Patterns for Managing Shared Objects in Groupware Systems

Stephan Lukosch and Till Schümmer

University of Hagen
Department for Computer Science
58084 Hagen, Germany
{stephan.lukosch, till.schuemmer}@fernuni-hagen.de

Abstract

Groupware is a technology that facilitates teamwork. Developing groupware is a difficult and time-consuming task. To enable collaboration, groupware applications have to share data. Sharing data is among the main obstacles during groupware development. There exist many groupware platforms which offer programming abstractions to relieve developers from recurring issues. However, these platforms have one problem. They are too prescriptive. To assist developers in the development process of groupware applications, we provide a pattern language that offers proven solutions for recurring issues and allow developers to reuse them in the intended context.

1 Introduction

An increasing number of applications are currently designed for the use of more than one user. Examples are multi-player games, web-sites that foster interaction between the audience, applications for mobile interaction between users, systems that foster collaborative learning, or peer-to-peer applications to name only a few application areas. All these systems support group interaction. Groupware supports communication, coordination, and cooperative work between team members. A well-known definition of groupware was stated in [11]:

> Groupware are computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment.

Developing groupware applications is a difficult and time-consuming task. Apart from the actual task of the application, e.g. editing texts or spreadsheets,

– network connections between the collaborating users have to be established,

– parallel input from many users has to be handled,

– specific group functions have to be included, and

– shared data has to be managed.

All the difficulties relate to the management of shared objects (or more general shared data). We define a shared object as an object that is used by more than one user. The usage may be at the same time (synchronous) or at different points in time (asynchronous).

To communicate about the shared objects (and exchange information about the shared object's state), the different users have to connect their clients (the computer that they are using) using a network connection.

Since they work on different machines, each user will be able to access the input devices (often at the same point in time). Thus, the control flow within the application gets much more complicated. Unlike in traditional applications where one user changes the shared data at a time (triggered by one input device) systems based on shared objects can have multiple requests for changing the shared object's state at the same time. Maintaining consistency becomes an critical issue in such situations.

Related to the problems raised by the multiple control flows is the problem of making the users aware of other users' activities. In traditional single user applications, data changes are initiated by only one user. Thus it is easy to understand for the user, why data changed (if we ignore annoying auto-changing data like in Microsoft Office, where so-called "Assistants" permanently mess up the user's data....). In groupware applications, it is intended that the shared objects change without the local user's initiative.

The patterns in this paper complement other patterns proposed for higher level groupware design. Tab. 1 provides an overview about how the current set of patterns in our work can be organized.

Our patterns follow the pattern structure outlined in the Oregon Software Development Process [43]. While using the pattern names in software labs and in our group, we noticed that it was easier for all persons working with patterns to remember more funny pattern names. Thus, we use more funny *pattern names*, which are followed by a more accurate *AKA* name. The names are followed by an image which gives a metaphoric description of the pattern and a small pattern map showing the position of the actual pattern in context of the pattern language. The images are followed by the *intent* and the *context* of the pattern. All these sections help the reader to decide, whether or not the following pattern may fit into his current situation.

Then follows the core of the pattern composed of the *problem* and the *solution* statement separated by a *scenario* and a *symptoms* section. The *scenario* is a concrete description of a situation where the pattern could be used, which makes the tension of the *problem* statement tangible. The *symptoms* section helps to identify the need for the pattern by describing aspects of the situation more abstract again.

After the *solution* section, the solution is explained in more detail and indications for further improvement after applying the pattern are provided. The *participants* section explains the main components or actors that interact in the pattern and explains how they relate to each other. The *rationale* section explains, why the forces are resolved by the pattern. Unfortunately, the application of a pattern can in some cases raise new unbalanced forces. These counter forces are described in the section labelled *danger spots*.

