. Thus, the users can drop a morph onto the SyncMorph and it will take care of the distribution to all other connected clients. Furthermore, if one user removes a morph from the shared workspace the corresponding morph representations on all other clients get removed.

Sharing content is a way to distribute knowledge, but if we want to enable users to work together, they need to be able to alter the shared contents. Consequently a more enhanced version of this SyncMorph also supports the editing of morphs on the shared space. In the interest of making changes to the synchronization state immediate and foreseeable it is also possible to see a morph as another user drags it onto the workspace of the SyncMorph.

Working with Morphic enables users create UI and behavior of their application. Nevertheless, sometimes an idea is not as concrete as it could be directly implemented. Therefore we decided to add drawing support to our collaboration system. The ability to draw enables users to sketch ideas. The drawings get synchronized as well so that ideas can be developed together. In order to create some freedom for the user while drawing, the system allows the customization[4] of pencils.

Having such a system, which synchronizes all contents to all clients, empowers the users to have a common sense of how far the progress of work is. However synchronization is not enough to create an awareness of what each collaborator is currently working on. In addition to the synchronization of contents, further mechanisms are needed for a system that should support synchronous collaboration. Collaboration on a physically shared workspace does not have those problems. Looking at why collaborators are aware of each other's work using such systems, can lead to a solution for a distributed system.

In a physically shared workspace like a large table, each team member is aware of the area in which the others are working by seeing them work there. The current working area in the digital sphere is where a user has its mouse pointer or finger[5]. By synchronizing the position of the mouse pointer or respectively the finger, an equivalent is found for distributed systems. Assigning each user a unique color, that is different enough from the others, enables the users to identify each of their collaborators.

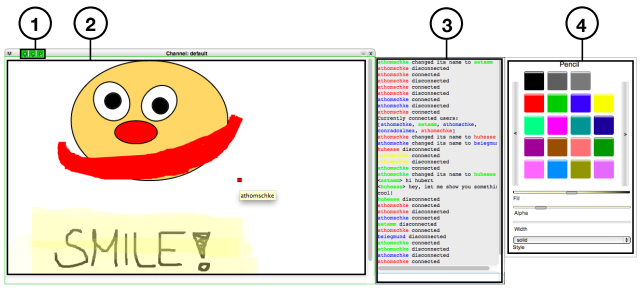
Besides, communication on a table is uncomplicated as one just says something out loud to pass the message to everyone. This kind of instant communication can be achieved via a group chat functionality that we implemented as well. Further our intstant messaging chat has the benefit that the message is only passed to everyone at the same table, but not in the same room.

The following chapter will describe our approach and the problems to solve. The succeeding chapter will explain the implementation of the system. The fourth chapter will evaluate the results by looking at some usage scenarios. Afterwards, the system will be evaluated performance wise. The sixth chapter will present related work. After that, we will give an outlook on how the system could develop. The last chapter will summarize the findings of this work.

[4] color, alpha, size, stroke style
[5] on a touch device

2 Approach / Problems

When people work together, they naturally gather at one place. Consequently, meetings in the virtual world need to evoke the same advantages. Especially this has to be done in order to escape the need of manual synchronization by the users. The term of the shared workspace in this thesis is meant as an analogy to the physically shared workspace as it is the model for the system that we built. An example of the user interface in action can be seen in figure 2. This chapter describes the problems that occurred while implementing our system. Additionally, it explains our approaches to solve the shown problems.



```
athomschke changed its name to sstamm
athomschke disconnected
athomschke connected
athomschke disconnected
athomschke connected
athomschke disconnected
athomschke connected
athomschke disconnected
athomschke connected
Currently connected users:
[athomschke, sstamm, athomschke,
conradcalmez, athomschke]
athomschke changed its name to huhesse
athomschke changed its name to bsiegmu
huhesse disconnected
athomschke connected
athomschke disconnected
athomschke connected
athomschke changed its name to huhesse
<sstamm> hi hubert
<huhesse> hey, let me show you somethi
cool!
huhesse disconnected
athomschke connected
athomschke disconnected
athomschke connected
sstamm disconnected
athomschke connected
bsiegmund disconnected
athomschke connected
athomschke disconnected
athomschke disconnected
athomschke connected
```

Fig. 2. User interface of the SyncMorph with (1) buttons to toggle and indicate connection state, chat pane and pencilstyler (2) synchronization pane, (3) chat pane and (4) pencil styler

In pursuance to make a system that supports distributed, synchronous collaboration work well it should be responsive. For responsiveness, it is crutial to synchronize the state or the content in a relatively small time interval.

A high-level overview of our applications functionality is shown in figure 3. It is based on message exchange to communicate modifications of the content. The figure shows an example of how content can be produced synchronously on two independent worlds.

2.1 Client-Server Architecture

We decided to implement the application with a client-server architecture. The decision was made for the following reasons:

First of all, the server as a central unit knows all clients. Having such an actor makes it easier to implement the distribution of messages. Althought, this does not say that a distributed networking system might not be equally well. Our solution was just the most suggesting one with the used technology.

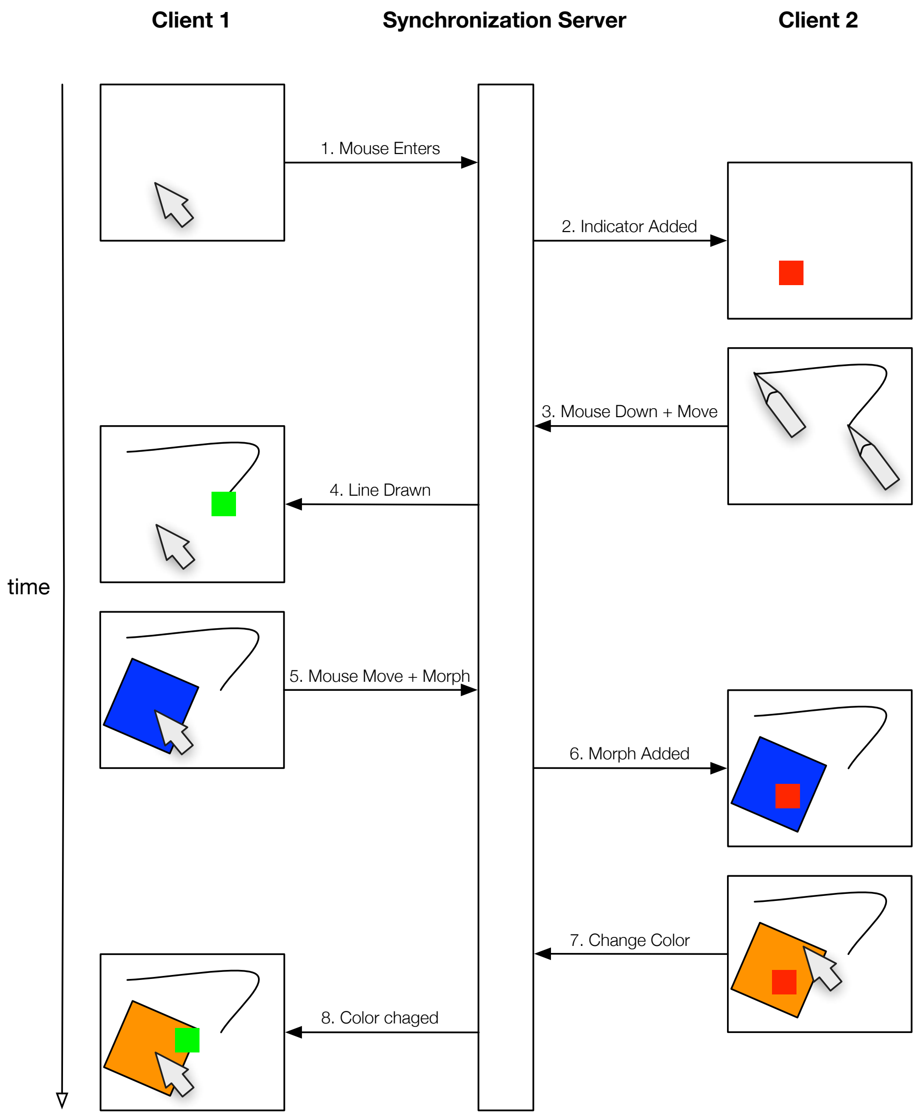


Fig. 3. High-level overview of functionality

Moreover, as the server receives all messages from all clients it is possible to store the input data into a persistent storage. At the moment, the persistence is realized by clients holding a certain state. This concept of persisting state in clients would of course work with a distributed system, but in this case one loses the central server as an additional archiving unit.

