

Building Distributed Web Applications based on Model Versioning with CoObRA: an Experience Report

Nina Aschenbrenner, Jörn Dreyer, Marcel Hahn, Ruben Jubeh, Christian Schneider, Albert Zündorf
Kassel University, Germany
nina.aschenbrenner@cs.uni-kassel.de, joern.dreyer@uni-kassel.de, hahn@cs.uni-kassel.de,
ruben.jubeh@uni-kassel.de, christian.schneider@uni-kassel.de zuendorf@uni-kassel.de

Abstract

Originally, model versioning has been developed to enable teams of developers to work on common model data, concurrently. We have the idea to use the same techniques to facilitate the collaboration of collaboration applications. Multi threaded applications share a common main memory. Thus, all threads have access to the full data structures and each thread may query and update the data structures, concurrently, in order to fulfill its tasks. In distributed applications, each distributed process has access only to its own share of the data model. In order to query and update remote data structure parts, the process has to send an appropriate request to the process, that owns that data. Transferring complex data structures, e.g. as query result, from one process to the other requires tedious data serialization and deserialization mechanisms. To overcome these problems, this paper proposes to replicate model data for each process and to use model versioning techniques to synchronize the different model data replicas. We have built a web based workflow editor and a web based version of a Ludo game to validate this idea. This paper reports about our experiences with the data replication approach and our experiences in using it for web applications.

1. Introduction

Building multi threaded applications is significantly easier than building a distributed (multi process) application. In multi threaded applications one deals with less heterogeneity and in addition one may exploit the concept of a shared memory. Thus, if the application deploys a complex object structure modeling its workspace, each thread has full access to this common object structure and may operate on this object structure, concurrently to the other threads. In case of write operations, some synchronisation mechanism like semaphores suffice to coordinate the multi-

ple threads. This is well addressed by many programming languages and approaches, cf. for example Java threads and their synchronisation concept.

In distributed applications, one deals with multiple processes that may run on different machines using heterogeneous system platforms with separate main memories. While the problem of heterogeneity is well addressed by approaches like Corba [5] and other techniques for modern service oriented architectures, the problem of the separate main memories is not yet sufficiently addressed.

However, many CASE tools provide mechanisms for versioning model data. These mechanisms enable multiple team members to work on the same model, concurrently. Thus, we came up with the idea to use similar versioning mechanisms in order to provide some concept of a shared memory for distributed applications.

We have developed this idea in the area of modern ajax based web applications. Such an ajax based web application runs within a web browser. It is started by visiting a certain URL. Then, the user may interact with the resulting web page and these interactions will perform some local computation or do some server communication and then show the result within the already loaded web page. Such an ajax based web application may deploy local data. However, whenever the user closes his or her browser or if the user just changes to another URL or the user does just a refresh of the current page, the web application might be terminated and all local data might be lost. Thus, ajax based web applications need some mechanisms to store their runtime data persistently, e.g. on a server. In addition, some of these web applications may work on common data (e.g. a database) shared by multiple users. Thus, such modern web applications may need to work on common data, concurrently.

The standard approach for such a setting deploys a relational database on some server that coordinates the data access. The business logic of the application is usually implemented on the server side, too. Thus, the web client merely serves as a visual user interface for the application.