Pattern (collection)	Description	Reference
Open Communities	A pattern language for supporting virtual communities in the process of integrating new members.	Shepherded at EuroPLoP2004 (also in this volume).
More than 1000 words	Pattern for integrating graphical artifacts in communication.	Pattern discussed at CHI2003.
Room	Support communication and group management.	Pattern discussed at CSCW2002.
The Public Privacy	Patterns for Relationship Management in Collaborative Systems.	Discussed and shepherded at CHI2004.
GAMA - A Pattern Language for Computer Supported Dynamic Collaboration	Patterns for creating dynamic teams based on the awareness of other users in the system.	Shepherded at EuroPLoP2003 [42].
Activity Awareness Patterns	Patterns for informing users about other users' activities in a shared workspace.	Shepherded at EuroPLoP2002.
Patterns for managing shared objects.	Technology centered patterns for concurrent use of shared objects (in centralized and distributed environments).	In this paper.

Table 1: Overview about low-level and high-level patterns

The pattern map in fig. 1 shows the patterns of this language and the relations between the patterns. The patterns in the grey boxes show patterns from other pattern languages that are related to our patterns. If a pattern A points to another pattern B, pattern A is important as context for the referred pattern B. One can start exploring the pattern language by reading CENTRALIZED OBJECTS$_{\rightarrow 2.1.1}$. One reasonable sequence through the pattern language can then for instance be CENTRALIZED OBJECTS$_{\rightarrow 2.1.1}$→REPLICATE FOR SPEED$_{\rightarrow 2.1.2}$→UPDATE YOUR FRIENDS$_{\rightarrow 2.2.1}$→BELIEVE IN YOUR GROUP$_{\rightarrow 2.3.1}$.

The patterns are:

CENTRALIZED OBJECTS$_{\rightarrow 2.1.1}$: Allow users to access data objects remotely.

REPLICATE FOR SPEED$_{\rightarrow 2.1.2}$: Allow users to access data objects without network delay.

REPLICATE FOR FREEDOM$_{\rightarrow 2.1.3}$: Provide wire- and waveless data access.

GUESS WHAT I NEED$_{\rightarrow 2.1.4}$: Reduce network communication costs and response time by replicating only selected shared objects.

UPDATE YOUR FRIENDS$_{\rightarrow 2.2.1}$: Distribute local state changes to the other users to achieve consistency.

MEDIATED UPDATES$_{\rightarrow 2.2.2}$: Minimize the administrative load for updates by using a central instance that dispatches the updates.

BELIEVE IN YOUR GROUP$_{\rightarrow 2.3.1}$: Change data optimistically and undo the operations if users performed conflicting changes.

DON'T TRUST YOUR FRIENDS$_{\rightarrow 2.3.2}$: Allow only one user at a time to change the same data object.

DETECT A CONFLICTING CHANGE$_{\rightarrow 2.3.3}$: Detect conflicting changes.

LOVELY BAGS$_{\rightarrow 2.3.4}$: Use bags for storing concurrent objects because they provide the best concurrency behavior.

The following sections describe each pattern in more detail.