Beides, implementing a distributed system without central infrastructure might be convoluted in Javascript.

The client-server architecture also enables the implementation of alternative clients that communicate over the same server. Since the server's purpose is only the message distribution, and possibly persistence, there is no need to replace it with an alternative implementation. Alternative clients could for example implement a different drawing algorithm for the purpose of interoperability. Here again a set of polymorphic clients without a central server unit is possible as well.

Furthermore, the server can act as a centeralized mixer [5] for messages [9]. This can save bandwidth to increase the performance when using slower connections.

2.2 Data Exchange Format

We decided to define a specific exchange format for message exchange. With this format it should be possible for the user[6] to decide who will get the message that is going to be sent. In favor of applying[7] messages in the correct order it is also possible to augment the format in that way so that each message contains an ID and a timestamp.

The message packages itself are serialized JSON objects. The message format looks like the following in code example 1.1.

```
{
        message: "message content (does not have to be a string)",
        me: false,
        broadcast: true,
}
```

Code Example 1.1. simple version of the exchange messages

The message content can be string or a data object itself. The properties *me* and *broadcast* specify to whom the server should send the messages. Whereas *me* set to true means that the message will return to its original sender. Moreover *broadcast* means that the message will be sent to every user being connected to the same channel as the original sender of the message but the original sender itself. These options can be extended by a new property *broadcastType* which will determine on which level the broadcast will happen. This option should default to "channel" with which it will behave as described above. Setting the option to "global" would mean that the message should be sent to every

[6] in this case a client application
[7] or resending

client connected to the server except the client who initially sent the message. An extended version of this exchange format can be seen in the following example.

```
{
        message: "message content (does not have to be a string)",
        id: "unique message id",
        time: 1234567890,
        me: false,
        broadcastType: "global",
        broadcast: true,
}
```

Code Example 1.2. extended version of the exchange messages; properties id, time and boadcastType are added

2.3 Synchronization

As our system should support distributed, collaborative work, a synchronization mechanism is needed. This section examines the synchronization process. Figure 4 shows an overview of the event – method mapping for the different kinds of synchronization that are used within the application.

Scope of Synchronization Since work in Lively Kernel happens on so called worlds, it is suggesting to synchronize whole worlds. The disadvantage of this approach is that this would prevent the presence of private workspaces. Private workspaces are important for various reasons. Users might feel a disturbance of their private sphere if all their thoughts and work are synchronized immediately to all other team members [12]. Consequently, a good system has to offer the possibility to decide which content should be synchronized. That is why we decided to implement our system as an application that can be loaded into every world to augment it with the collaboration facilities our system is offering. All data and objects outside of the application then belong to the private workspace of the user. By dragging objects into the application they and all actions on them get synchronized. Therefore, we build a clear boundary between private and shared workspaces. Further, all work within the world remains the same just as the users are used to.

However the development of a clear synchronization boundary makes the application logic, and by that the implementation, more complex. Since our system is just another application within Lively Kernel it is part of a world as well, just like every other morph. The contents of the SyncMorph are created in one world. They can have references to other objects in the same world. If a user now decides to synchronize such an object having references to other objects in his or her own world the problem will arise that this reference is actually missing in all other worlds where the object was added by the synchronization algorithm. Naively, it is possible to implement a reference tracer in order to synchronize the references as well. The problem is that the world is a morph

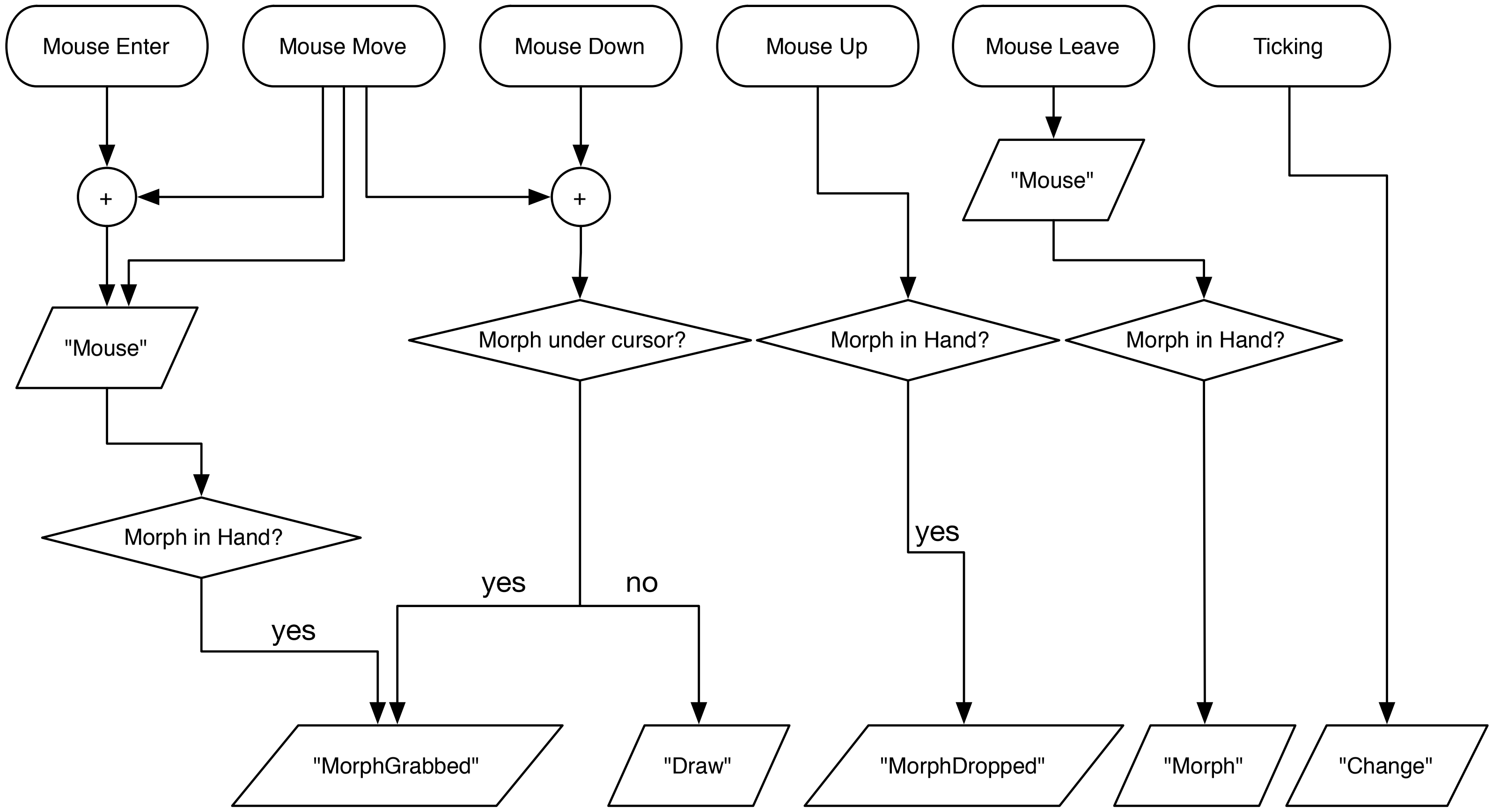


Fig. 4. event – synchronization method mapping

as well which can be referenced[8]. As a result, only one reference to the world would cancel out the concept of this synchronization boundary, because the world would have to be synchronized as well.