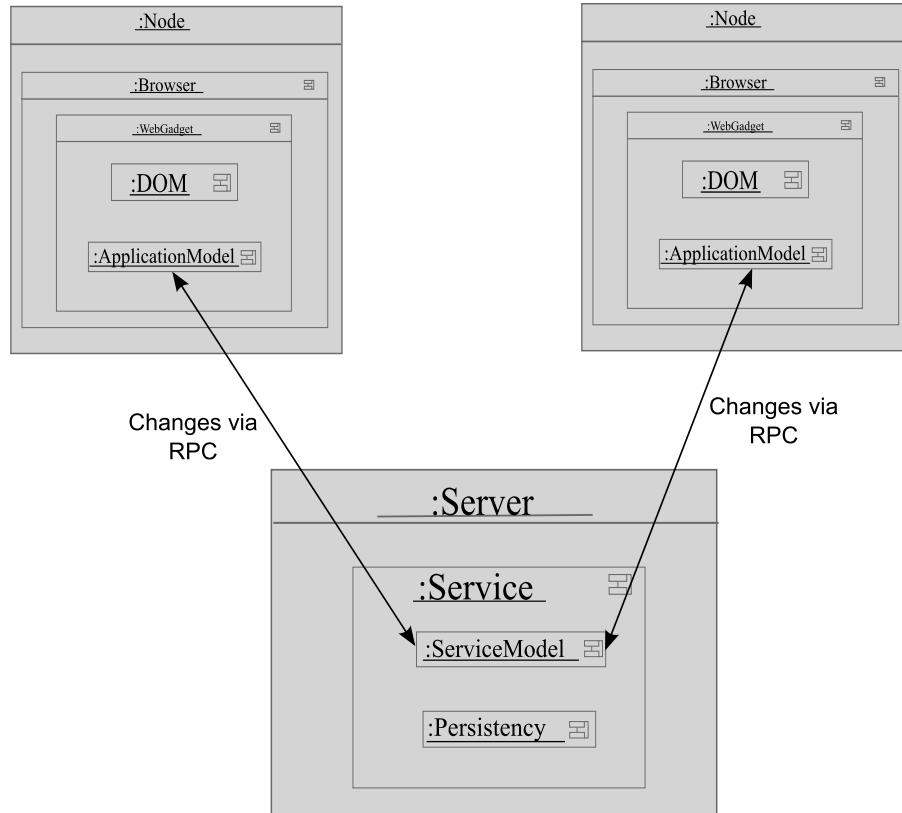


Figure 1. Reference Architecture of Distributed Web Applications

However, modern ajax based web application tend to do certain computations on the client side and thus they will need some local data on the client side and thus they need some mechanisms to handle persistency of such local data. If more functionality is transferred to the client, at some point, the need for coordination of local data between multiple clients arises. This paper shows our experiences in using model versioning techniques to leverage such applications.

2. Architecture

Figure 1 shows our proposal for the architecture of a distributed web based application as developed in [1]. We call it a Web 2.0 application. These applications, compared to ordinary web based applications delegate work to their client sides. The client sides use browsers to show some web gadgets, which are small web applications. These gadgets should be programmed according to the model view controller pattern, i.e. they should separate their representation and their data model. The standard representation of a web page in a web browser is the so called document object model (DOM). To update the view, the application may just transform its current DOM and the browser will show the

new content.

Usually, the client has only this representation data and the server has the model data and the application logic. Here our reference architecture differs from traditional web applications. We propose to deploy the model data and the application logic, i.e the business rules and model transformations to a large extent on the client side. This application data model and its operations may be developed with usual object oriented techniques or with model transformation e.g. with Fujaba story diagrams, cf. [3]. In order to deploy the resulting Java code on a web browser, we use Google Web Toolkit technology [6]. With the GWT compiler the Java implementation of our application model is translated into JavaScript code that can be executed inside a web browser.

To enable the model view controller pattern for connecting the clients DOM based GUI with the clients application data model, we need a property change management infrastructure. In our approach, we generate this infrastructure together with application data model code and use GWT to translate it to Javascript for the web clients.