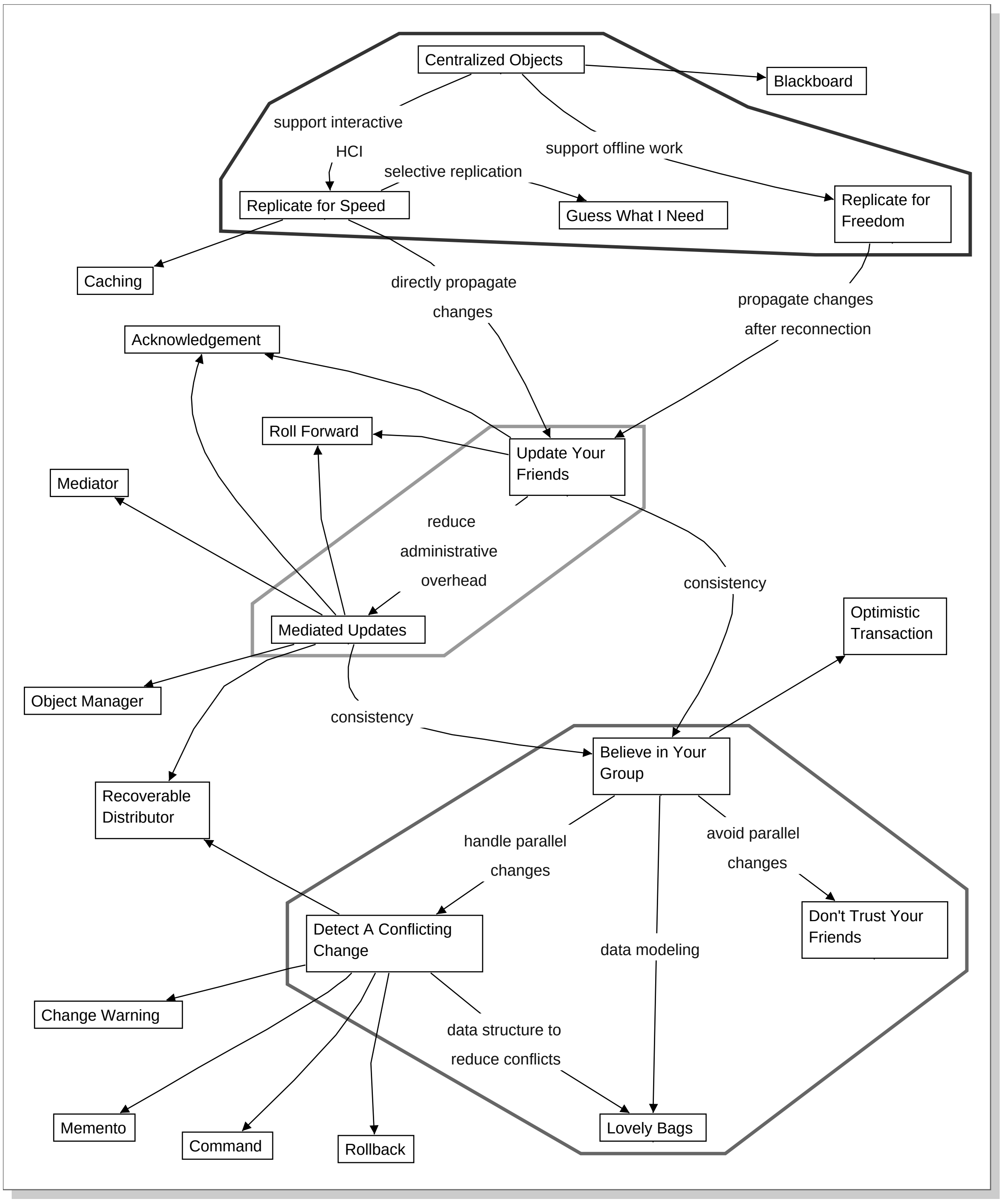
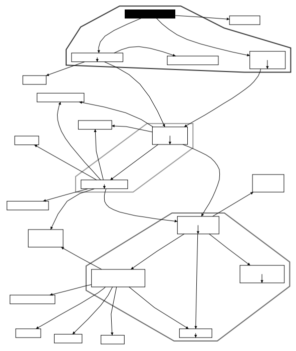


Figure 1: Pattern map

2 The Pattern Language

2.1 Data Sharing

2.1.1 Centralized Objects



Intent — Allow users to access data objects remotely.

Context — You are developing a groupware application. Now you are thinking of how to organize the data so that many users can work with it.

✣ ✣ ✣

Problem — **To enable collaboration users must be able to share the data.**

Scenario — Think of a group of researchers who want to exchange new research ideas. They meet at a conference and present their ideas (on a large screen that sets a group focus). After exchanging first ideas, they arrange the research concepts on a blackboard to identify relations between the different research ideas.

Now think of the same group acting in the virtual space. How can such a group share a common focus (on the presentation) or collaboratively arrange the different research concepts without physically meeting at the same place?

Symptoms — *You should consider to apply the pattern when ...*

- users have to meet in person to collaborate because they don't know how to exchange data.
- users with a permanent network connection want to collaborate in an interactive application but cannot share data.
- users cannot establish a common ground, i.e. get an understanding of the shared data.