Synchronizing the world brings further problems. Since there is a world on every client, and there should only be one world, the algorithm would have to merge as many worlds as there are participants in one session. Moreover, the SyncMorph is part of the world as well ergo it would have to be synchronized as well. This would lead to an infinite recursion.

In conclusion, we decided not to synchronize references that point to objects which are not in the space of the shared workspace. The topic of references is part of the future work.

Synchronization of Generic Content For the synchronization of generic objects we needed a serialization algorithm. The first possibility is to implement an algorithm that uses a whitelist to create a serialized object with certain properties (e.g. the visual properties). On all other clients, an object of the same class would be created. The new properties would be applied to this object. Even if this approach needs less network bandwidth, the focus is placed on certain properties which makes the application harder to extend. To support additional properties, the serialization algorithm must be reworked.

Instead, the approach we took for serialization uses Lively Kernel's serialization algorithm to store worlds and morphs. The advantage is that the whole object is serialized, no matter what it looks like. This approach uses more bandwidth since more information are communicated. Initially an object gets serialized via this algorithm. Afterwards only changes [15] on the object get synchronized in order to save network bandwidth.

Generic objects and all changes on them are synchronized in an optimistic way which means that they are first executed on the client and then communicated. This has the benefit that the responsiveness of the system does not change by a considerable amount.

Synchronization of Specific Content For specific content with a limited benefit for the user we took a different approach. Drawings are an example for such content with limited benefit. Communicating whole objects would use more network performance than needed in this case. Restricting the sent information is acceptable, since the possiblities to extend the drawing of lines are endless[9]. We consider the benefit of saving bandwith here to be higher than the need of extensibility. The clients receiving such specific information recognize what has to be done by the kind of event that delivered them. By that our client application is a thick client [5]. With this, it is possible to create a responsive drawing surface.

Specific content is again optimistically synchronized to keep the corresponding interactions responsive.

[8] like any other morph

[9] We augmented the sent information by line style, width and color

Creation of Awareness Working together with a software that synchronizes its contents can sometimes be surprising as a remote user makes an action whose result is displayed on the user's client. To reduce the effect of being surprised, the user actually has to see the action happening on the remote client. Besides, the user should be able to anticipate what action a collaborator will do. This concept is called awareness. If a user is not surprised by what happens on the synchronized board he or she will be able to plan his or her own work in a better way. Good planning of work by each user also leads to less editing conflicts as well as less duplicate work. Consequently, awareness is an important concept in collaboration systems [2,3]. We implemented two concepts to create awareness.

Telepointers Looking at how users anticipate what and where a co-worker is doing something when working at a physically shared work space[10], we found a solution for the anticipation problem. When a user sees another one approaching a certain object on the work space he or she will think that it is likely that his or her co-worker is going to interact with it[11]. Furthermore in a physically shared workspace users are able to point on objects to talk about them.

Both action are performed with the hands of the user. Consequently, it is suggesting to synchronize the representation of the hand in a distributed workspace. By doing so, a user gets a telepointer for each collaborator.

For the sake of being able to distinguish different users, each telepointer has a unique color that is different enough from all colors of the other collaborators. Since we have a synchronization server as a central communication unit that knows all clients we implemented the user – color mapping there. When a new user logs into a shared workspace the server will assign a new color to this user.

Telepointers are, like drawings, specific content that is optimistically synchronized with a reduced set of information since the mouse and touch interaction needs to be highly responsive with respect to a usable system.

Chat With the given system, it is straight forward to design a chat application. We use our data exchange format to send textual messages between users. To enable the user to see if the message was really sent, the messages are pessimisticly synchronized. This means that a message is first sent to the server which distributes it to all clients. Each client executes the action corresponding to the type of message.[12] So if the connection is broken, the user will recognize it by seeing that his or her chat message does not show up in his or her chat client. The server itself stores chat messages not only by saving the user name - message mapping. It extends the information by the time the message was received by the server. With that, a chat log is created on the server that can be replayed to clients that have been offline.

[10] such as a table or a whiteboard

[11] e.g. draw on the area; manipulate the object; move the object

[12] In case of a chat message this would be to display the message and the username in the chat GUI.

3 Implementation

In this chapter, the implementation of the system is illustrated. The implementation is described feature by feature. For each feature we discuss the benefits it conducts and the limitations that remain. The features can be divided in three main categories of features: synchronization of morphs, synchronization of drawings and awareness features. Each feature set is part of the same software system.

3.1 Synchronization of Morphs

As morphs are the objects with which the users implement their applications, it is important to use the synchronization approach of generic content in order not to restrict the possibilities Lively Kernel is offering.

Fast Sharing At the time we implemented the first version of the SyncMorph, we routinely used CouchDB [1] for storing data that we created in Lively Kernel. Consequently, we used CouchDB here as well to store serialized morphs. The serialization was already implemented and we just had to use it. It basically linearizes the object tree and writes it into a JSON string.

On top of serialization, we needed some interaction event on that we would start the synchronization process. Here again we were looking for something that Lively Kernel was already offering. Since our application was implemented as a morph, we wanted it to be as simple as adding another morph to it. After a morph is added our application should take care of its distribution. The *onDropOn* method seemed to be a good place to hook into since it is called on the target morph when a user drops a morph onto it. We patched the behavior to the class *Morph* with a layer. Code example 1.3 shows this. If a morph is dropped onto the SyncMorph, the method *saveMorph* will be called on the SyncMorph. For convenience, we wrapped the usage of the built-in serializer in a *serialize* method on the class *Morph*.

The functionality of saving a dropped morph to the database is of course implemented on the SyncMorph. Apart from saving the morph to the database, metadata will be added to the morph object in order to be able to manage[13] the content as updates come in.

After this iteration, we were able to distribute morphs fast between worlds without using the PartsBin or a saved page. Though the conflict avoidance strategy was to disallow interaction except grabbing for the morphs on the SyncMorph. So for being able to edit the morph, the user had to grab it and drop it into his or her own world to enable the editing features again. As a consequence synchronous collaboration was not possible with this revision, but we fastened up the sharing process.

[13] e.g. delete the morph if it was deleted on another client

```
module('projects.BP2012.SyncMorph').requires().toRun(function() {
  cop.create('SyncMorph').refineClass(lively.morphic.Morph, {
    onDropOn: function (aMorph) {
      if (aMorph.saveMorph) {
        this.disableHalos();
        aMorph.saveMorph(this);
      } else {
        this.enableHalos();
      }
    },
    serialize: function () {
      // ... implementation intentionally left out
      return serializedObject;
    },
  });
  SyncMorph.beGlobal();
})
```

Code Example 1.3. extension of class Morph

Message Passing Since the CouchDB server was quite slow if the SyncMorph contained many morphs, we were looking for a faster alternative. We decided to do the message passing ourselves and implement a synchronization server. We used node.js [16] for that. Fortunately, Lively Kernel is offering a mechanism to create and run node.js servers. This made it easy to develop this application completely within Lively Kernel.

The implementation of the synchronization server was straightforward since node.js and socket.io [10] offer all features we need for a message passing server. First we need to hold a reference to all clients that are connected to the server. Since node.js abstracts different technologies to sockets, it was easy to accept incoming connections and close them again as the client disconnects.