Next, we need persistency support, i.e. we want to store our application data on some server and in case of data modifications we want to keep the server model data

consistent with the application model data. Therefore, we have adapted our CoObRA framework [4] to the specific needs of web applications. The CoObRA framework provides a generic data replication mechanism. It persistently stores object graphs as a sequence of transactional change streams. CoObRA supports the merging of the data model copies when multiple users work collaboratively on the same model. With CoObRA, it is very simple to make application models persistent, and because of the underlying change stream architecture, undo and redo functionality comes for free. In order to deploy CoObRA mechanisms on the web clients, we had to downstrip some core CoObRA classes in order to meet some limitations of GWT. In GWT, all functionality needs to be provided as Java source code (to be compiled into Javascript). This holds also for the libraries that one deploys. Only certain parts of `java/lang` and `java/util` are provided as part of the GWT runtime package. Thus, we had to get rid of library dependencies, e.g. the `log4j` library. In addition, the parts of CoObRA that write model changes to the disk make no sense on the web client side. Instead, we had to add some code to be able to transfer model updates between web clients and the corresponding server via HTTP mechanisms.

With these modifications our web clients may now use a WebCoObRA component that provides undo redo functionality and that may send change protocols from the clients to the corresponding servers. The latter mechanism enables CoObRA based data replication mechanism and team collaborations. On the server side, we use a usual CoObRA implementation in order to achieve persistency and recovery.

Altogether, using WebCoObRA one may develop modern Web 2.0 applications like usual Java applications based on a meta model and a graphical user interface and the usual MVC pattern in order to keep visualization and data consistent. With the help of the WebCoObRA framework the web application may replicate its application data on a server and share it with concurrent applications. This also provides a simple mechanism for persistency and recovery on the server side. This persistency mechanism avoids the usual object relational mismatch acquainted with relational data bases and thus it provides a more lightweight implementation approach. On the other hand, CoObRA is not yet appropriate for handling large data volumes (which is a strength of relational databases). These limitations will be addressed in our future work section.

3. Example

In [1] we have developed the reference architecture for Web 2.0 applications and a first version of the WebCoObRA framework. In [1] we used this approach to build a distributed web based workflow system. In this paper we

report about our experiences in applying the same approach to a web based version of our Ludo game, cf. Figure 2

The application logic and application data of our Ludo game had already been modeled with the CASE tool Fujaba. We had just to adapt it somewhat to the new WebCoObRA framework. For the user interface, the old Ludo game deployed WhiteSocks, our generic user interface framework for sprite graphics based on Swing, cf. [2]. For this work we have ported WhiteSocks to BlackSocks providing almost the same sprite graphics based user interface framework but now based on the user interface components of the GWT framework.

In addition, the code generation of Fujaba had to be further adapted in order to provide some more reflective features used by WebCoObRA and by the BlackSocks framework.

As a result, we have a browser based Ludo game for multiple players on multiple computers, cf. Figure 2. All player clients that collaborate in the same game share the same replicated ludo model data e.g. for fields, stone positions or die values. Figure 3 shows a cutout of this application object model that is copied to each client. Objects `fa` through `ff` and the corresponding next links model the track where the stones move, e.g. stone `sa`. This object structure is modified when a player rolls the die or moves a stone to another field. These modifications are automatically observed by the WebCoObRA component and replicated / committed to the server that hosts this game. The server then replicates this data to all (other) players, immediately.¹ Each change to the ludo model data, either initiated by the user directly or caused by an update event from the server is observed by our generic BlackSocks user interface component and displayed immediately. Thus, each player sees the progress of the game and may rethink his or her strategy of gameplay and everything else as if the game was played by all players together on one computer (or on a real ludo board).

4. Lessons Learned

Our lessons learned have to be split into two parts. First, we discuss the use of model versioning for distributed applications and second, we discuss the special case of modern web applications.

For general distributed applications, having local data in each client is the normal case. These applications deal with the problem of distribution usually by assigning the responsibility for certain parts of the overall data model to distinct applications. This results in components that combine data and functionality and provide this as a service to other components. Instead of sharing data, common data is transferred to a common component that may be used by multi-

¹The WebCoObRA server keeps a socket connection to each client in order to be able to inform its clients on update events.