Solution — **Therefore: Manage the data necessary for collaboration on a server that is known to all users. Allow these users to access the data on the server.**

Collaborations — The main participant is the server. The users who want to collaborate must know the address of the server. The server stores the data which is shared for collaboration. The users access the server to retrieve shared data and locally display it. When changing the data, they again have to contact the server which performs the change. Accessing and changing the data can be implemented by, e.g., using remote procedure calls (RPCs) [2].

Rationale

Since all users rely on the same shared data, they can access this data and collaborate. Besides this, the main advantages of this pattern are:

- All users know the address of the server and therefore can find the shared data easily and fast.
- As the shared data is kept and only changed on the well-known server, it is easy to maintain consistency.

Danger Spots

- The central server is a bottleneck for accessing and changing the shared data. When all users intensively access the shared data, the response time of the application increases. This can make collaboration difficult.
- If the central server becomes unavailable, users cannot access the shared data anymore.

Known Uses

Suite [10] uses a DOCUMENT-VIEW variant of the MODEL-VIEW-CONTROLLER pattern [4] for dividing model and presentation. Suite keeps the model objects (called active variables) on the server. The views and controllers (called dialogue managers) are kept on the clients. Whenever a client's dialogue manager needs to modify the data, it asks the server to change the model.

NSTP [33] provides a service for data sharing in synchronous multi-user applications. The service is offered by a well-known notification server. The server provides clients access to a shared state and notifies these clients, whenever the shared state changes. The server contains two kinds of objects: places and things. A place contains the shared state and partitions the resources of the server among several applications. Each application uses at least one place. A client joins a collaborative application by entering the place. Things are the actual objects that maintain the shared state and can be created, changed, locked, unlocked, and deleted.

DreamObjects [29] supports a variety of distribution schemes. Among these distribution schemes DreamObjects supports a variant of the centralized distribution scheme. In this variant each participating site can act as server for a central object and thereby easily introduce local data in a collaborative session. Changes to these centralized object are only performed at the object hosting site. All other sites can transparently access a centralized object via so-called local substitutes that handle all necessary mechanisms.

WIKIs [28] are a special kind of web applications, on which users can modify the stored pages using a simple set of editing rules. Typically, a client first requests a page for viewing. The WIKI engine creates a HTML version of the page that is stored (in plain text) on the server. If the user decides to edit the page, the server creates a form-version of the page that includes an input field for editing the page's source. Finally, if the client decides to store the page again, the changed text is sent to the server and stored in the server's pages.

✣ ✣ ✣

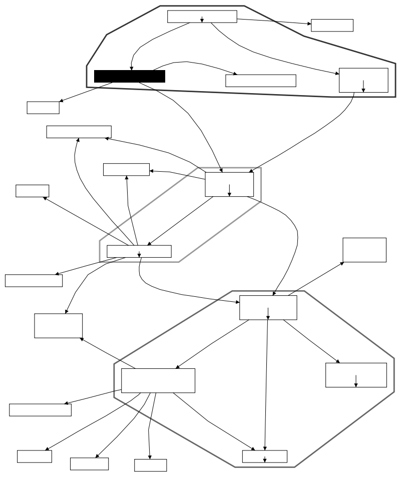
Related Patterns REPLICATE FOR SPEED$_{\rightarrow 2.1.2}$: If the collaborative application is highly interactive and all users perform many changes, REPLICATE FOR SPEED decreases the response time of the application. Additionally, REPLICATE FOR SPEED increases the availability of the shared data.

REPLICATE FOR FREEDOM$_{\rightarrow 2.1.3}$: If there is no permanent network connection, use REPLICATE FOR FREEDOM to make the data accessible for disconnected users.

BLACKBOARD: The BLACKBOARD pattern [4] makes use of structures described in the CENTRALIZED OBJECTS pattern. It is an example for distributed processing with exchange of intermediate results (called hypotheses) at a central location. The hypotheses are thus the objects that are shared between processing units. The blackboard that manages hypotheses is an instance of the server in the CENTRALIZED OBJECTS pattern.