The channel feature of socket.io was a way to implement different workspaces within one system. Also, it was easy to send a message to all clients of a certain channel, as the library offers a broadcast mechanism that sends a message to all clients[14] of a channel or the whole system. Further, we needed to define different message types for different actions that should be synchronized. Figure 1.5 shows an exerpt of the implementation of the synchronization server. We wrapped storage and message broadcasting functionality in an object called *WhiteboardServer*. The set of events that we need for the communication between clients was defined with the *io* object of socket.io. Lines $16 - 18$ show an example of such an event definition. The first parameter of the *socket.on* call is the name of the event. Additionally, the second parameter is the functionality description of the event as a Javascript function.

[14] except the sender

```
function saveMorph(aMorph) {
  aMorph.databaseID = undefined;
  aMorph.databaseRev = undefined;
  if (this.active) {
    var newMorph = aMorph.serialize();
    var result = this.getDB().save(newMorph);
    if(result.error=="conflict"){
      alert("an error occured while synching the morph");
    } else {
      aMorph.databaseID = result.id;
      aMorph.databaseRev = result.rev;
    }
    this.updateDBObjectIDs();
  }
}
```

Code Example 1.4. save functionality of SyncMorph

The version of our application after this iteration improved the performance of the application when it contains many morphs. Still, the synchronous collaboration features were not present at this time.

Enabling Collaboration Finally we decided to integrate a diffing and merging algorithm for objects [15] in order to be able to only communicate changes instead of whole morphs. Unfortunately an event for a change that happened on an object is missing in Javascript. That is why we used a ticking script that observes the changes on all morphs[15] that are synchronized via the SyncMorph. This ticking script is executed in a well defined time interval. It performs a diff to the current version of a morph. To have a reference to compare to, we copy the morph after diffing as a current version.

When a change is detected, it is sent to the synchronization server that distributes that message to all other connected clients. A client receiving a change merges that change into the regarding morph.

After this iteration the SyncMorph is finally capable to support synchronous work. The users will get updates of the morph shortly after they are done on the client of a collaborator.

3.2 Synchronization of Drawings

Drawings are an example for content that has a well defined set of features and which therefore can be synchronized as specific content in order to save network bandwith and by that improve the responsiveness of the system.

[15] Morphs are usual Javascript objects as well.

```
WhiteboardServer = {
  port: 4000,
  // ...
  send: function (socket, channel, messageType, data) {
    if (data.broadcast) {
      socket.broadcast.to(channel).emit(messageType, data);
    }
    if (data.me) {
      socket.emit(messageType, data);
    }
  },
}

io.sockets.on('connection', function (socket) {
  // ... code intentionally left out
  socket.on('ping', function (data) {
    WhiteboardServer.send(socket, /*channel*/, 'pong', data);
  });
  // ... code intentionally left out
});
```

Code Example 1.5. code exerpt from synchornization server implementation

Drawing on Canvases The drawing facility needs to have a surface on which the users can draw. We used the HTML5 canvas element for that purpose. The canvas element offers an API to draw on it. Since Lively Kernel did not include a canvas morph at the time we implemented this, we needed to create such a morph. Fortunately, Lively Kernel is offering a flexible mechanism to create morphs. The only task we had was to create a canvas element that we used as the shape for the morph.

For the ablity to send events to the synchronization server, we encapsulated the API calls in own methods on the new morph that we called Whiteboard. Conveniently, the API functionality of the canvas element matches our drawing metaphor as it has a function *lineTo* that acts as you would have a pencil. You can call it several times giving it a point on the canvas in order to draw a line. The last drawn point is the point where the next stroke would start as if there would be a virtual pencil. Aside from that function *moveTo* that sets the position of the virtual pencil to a given point on the canvas.

In addition, we defined which interaction event[16] calls which API call.

First a client application sent out whole lines. This had the consequence that lines popped up at remote clients. When drawing larger shapes this actually spoiled the process of drawing together as the collaborators can not anticipate where another user is drawing. To solve that, we implemented that each stroke on one line was synchronized. Consequently, if a client receives a message from

[16] mouse down, mouse move, mouse up and touch start, touch move, touch end

the synchronization server, it will call the assotiated method on itself to draw the stroke.

With this implementation the users are able to make drawings together. Unfortunately, the canvas does not seem to be fast enough when synchronizing single strokes of a line. More importantly, this implementation does not work on the iPad. That is why we decided to use a different technology for the drawing surface.

Drawing Lines Lively Kernel supports SVG rendering[17] which was the reason why we decided to use SVG paths for the drawings.

As SVG does not have a convenient API for drawings such as HTML5's canvas element, we implemented a workaround in order to map the code to the drawing metaphor. When an event arrives, that starts the drawing of a new line[18], a new SVG line morph is created. With each arriving event[19] that indicates the continuation of the drawing, the current position of the mouse or finger, which is stored in the regarding event, is added to the vertices of the SVG line.

Additionally, we abstracted the call of drawing functions and the regarding events that should trigger them. For example, we mapped the mouse move event to a method that handles the process of drawing a new stroke. When we added touch interaction we only had to map the touch move event to the same method.

Actually drawing at the same time on different clients requires that each message contains a point[20] and an identifier for a line. By providing these information, the client application is capable of drawing multiple lines at the same time.

This version of the Whiteboard offers a way to draw together on a morph with a better performance than the previous iteration.

Additional Features To better support the expressiveness of the drawings, we implemented a virtual pencil that could be styled in different ways. We implemented the customization of line thickness, color and line style. With that users have a versatile tool at hand for different drawing tasks.

To implement this additional feature, we only had to set the width and color as well as the style of the border of the SVG path. Moreover, we built a GUI tool to set all those parameters in a convenient way. This GUI can be seen in figure 2 on the right.

Finally after this iteration we have a software that supports the collaborative drawing on desktop and touch devices. Likewise, simultaneous drawing on different clients and with a customizable pencil style is possible.

[17] SVG rendering was actually the default rendering mechanism before HTML rendering was implemented

[18] mouse down + mouse move or touch start

[19] mouse move or touch move

[20] that should be added to the vertices array of the SVG line

3.3 Awareness Features

Telepointers Javascript does contain events for mouse interaction and Lively Kernel does contain events for touch interaction as well, which is an enhancement we made that is described elsewhere [11]. Looking at the move events for each interaction method one gets the current position of the representation of the hand in the system. Since telepointers are specific content, they are synchronized optimistically with a limited set of information. The mouse event that is distributed by the server to its connected clients contains the position and identification information for the telepointer. An example of how a mouse message event looks like can be seen in code example 1.6.

```
{
  message: {
    indicator: 120938479283,
    position: {x: 42, y: 23}
  },
  me: false,
  broadcast: true,
}
```

Code Example 1.6. an example of a mouse message event

The remote clients create and display a telepointer for each other client. With the identifier that is provided in the mouse message, the corresponding indicator can be found. With each arriving event the remote clients update the position of the corresponding indicator.

Chat The implementation of a chat system was straightforward, since there is already synchronization server that can distribute messages, and that is already able to communicate complex information. On the server side a new message type had to be added that does nothing but simple routing of messages to the clients.

On the client side, the sending and displaying of messages had to be implemented. For that, we build a separate GUI pane that users can trigger from the main view.[21] The chat pane consists of two elements for its two tasks: First, an input field where users can type in text and send it by pressing the return key. To have this behavior implemented, we watched on each key stroke[22] if the return key was pressed. Second, a pane to display incoming messages. This message log was realized by taking the data that is coming in and concatenating the message's content to the current content of the text pane.