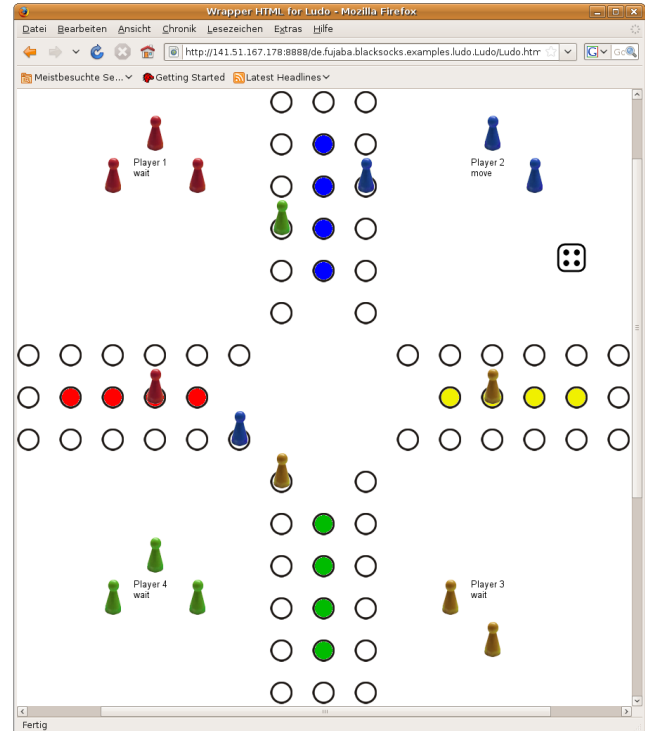
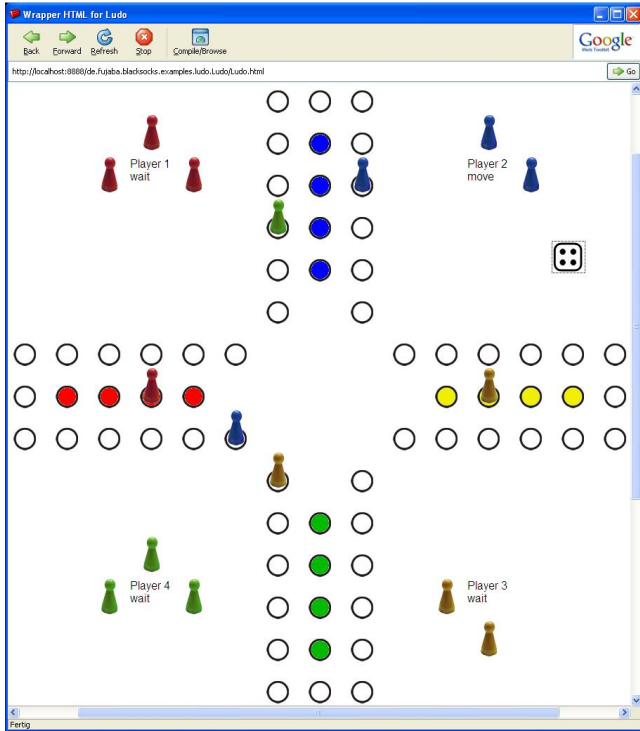


Figure 2. Two web Ludo games collaborating via WebCoOBRA

ple applications. In the case of our Ludo game this means, one server component would own the game data and the game logic and the user applications would just provide the user interface.

However, modern graphical user interfaces are build according to the model view controller pattern. Such an application has a representation layer for the user interface and a model layer for the data and these two layers are connected by a controller layer that listens to action events on the representation layer and to model changes on the model layer and keeps these two layers consistent. Due to network overhead and latency, having the view and the model on separate nodes leads to a poor user experience. Thus, the modern MVC user interface concepts has a conflict with the service component concept of usual distributed applications.