2.1.2 Replicate for Speed

Photo: Loikkanen, Antti, Finland, `www.visipix.com`



When the hare therefore arrived in full career at the lower end of the field, the hedgehog's wife met him with the cry, "I am here already!" The hare was shocked and wondered not a little. He thought it was the hedgehog himself who was calling to him, for the hedgehog's wife looked just like her husband.

From Jacob Ludwig Grimm and Wilhelm Carl Grimm: "Hase und Igel"

Intent — Allow users to access data objects without network delay.

Context — You have considered to manage the shared data using Centralized Objects$_{\rightarrow 2.1.1}$. Now you are concerned about speed.

✣ ✣ ✣

Problem — **The response time of interactive applications has to be short. The network latency and delay wastes time in distributed systems. Thus interactive applications are inappropriate if the response time depends on client-server communication.**

Scenario — Imagine a group that works with a collaborative diagram editor. Each diagram element is represented by a shared data object. Now, imagine that a user wants to drag a diagram element and drop it somewhere else on the drawing areas. If the diagram elements are managed on a server (compare Centralized Objects$_{\rightarrow 2.1.1}$), a user has to access the server for each movement step while dragging the diagram element. As this causes network communication, the movement of the diagram element is not smooth or may even freeze. This makes interaction with the editor hard.

Symptoms — *You should consider to apply the pattern when . . .*

– users with a permanent network connection collaborate in a highly interactive application.

- users perform many incremental changes on large objects.
- the response of the application is too slow.

Solution

Therefore: Replicate the shared data to the users' sites. Let a user change its local replicas and ensure consistency by using the UPDATE YOUR FRIENDS$_{\rightarrow 2.2.1}$ pattern.

Collaborations

When a user joins an already collaborating group, its site has to request all shared data objects from a site which already participates in the collaboration. Upon request the site transfers the shared objects to the requesting site.

Rationale

Since users can access the data locally, they can perform local changes or display refreshes without network costs.

Danger Spots

- Replication ensures that the delay for accessing the shared objects is low, but it may cause a large initial delay, when the common objects are accessed for the first time. Thus, you should schedule the initial transmission in times, where the user does not need fast response times (using the GUESS WHAT I NEED$_{\rightarrow 2.1.4}$ pattern).
- Replication causes high communication costs [16]. If this is an issue and you want to reduce the communication costs, use the GUESS WHAT I NEED$_{\rightarrow 2.1.4}$ pattern.
- As every user can locally access and modify the shared data, the consistency of the shared data is an issue. To ensure consistency, use the UPDATE YOUR FRIENDS$_{\rightarrow 2.2.1}$ pattern.
- Requesting and transferring the state of a shared object is complicated, as you have to ensure that the transferred object is consistent to all other replicas. There exist different approaches to solve this problem, compare [26] for a starting point.

Known Uses

GroupKit [38] organizes shared objects in so-called environments [37]. An environment is a hierarchical data structure, where a node either can hold a value or have other nodes as children. The runtime system transparently replicates an environment. A developer can bind callbacks to an environment and receive a notification when a node is added, changed, or removed.

COAST [40] allows clients to keep replicas of shared objects. To maintain a replica, the clients register at a central server, who provides a primary copy of the objects. Clients can directly change their replicas and visualize the new state of the replica in their user interface. This ensures a high level of interaction in the application. Whenever a client changes the state of a replica, this change is propagated by means of the MEDIATED UPDATES$_{\rightarrow 2.2.2}$ pattern. Changes can be state changes or the creation of new replicated objects.

DreamObjects [29] supports replicated objects for highly interactive applications. All participating sites maintain a replica and can perform reading accesses locally. Changes to the replica are handled transparently by a local substitute for the shared object that is returned to the developer when creating a new shared object.

HTTP/1.1 [12] supports caching at the client and the server site. Caching at the client site is the most common used practice. To ensure the consistency of the cached document HTTP/1.1 uses an expiration mechanism. Instead of again requesting a cached document the client asks the server if the document is still valid or is already expired. In connection with REPLICATE FOR SPEED a cached document can be compared with a replicated shared object.