[21] This can be seen in figure 2
[22] onKeyDown

In order to support shortcut commands for power users, we added commands that can be entered in the same input field. Every command begins with a slash that is followed by the name of the command. The commands are stored in an object that has the command name as a key and the description of the functionality of the command as a value. The key is a string and the functionality description is a Javascript function. An exerpt of the definition of this object can be seen in code example 1.7.

```
this.commands = {
  'nick': function (name) {
    this.whiteboard.setUserName(name);
    this.showMessage("changed nick to " + name);
  },
  'names': function () {
    this.whiteboard.getConnectedUserNames();
  },
  'channel': function (channel) {
    this.whiteboard.setChannel(channel);
  },
  'chan': this.commands['channel'],
  'clear': function () {
    this.whiteboard.clear();
  },
  /*
    ... more commands intentionally left out
  */
};
```

Code Example 1.7. exerpt of the command object

Having this object available, the functionality gets called by accessing the commands object with the command string the user had just entered as a key and calling the *apply* method on the function that was returned. The function will be applied to the chat pane so that the description of a command must be written with that in mind. All parameters given by the user will be routed to the function that is called on the chat pane.

Furthermore, the convenience feature of having a history of the entered text was implemented by again watching the pressed keys. If the up or down keys were pressed, the user will be able to cycle through an array of the entered messages. Those messages were saved by pressing enter[23].

[23] The same mechanism as sending messages

4 Examples and Scenarios

This chapter points out some scenarios of usage that will be discussed on the basis of the current version of the collaboration system we implemented.

4.1 Exchange of Content Between Worlds

Let us assume the following collaborative scenario: Two users working in Lively Kernel on two different worlds want to exchange ideas in form of an application written within the system. Having the PartsBin, the creator of the application can publish it. The other user can from now on load it using the PartsBin.

This process invokes several problems when working together closely. First, the overhead of publishing the application and writing a commit message might be too high to justify the benefit of sharing the work. Second, the user that shall have a look at the work of his or her coworker might not want to invest the time to search the application in the PartsBin in order to load it. Besides, the progress made on the implementation might not be in a state where one wants to publish it to a broad audience[24].

Consequently, a fast exchange of applications[25] should have the least overhead to share the application with others.

The SyncMorph implements this workflow by letting the user simply drop the morph into the synchronization pane. The system will take care of the synchronization. As an effect, the morph appears in the coworker's application without the need that this person does anything except being connected to the server.

4.2 Developing Ideas Together

Working with a system that follows the metaphor of a wiki, a user will create content on its own and save the content page in the interest of making it available to other users of the wiki system. The collaborators can open the saved page to see what the user created and add their own ideas to it.

This kind of conflict handling is called "Single Active Participant" [4]. This process is relatively slow since every participant has to wait until someone saves the page to add own content to it. If the style of work is highly asynchronous, this will not be a problem. But as the system should support synchronous work, this approach does not fit the requirements of simulaneous editing. Figure 5 shows the difference of asynchronous and synchronous styles of working together.

Consequently, conflict handling in our application is not done in such a blocking manner. Our approach is aiming at group dynamics to solve editing conflicts by giving each user the information he or she needs to know where

[24] This is what the PartsBin actually does.
[25] generally Morphs which are Javascript objects

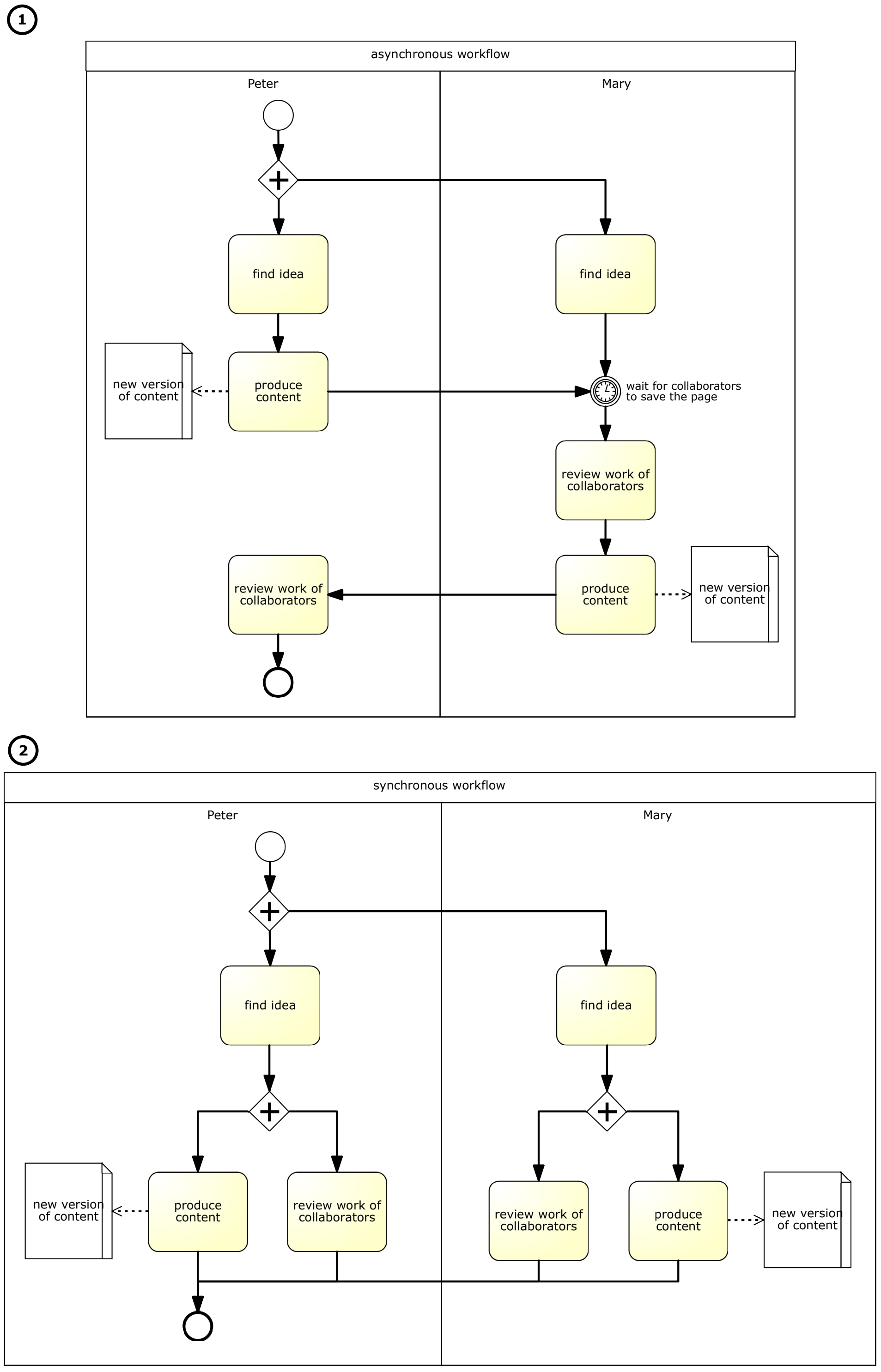


Fig. 5. Examplary comparison of (1) asynchronous and (2) synchronous collaboration style

and what his or her coworkers are doing. Furthermore, communication is important to create group dynamics. That is why we implemented an instant chat that is located next to the synchronization pane. Besides, the actual editing happens in a fully synchronized way. If a person makes an update this change will be sent to every other client that is connected. With such a level of synchronization, we create the feeling that the group is working on the same content.

4.3 Communication Within the System

Communication in wiki systems is often done via comments that are just another variation of content of a wiki page. Consequently, this comes along the same problems as other content[26] when working synchronously.

In favor of a synchronous working style, the system should distribute those messages instantly. As mentioned in the previous section, the SyncMorph implements this instant messaging with a chat interface that distributes messages to all users of a channel.

[26] see previous sections

5 Performance Evaluation

This chapter deals with the evaluation of the performance of the system. Since synchronization of contents is time critical for synchonous collaboration, the focus lies on how fast messages are exchanged between clients using differnt networking technologies. The experiments were all done using the same laptop computer. This machine was connected over WiFi to an access point that was connected to the internet using one of the following technologies: local network[27], DSL, UMTS-Broadband, GSM. Further, the experiments were realized using two clients connected to the server. Both clients ran on the same machine and used the same networking technology.