By replicating the model data in all applications, our approach resolves this conflict. The applications may be build according to the usual MVC approach. This allows to apply the mature MVC technique for distributed applications. Actually, we made the experience that it is reasonably simple to turn a non-distributed application like our first version of Ludo into a distributed application for multiple users. We just added the CoObRA support and we were able to run multiple instances of the Ludo application. Each instance sends all its data changes to a common version repository

which acts as a central synchronization point. All other instances receive these changes via updates. In case of the Ludo game only one player acts at a given time, thus we have no problems with concurrency and the model doesn't need to support certain synchronization points. In general, concurrent use of an application which was designed as a non-distributed applications may challenge the versioning mechanism and result in certain merge conflicts. Such merge conflicts need then to be resolved somehow in order to coordinate the work of multiple users. In principle, this is a severe problem that might invalidate our simple approach for turning non-distributed applications into distributed applications for multiple users. However, due to our experiences a simple first-come-first-serve strategy serves very well for the resolution of merge conflicts. This means, in case of a merge conflict, the local changes are discarded and the server changes are applied. If changes are grouped according to user actions, this results in a kind of transaction mechanism: either a user action creates no conflict and succeeds or it has a conflict and then it is undone. Thus, according to our experiences, using our data replication approach it is surprisingly simple to turn a single user application into a distributed multiple user application.

Actually, the most tricky part of coordinating multiple instances of the same application via our replication approach is that initially all instances have exactly the same

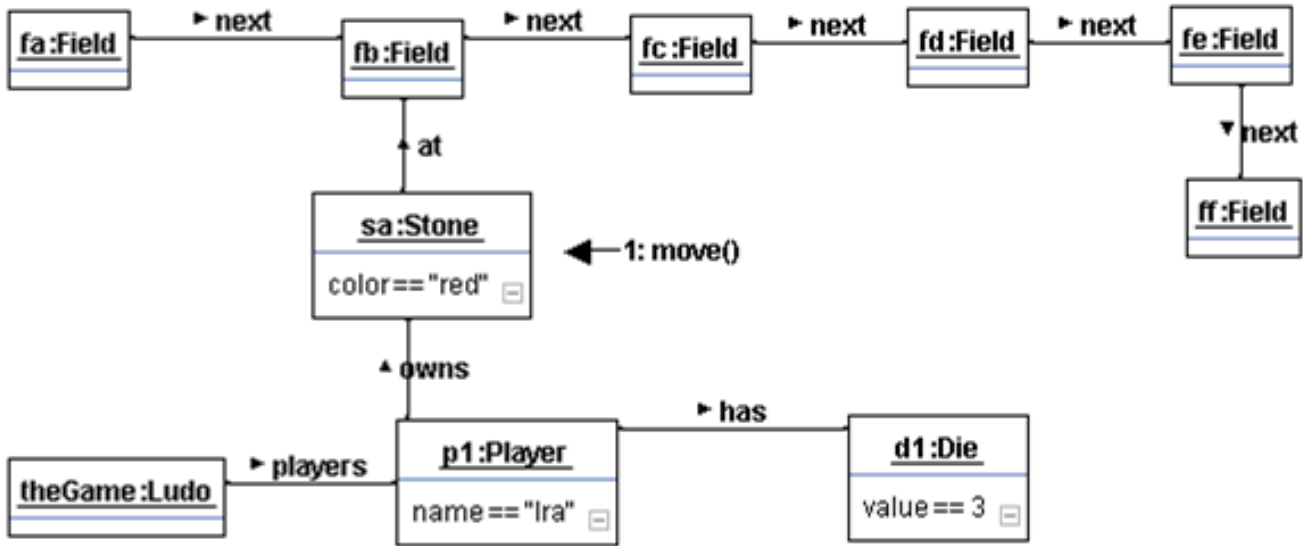


Figure 3. Application Model for Ludo (cutout)

data. In case of Ludo this means, the application did not know which player was using it and thus, each player could move the pawns of each other player. Thus, we had to split our data into replicated data (the major part) and a small part of actually local data describing which player is using this instance of the game. Generally, distributed applications that use our data replication approach will have to distinguish the shared model data and some local session data like user name or login, session specific preferences or session specific view information like personal scrolling, zooming, layouting, or opening and closing of different model parts.