✣ ✣ ✣

Related Patterns

CENTRALIZED OBJECTS$_{\rightarrow 2.1.1}$ can be used to reduce the communication costs and to simplify the management of the shared data.

GUESS WHAT I NEED$_{\rightarrow 2.1.4}$ reduces the communication costs by distributing a replica to only those users who access the shared data.

REPLICATE FOR FREEDOM$_{\rightarrow 2.1.3}$ supports collaboration between users that are not permanently connected to the network.

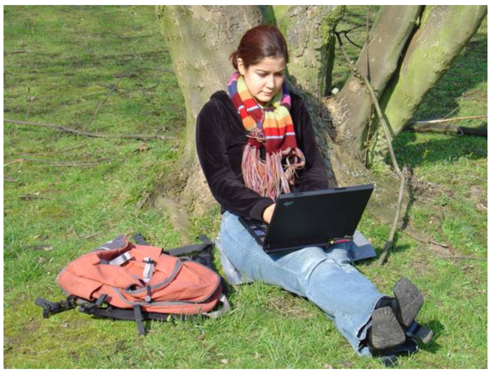
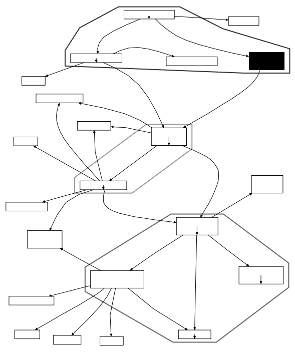
UPDATE YOUR FRIENDS$_{\rightarrow 2.2.1}$ can be used to propagate changes to all other sites that maintain replicas of the shared data.

FAIL-STOP PROCESSOR [39] discusses replication for another purpose: system failure. As in the REPLICATE FOR SPEED pattern, different clients keep replicas of the shared data. If one client fails, users are redirected to other clients. In the terminology of this pattern language, such behavior would be called REPLICATE FOR RELIABILITY.

CACHING [24] addresses the same problem as the REPLICATE FOR SPEED pattern. In both patterns, the response time becomes critical due to expensive retrieval of shared data. The CACHING pattern focusses more on applications in which users only retrieve data. REPLICATE FOR SPEED instead focusses on applications in which users interactively modify shared objects.

2.1.3 Replicate for Freedom

Alternative names: Offline work by replication

Intent Provide wire- and waveless data access.

Context You have created an architecture that allows users to access shared objects over a network, e.g. by using CENTRALIZED OBJECTS$_{\rightarrow 2.1.1}$. Now you want to allow them to work disconnected as well.

✣ ✣ ✣

Problem **Users may not have a permanent connection to the system, where relevant data is kept. Without a permanent or just a poor connection to the data, users will not be able to finish their work, if the data cannot be accessed.**

Scenario Imagine a group of authors who are working together on course on software engineering. hey have made the course draft accessible on a file server so that all authors can access it over the internet.

Unfortunately, they do not find much time to work with the documents at the customers' sites (where they would have web access). Fortunately, the authors also travel a lot, which gives them plenty of time to work on the course during their journey. But during the journey, they cannot access the internet and thus cannot access the shared data.

Symptoms *You should consider to apply the pattern when ...*

- users work disconnected with various devices on data that is kept at least on one system.
- users cannot work wherever they want.
- users need to work on correct and complete shared objects.
- users are connected via an unreliable network (e.g. GSM on a train of the German rail).

Solution **Therefore: Replicate the data to the user's device. Update the replicas whenever two systems which hold copies of the data connect.**

Collaborations Whenever a client connects to the network, it identifies relevant objects provided by a central server or another client. It replicates these objects by asking the publisher of the object for a replica. The client may then disconnect from the network.

If the client connects to the network again, it contacts the publisher of the replicas and ensures that both parties have an updated version of the replicated object (mechanisms for achieving this are presented in the UPDATE YOUR FRIENDS$_{\rightarrow 2.2.1}$ pattern).