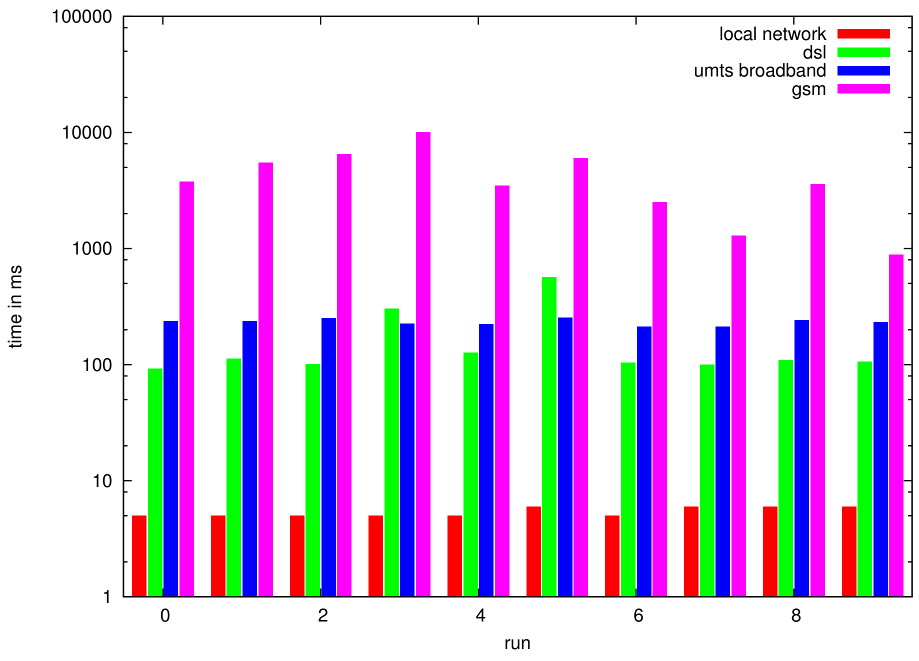


Fig. 6. Roundtrip time of a simple ping message through the synchronization server

For the sake of being able to interpret the results better, we first measured how long it takes for a simple message to get from one client to the other and back. For this roundtrip, we sent a *ping* message through the synchronization server to the other client. The other client then responded with a pong message. As the pong arrived at the original sender the time measurement was over. Figure 6 shows the results of the measurement.

[27] refers to the same network the application server is in

The different technologies performed as expected. The roundtrip time in the local network was 5.4 ms in average. Consequently, a message arrived at the other client after approximately 2.7 ms. The other technologies performed worse as expected. For the DSL internet connection, the roundtrip time was 172.4 ms in average which means that a message was received by the other client after approximately 86.2 ms. UMTS-Broadband[28] was not that much slower than DSL with 232.7 ms in average for a roundtrip meaning that after approximately 116.35 ms the other client received the message. Using GSM as an internet connection slowed the roundtrips of messages down to 4346.5 ms in average. Consequently, a message arrives at the other client after more than 2 seconds.

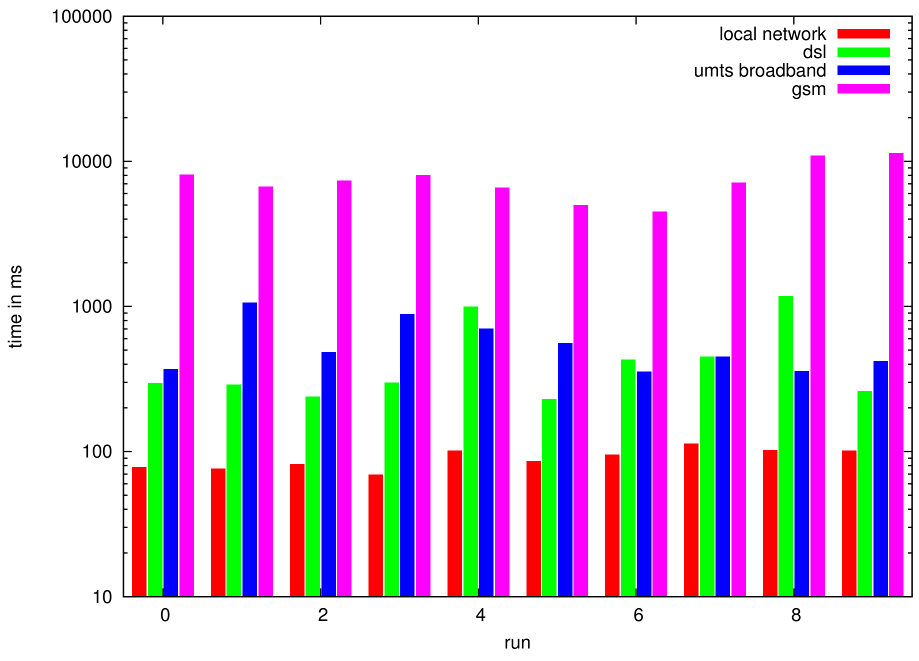


Fig. 7. Roundtrip time of a message with a whole morph as content

For messages with a considerable amount of content the throughput of the network connection is important. Using the system ourselves we figured that a synchronization time around 500 ms seems to be enough not to break the synchronous workflow.

[28] also known as 3G or HSDPA = High Speed Data Packet Access

Adding content to the messages has an effect on the timings of the messages. The roundtrips in average were: 90.3 ms for the local network, which results in a time of 45.15 ms that the other user needs to wait for message arrival; 466.7 ms for the DSL connection, so that messages arrive at their destination after approximately 233.35 ms; 564.6 ms for the UMTS-Broadband connection, so that messages arrive after approximately 282.3 ms; 7569.3 ms for the GSM connection, meaning that messages reach their goal after approximately 3784.65 ms.

Conclusively, the throughput on connections using the local network, DSL or UMTS-Broadband is good enough to use the system. Whereas, connections using GSM are definitly too slim to use them for synchronous collaboration.

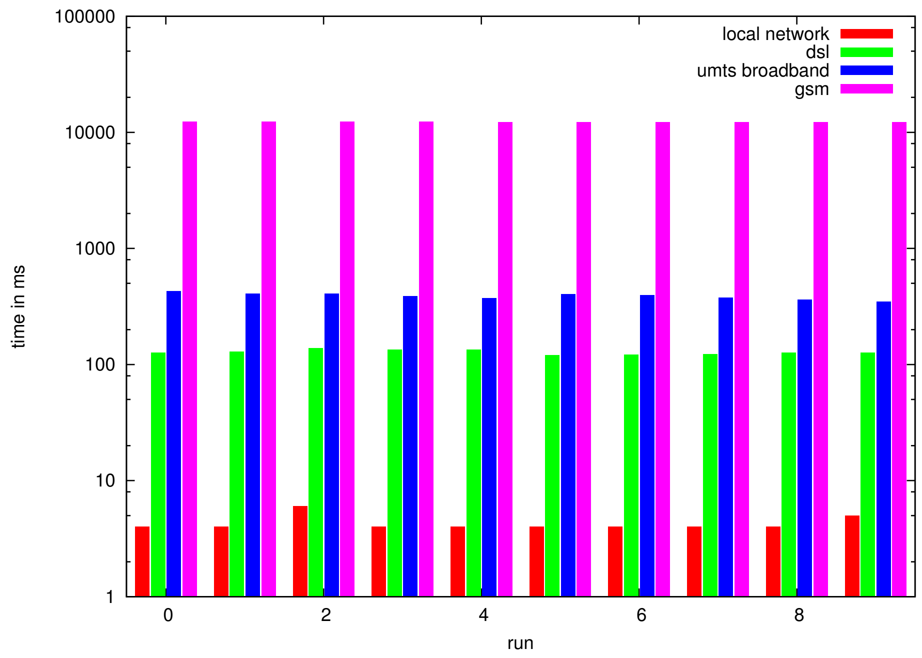


Fig. 8. Roundtrip time of a mouse message

There are also messages that show a continuous action like the mouse moves on a client that are represented by telepointers on the other clients. For those messages, the throughput of the network connection is not important but speed of the network[29] Those message are not as large as a message with a morph as content since they basically only contain a point that says where the mouse

[29] How fast messages arrive at the destination.

pointer moved. But since the action that is shown is continuous and the users expect it to be continuous, the time it takes to synchronize should be low.