Second, we learned that it was again surprisingly simple to follow this data replication approach for a distributed web application. At the first sight, it is somewhat bizarre to run a full MVC based application as a JavaScript component in a web browser and to use a data replication mechanism to enable data persistency and multiuser support. However, it works very well. The GWT user interface library provides user interface elements which are pretty similar to Swing elements. Thus, it was very simple to port our Swing based WhiteSocks framework to the GWT based BlackSocks framework.

Similarly, it was very simple to adapt the code generation of Fujaba to the limitations of GWT in order to allow the deployment of complex object models in a browser. Actually, even Fujaba's graph transformations required only very little efforts in order to run as GWT code within a browser.²

Providing WebCoObRA required somewhat more effort. We had to identify the required parts or the CoObRA framework and to strip off the usage of foreign libraries from

²Some small parts of the Fujaba runtime library had to be provided as source code for GWT

these parts. Actually, since CoObRA is based on streams while our GWT based client server communications deals with strings, we have reimplemented the core class of CoObRA that interprets change streams. However, this results in some thousand lines of code, only.

Altogether, provided with our infrastructure, building a multi user, distributed web application has shown to require only reasonable additional overhead compared to a standalone single user application. To enable data replication one just adds WebCoObRA support to his or her data model. Concurrency conflicts manifest in merge conflicts that may be resolved by discarding local changes. If changes are grouped to user interactions, this behaviour is comparable to database transactions. For the user interface, one uses GWText instead of Swing. This results in a modern ajax based web application with a lightweight, generic persistency mechanism and a simple multi user support. Our experiences are such encouraging that we plan to apply and extend this approach in our future work.

5. Future work

In our current approach, the (Web)-CoObRA-Framework holds the whole object data model of an application in memory. That is not sufficient for large-scale data models, or when one wants to distribute the data model on different servers. A first step towards distributed models is to split the whole object graphs into partitions, which are handled separately. Each partition is stored individually as change stream on disk, but can be hold in memory when needed. That is reducing the memory requirements dramatically. At partition transitions, we plan to implement

a transparent mechanism with proxy objects, that will load the connected objects only on request, so called lazy-loading. The server repository itself can monitor the access to those partitions and discard the object graph for a partition to free up memory when not in use. Combined with that, we plan to extend the CoObRA property change mechanism to a general client/server publish-subscribe architecture: servers can always track connected clients and don't need to hold the object model, but only to deliver changes to other subscribed clients.

Another important issue left out so far was authentication and access control. Applied to the simple Ludo example, it should not be possible to move pawns of the other players. By annotating the UML associations between objects, we plan to implement access control seamlessly within the access methods, which are generated by our code generator. Technically, the server should be able to filter incoming and outgoing change streams to implement the access rights check.

6. Summary

Using our approach, we are able to build distributed, collaborative applications with little effort on top of our web-enabled versioning framework, (Web-)CoObRA. Existing single-user applications based on this framework are surprisingly simple to adapt. But there is room for further improvements: Scalability and security, traditional problems of web applications, require some extra work to be done.

References

- [1] N. Aschenbrenner, J. Dreyer, R. Jubeh, and A. Zündorf. Fujaba goes web 2.0. In U. Aman, J. Johannes, and A. Zndorf, editors, *6th International Fujaba Days*, pages 10–14, Dresden, Germany, 2008.
- [2] I. Diethelm, R. Jubeh, A. Koch, and A. Zündorf. WhiteSocks - A simple GUI Framework for Fujaba. In *5th International Fujaba Days*, Kassel, Germany, October 2007.
- [3] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation*. Paderborn, Germany, 1998.
- [4] C. Schneider. *CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Sperrkonzepten*. PhD thesis, 2007.
- [5] W. A. R. Thomas J. Mowbray. *Inside Corba*. Addison Wesley Logman, 1997.
- [6] The Google Web Toolkit. <http://code.google.com/webtoolkit>, 2008.