Rationale Since the needed data is always on the user's device, the user is able to access this data when using the device.

Danger Spots In addition to the danger spots from REPLICATE FOR SPEED$_{\rightarrow 2.1.2}$, it might be complicated to decide which data is needed in disconnected situations. However, this issue cannot be solved on a technical level. Instead, it requires end-user involvement.

Known Uses **Sync** [32] is a framework that supports the development of mobile collaborative applications. It bases on replication and offers some basic classes which a developer can use to implement shared data objects. Applications developed with Sync use a central asynchronous synchronization, i.e. users can connect at different times with the server and balance their changes. The basic classes contain mechanism which solve a set of predefined conflicts that can occur when users changed the same data.

IMAP [8] uses replication of e-Mail messages and folders to support users to read and archive their mail on different clients. Each client can replicate the mail messages when connected to the IMAP server. In the off-line mode, the client can organize the mail (e.g. move messages to other folders or create new folders). When the client reconnects to the network, it synchronizes its local state with the state on the server (and makes the users changes persistent).

CVS: An example for asynchronous collaboration is the well-known CVS system. Users can checkout modules from a well-known server. The CVS server transfers the requested module to the user's site. After checkout the user can disconnect from the network and can access the module's data locally.

✣ ✣ ✣

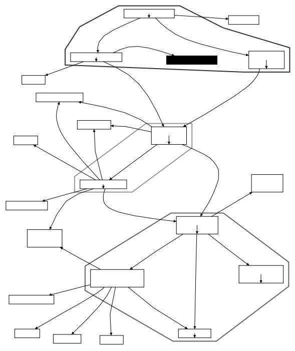
Related Patterns REPLICATE FOR SPEED$_{\rightarrow 2.1.2}$: The solution of REPLICATE FOR SPEED is comparable to the solution of REPLICATE FOR FREEDOM in the aspect of replicating objects to the clients. The difference is the way how changes are handles. Since the clients stay connected in REPLICATE FOR SPEED, they can ensure consistency directly after a user changed an object.

UPDATE YOUR FRIENDS$_{\rightarrow 2.2.1}$: To ensure consistency, a client has to inform other clients about changes in its replicas.

FAIL-STOP PROCESSOR [39] models replication for the purpose of reliability (cf. the discussion in the related patterns section of REPLICATE FOR SPEED$_{\rightarrow 2.1.2}$).

2.1.4 Guess What I Need

Alternative names: Partial or adaptive replication



Intent

Reduce network communication costs and response time by replicating only selected shared objects.

Context

You decided to use Replicate for Speed$_{\rightarrow 2.1.2}$ to speed up your application. Now you are thinking about reducing communication costs; especially in large data spaces.

✣ ✣ ✣

Problem

The response time of interactive applications has to be short. The network latency and delay wastes time in distributed systems. Full replication, i.e. every user maintains a replica, demands high communication costs for initialization and consistency. If network communication between the users is slow, the response time of interactive applications increases and makes collaboration hard.

Scenario

Imagine users collaboratively working in a text editor. The text document is structured into single logical pieces, i.e. words, sentences, paragraphs, and sections. Each section is represented by a replicated object. Normally, not all users view or edit all sections of the text document. However, as all sections are replicated, each user maintains a replica, which causes network communication costs during the initialization and whenever a section is changed. This wastes network communication capacities which is already low as the network connection between the users is slow.

Symptoms

You should consider to apply the pattern when . . .

– users complain that the initial replication of shared data takes too long.

– users do not need all shared objects at every point in time.

– users usually request replicas and then work on these replicas. Before working with another object, they request its replica, which takes time.

– the communication costs for maintaining shared objects are too high.

Solution **Therefore: Let a user only hold a replica for a shared data object which she currently accesses or is supposed to access in near future. The other shared data objects that also belong to the state of the application can be ignored without any adverse effect to the user interface.**

Collaborations Replicate a shared object as soon as a user accesses it. For this purpose, the user's site has to contact a site which already holds a replica of the necessary object. When a site does not access a shared data object anymore, e.g. the local user changed his working focus, discard the shared object and thereby reduce the communications costs that are necessary for consistency. Otherwise the set of replicas continually grows and finally the shared data object is replicated to all participating sites.