On the local network it takes 4.3 ms rountrip time in average. So that the message arrives approximately after 2.15 ms at the other client. This is as much time as it needs to send a ping message. On the DSL connection it takes 128.3 ms for one roundtrip in average. So that a mouse message will arrive after approximately 64.15 ms on the other clients. The UMTS-Broadband connection reaches 389.2 ms in average for a roundtrip. So that the message will approximately arrive 194.6 ms after sending it. On the GSM connection a roundtrip takes 12267.2 ms which means that the message will arrive 6133.6 ms after sending it at the destination. Again, as we used the system ourselves, we figure that a time around 200 ms is enough to create a reasonable smooth continuous action on remote clients.

Conclusively, connections using the local network, DSL or UMTS-Broadband are fast enough to use them with our system. But connections using GSM are definitly too slow.

Another question is if the system is usable and responsive enough to collaborate with people from all around the world. As an example we assume a work group that is in Potsdam[30] and Palo Alto. The distance between the two cities is about 9154 km[31]. The fastest connection between the two cities would be a roundtrip time around $2 \times 9154\, km/speed\, of\, light = 2 \times 9154\, km/299.792\, km/ms \approx 60\, ms$. The people working in Potsdam should not have a problem as they might use the local network. But the question is if the system is usable and by that fast enough for the people in Palo Alto.

The actual roundtrip time would not be as low as 60 ms since the information has to travel with the speed of light. Additionally, we can not assume the same network performance as in the local network. Consequently, we will assume that the network connection performs as well as a DSL connection. Adding the additional latency to the values of the DSL connection results in times of 116.2 ms[32] for a small message and 263.35 ms[33] for a larger message to arrive. Since this estimation is based on the speed of light and the direct connection between the two cities it is very optimistic. The latency in reality might be higher. Conclusively, the user experience might not be good enough that far away from the server.

[30] Where the server of lively-kernel.org is located.

[31] Queried at http://www.wolframalpha.com/input/?i=distance+from+potsdam+to+palo+alto

[32] $172.4\, ms + 60\, ms/2$

[33] $466.7\, ms + 60\, ms/2$

6 Related Work

In this chapter we shortly point out some related work that has been done by other research groups. The work discussed here was done in the 1980s and 1990s.

6.1 WYSIWIS[34]

In 1987 a group of researchers at XEROX Palo Alto Research Center worked on a collaboration system for meetings [12]. For that, they took a closer look at the WYSIWIS principle that in their opinion supports many features an analog system such as a whiteboard[35] has. But they also hold that WYSIWIS interpreted strictly is too inflexible. This work is an example of how to handle problems evoked by the strict interpretation of the principle. It gives as well ideas for a collaborative drawing system.

6.2 Cognoter

The same group who did the research on the WYISIWIS principle developed an application named Cognoter that serves as a support system for collaborative organization of ideas [13]. They point out the benefits of working in a group. Further, they have a closer look on how to support the process of finding ideas[36], as well as organizing and evaluating them.

6.3 GROVE[37]

In 1989 a group of researchers at the Microelectronics and Computer Technology Corporation in Austin, Texas worked on a collaborative outline editor named GROVE [4]. In their findings they describe the difference between shared and private workspaces. Along with that they developed synchronization boundaries for their system. The work also contains a discussion about WYSIWIS. Concurrency control in synchronous collaboration systems is also discussed by taking different approaches like locking of content and operational transformations[38] into account.

6.4 Portholes

In 1992 researchers at Xerox PARC[39] developed a system to increase awareness of distributed work groups called Portholes [3]. The basic idea is that each collaborator has a camera and a microphone installed at his or her work place. The

[34] What You See Is What I See
[35] Their metaphor is a chalkboard.
[36] Brainstorming
[37] GRoup Outline Viewing Editor
[38] Optimistic approach to synchronization where an operation is executed immediately with the possiblity to undo it later.
[39] Locations in Cambridge, Great Britan and Palo Alto, CA, USA

system shows the images of the whole group in one view. Consequently one can see who is actually working at a given time. With the microphones short audio messages could be recorded. The system they developed made it possible to be aware of ones collaborators without much need of information gathering. In their work they also describe the architecture of the system which consists of serveral servers for the different locations. Those servers are responsible for the image processing of their connected clients. In addition, the servers synchronize to the servers of the other locations so that a client only has to query one server to get the information being stored by the whole system.

6.5 ShrEdit

In 1992 researchers at Xerox PARC in Cambridge did research on a collaborative text editor as well [2]. In their findings they compare different approaches made by other research teams. Moreover, their approach focuses on the concept of shared feedback. The main statement of the group is that giving a group enough information on what is happening on the shared workspace and enabling them to communicate informally is better than predefined roles[40].

6.6 TeamWorkStation

In 1990 a group of researchers at NTT Human Interface Laboratories developed an interesting approach to integrate virtual and actual workspaces [9]. The basic concept is to make video overlays of different workspaces. This creates a high acceptence since each user can use the tools he or she is most comfortable with. Obviously a problem is to merge the work a group has done into one artifact. They integrated microphone and camera as well to make communication between collaborators as easy as they would sit next to each other.

6.7 Single Display Groupware

In 1999 a group of researchers at the University of Maryland developed a collaboartion system that has its focus on local collaboration on one computer [14]. The challenge of this work was to design a system that offers individial input possibilities for each collaborator. At this time multitouch enabled devices where not present and the guiding input principle was one mouse and one keyboard on one machine. This work offers an insight on how collaboration support works on a level of most narrow cooperation.

[40] which would mean additional management overhead for the collaborative work session

7 Future Work

As this work is only a part of a project that took place at Hasso-Plattner-Institut in 2012 there are some ideas that not have been implemented. This chapter will suggest some of the work that might be done in order to improve our system.

7.1 Software Quality

As the system served as a platform to experiment and try out different approaches, the software quality is not very high. In order to make the system ready for productive use, there should be some work done to increase the quality of the software. To achieve that, the used algorithms and architecture should be much more robust since it seems to fail unexpectedly when using it. Note that the amount of tests is rather small. To ensure that the written software does what we expect, there should be at least a unit test suite that tests the functionality of the system.

Furthermore, to ensure a good extensibility and understanding of the system there should be additional documentation. Especially the architecture and functionality should be documented well for creation of better understanding.

7.2 Client Features

As seen in the chapters above, the client application has some features that support synchronous or asynchronous collaboration style. In order to create more possibilities of working styles and improve the collaborative work we have some features in mind that the client could support.

Chat System The chat is a way to communicate within the system. However, the text messages that are sent and displayed to all participants of a working session form a single stream. As there might be groups of people within the workgroup working focused on special features, a stream for the whole group might not work out well. Since different threads in a single stream are difficult to read, an additional information to the chat message can help to split the stream up into different topics. This additional information could be a position within the shared workspace. We imagine a chat system which we called ObjectTalk in which conversations happen around the object of interest. By doing so, everyone who is interested in this coversation can scroll the view to the object where the chat message are displayed as well.

Drawing The drawing of sketches works quite well in the current version of the system. But an opposite operation is missing in the system. There is a way to clear the whiteboard by deleting all contents, but when drawing users also want to erase single strokes that are wrong. For deletion, we imagine to different kinds of erasers: The first approach is an eraser that works just as a eraser on paper. Usability wise, it is just a special pencil that deletes content instead of creating it. The second approach is a tool to delete whole lines. The basic idea is that the user draws a line across all lines that he or she wants to delete. With that approach, much content could be deleted with one action by also being able to select what should be deleted.

Change Detection The detection of changes on properties[41] which are part of the Morphic Framework do not have to be discovered by diffing the objects over and over again. For that pupose the Morphic's getter and setter functions can be utilized. In addition, other than properties, the addition of scripts to an object can be detected via the *addScript* function of a morph. The need of diffing and merging of objects does not cancel out since the adding and changing of arbitrary properties of the object must be detected as well.

Having this utilization of Morphic's getter and setter functions, it is possible to keep a log of the changes that have been made. This log allows to implement an undo-redo feature for the changes on Morphic properties.

Conflict Handling Currently, the handling of conflicts is not actively done by the application. Our approach is that the group will coordinate itself to avoid such editing conflict. Since human communication is not free of misunderstandings, the system should somehow handle editing conflicts. A possible solution would be to accept the most current change that happened on an object. For example, if two users change the size of a morph in an overlapping way, the change that happened lastly will be applied. Doing so the optimistic synchronization approach can still be used. The client which made the change that is going to be accepted will discard the update it is receiving from the other client because it's change is more recent. The other client whose change is discarded will apply the change it is receiving to its synchronization state.

Connections The current implementation does not synchronize connections since the synchronization boundary does not allow to synchronize objects that were not dropped onto the synchronization pane. In order to have connections available, we could at least allow connections to objects that are already synchronized.

Creating Awareness of Changes Another problem occures when one user is rejoining an existing session. The work that has been done in the time he or she was offline will be synchronized completely to his or her client. This hard synchronization of the state is lacking of awareness of what is changed. We imagine to create a timeslider with wich the user can replay the actions at his or her own speed. Additionally, diffing to the latest version of an object can create an understanding of what changed with the updates that the user just received.

[41] e.g. color, size, position

Snapshot of Drawings Currently drawings could not be reused since they can not be accessed as usual morphs when the user interacts on the whiteboard. For recomposition puposes we thought of a tool that would allow the user to take photos of the drawings which are then pasted to a new morph that can be dragged around just like any other morph.

Alternative Clients The design allows to implement different clients that communicate over the same synchronization server. It would be interesting to implement clients in systems different from Lively Kernel. That could attract new users and by that create more freedom of the tools the users have to use.

7.3 Server Improvements

The software of the synchronization server could improve as well. By now when the server crashes it has to be restared by hand. By executing the server in a loop that restarts the server as it crashes, this problem can be solved.

Further, we made sure that broadcasting is possible, but only on the level of channels. To be able to send messages to every client that is connected to the server for example for maintainance reasons there should be a way to broadcast messages to all connected clients.

In order to enable a better integration into other existing systems we could use a standard instant messaging protocol like XMPP.

Lastly, it could be interesting to implement the system without a synchronization server. Communication would have to happen in a peer-to-peer manner. A problem with that approach could be that it would possibly use more bandwidth since one client has to send its changes to multiple clients instead of just the server. Another problem could be how the clients get to know each other.

8 Conclusion

This work builds a basic implementation of synchronous collaboration support in Lively Kernel. We found solutions for fast sharing of content. This enables users to work together on synchronized objects. Further we illustrated how collaborative work can be supported on the levels of idea finding and implementation.

Nevertheless two main problems remain that need to be faced in order to create a productive collaboration system. First, the problem of overlapping actions [4] needs to be solved. The present system does not handle actions that happen at nearly the same time on the same object. As an example, one user might change the color of a morph. If his or her collaborator is changing the color of the same morph before the change arrives at his or her client both synchronization states will not be the same. This is because both clients receive the update of the other client. An approach to solve this was given in Future Work.

Secondly, the awareness for people (re-)joining a session needs to be improved. Currently there is no way to see what happend in the meantime[42]. We thought of a solution for this problem as well. An approach could be a timeslider that enables the user to scroll through the actions that happend.

The chapter Future Work showed that there are several ideas for features to improve the system. Finally, built on Lively Kernel the system is a suitable platform for collaboration research.

[42] The time the participant was offline

References

1. Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: The Definitive Guide Time to Relax. O'Reilly Media, Inc., 1st edn. (2010)
2. Dourish, P., Bellotti, V.: Awareness and coordination in shared workspaces. In: Proceedings of the 1992 ACM conference on Computer-supported cooperative work. pp. 107–114. CSCW '92, ACM, New York, NY, USA (1992)
3. Dourish, P., Bly, S.: Portholes: supporting awareness in a distributed work group. In: Proceedings of the SIGCHI conference on Human factors in computing systems. pp. 541–547. CHI '92, ACM, New York, NY, USA (1992)
4. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. SIGMOD Rec. 18(2), 399–407 (Jun 1989)
5. Graham, T., Phillips, W., Wolfe, C.: Quality analysis of distribution architectures for synchronous groupware. International Conference on Collaborative Computing: Networking, Applications and Worksharing 0, 41 (2006)
6. Ingalls, D., Palacz, K., Uhler, S., Taivalsaari, A., Mikkonen, T.: The lively kernel a self-supporting system on a web page. In: Hirschfeld, R., Rose, K. (eds.) Self-Sustaining Systems, Lecture Notes in Computer Science, vol. 5146, pp. 31–50. Springer Berlin / Heidelberg (2008), 10.1007/978-3-540-89275-5_2
7. Krahn, R., Ingalls, D., Hirschfeld, R., Lincke, J., Palacz, K.: Lively wiki a development environment for creating and sharing active web content. In: Proceedings of the 5th International Symposium on Wikis and Open Collaboration. pp. 9:1–9:10. WikiSym '09, ACM, New York, NY, USA (2009)
8. Lincke, J., Krahn, R., Ingalls, D., Roder, M., Hirschfeld, R.: The lively partsbin–a cloud-based repository for collaborative development of active web content. Hawaii International Conference on System Sciences 0, 693–701 (2012)
9. Ohkubo, M., Ishii, H.: Design and implementation of a shared workspace by integrating individual workspaces. SIGOIS Bull. 11(2-3), 142–146 (Mar 1990)
10. Rauch, G.: socket.io (2012), `http://socket.io`, visited 27.06.2012
11. Stamm, S.: Handling Touch Events on Mobile Devices for Lively Kernel. bachelor's thesis, Hasso-Plattner-Institut, Potsdam, Germany (June 2012)
12. Stefik, M., Bobrow, D.G., Foster, G., Lanning, S., Tatar, D.: Wysiwis revised: early experiences with multiuser interfaces. ACM Trans. Inf. Syst. 5(2), 147–167 (Apr 1987)
13. Stefik, M., Foster, G., Bobrow, D.G., Kahn, K., Lanning, S., Suchman, L.: Beyond the chalkboard: computer support for collaboration and problem solving in meetings. Commun. ACM 30(1), 32–47 (Jan 1987)
14. Stewart, J., Bederson, B.B., Druin, A.: Single display groupware: a model for co-present collaboration. In: Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit. pp. 286–293. CHI '99, ACM, New York, NY, USA (1999)
15. Thomschke, A.: Object Diffing and Merging. bachelor's thesis, Hasso-Plattner-Institut, Potsdam, Germany (June 2012)
16. Tilkov, S., Vinoski, S.: Node.js: Using javascript to build high-performance network programs. IEEE Internet Computing 14, 80–83 (2010)
17. Ungar, D., Smith, R.B.: Self. In: Proceedings of the third ACM SIGPLAN conference on History of programming languages. pp. 9–1–9–50. HOPL III, ACM, New York, NY, USA (2007)