Rationale As a user only maintains a replica for those shared objects, she currently uses, the communication costs compared to REPLICATE FOR SPEED$_{\rightarrow 2.1.2}$ are reduced. However, as a user can still locally change and access the data she is working with, the response time of the interactive application is reduced.

Danger Spots

– As the working style of a user is not deterministic, this pattern can also lead to high communication costs. For instance, consider a user who just takes a short look on a document part and then continues his work on a different document part. In the worst case, the runtime system would request the shared data object and almost immediately discard it.

  For a more sophisticated approach, use heuristic approaches to guess, if a user really needs a shared object or not. Wolfson et al. [48], [49], [50] introduce several distributed algorithms for an adaptive replication. The algorithms use different cost-functions to adapt the replication scheme of a shared data object. The cost functions base upon the read-write pattern on the shared data object. They also show that their adaptive replication algorithm significantly reduces the network traffic.

– You might not know a site that already holds the replica you are looking for.

– Requesting and transferring the state of a replicated object is complicated, as you have to ensure that the transferred object is consistent to all other replicas. There exist different approaches to solve this problem, compare [26] for a starting point.

Known Uses **DreamObjects** [29] offers two predefined adaptive distribution schemes. These distribution schemes dynamically change the distribution of a shared data object according to a user's working style, i.e. how often a user accesses a shared object, or according to the topology of the connecting network, e.g. one replica per subnet.

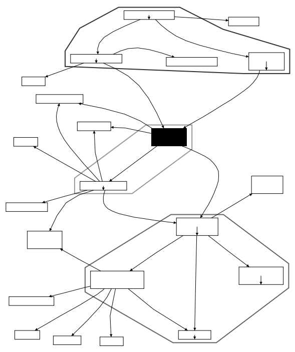
Mozilla is a popular Open Source web browser that uses link prefetching [13] to reduce latency for the user when visiting a web site. The prefetching mechanism depends on the origin server or an intermediate proxy server. One of these servers, has to determine the set of documents which the browser shall prefetch. When prefetching a document, a browser stores the document in its local cache. Thereby, a user can access the document without network latency. Though the server specify which documents the user will probably access next, the browser uses its own heuristics to determine when to prefetch the specified documents.

✣ ✣ ✣

Related Patterns REPLICATE FOR SPEED$_{\rightarrow 2.1.2}$: The REPLICATE FOR SPEED pattern increases the network communication costs that are necessary for data consistency. However, as a user receives a replica for each shared object as soon as he joins the collaboration, its site can contact an arbitrary site to request a replica.

2.2 Change Notifications

2.2.1 Update Your Friends



Intent — Distribute local state changes to the other users to achieve consistency.

Context — You have developed support that lets users access replicated objects (c.f. Replicate for Speed$_{\rightarrow 2.1.2}$ and Replicate for Freedom$_{\rightarrow 2.1.3}$). Now you are thinking about effects of changes to these objects.

✣ ✣ ✣

Problem — **Users change their local copies of the replicated artifacts and the other users cannot notice these local changes. This makes collaboration impossible.**

Scenario — Consider, e.g., a collaborative diagram editor where all diagram elements are represented by a replicated object. Users collaborate in the diagram editor to specify the class diagram of a new project they are working on. Each user locally modifies the class diagram by adding new classes or changing the interface definition. As these changes are not propagated, each user has a different class diagram at the end of the collaborative session. After implementing their part of the class diagram, the users try to integrate their work and wonder why this is not possible without a lot of errors.

Symptoms — *You should consider to apply the pattern when ...*

- users collaborate by sharing and changing replicated artifacts.
- the state of the replicated object diverges, as users locally change their replicas.

Solution — **Therefore: After changing a replicated object locally**

- **send an update message for this object to all clients that also maintain a replica,**
- **take care that all clients receive this update message, and**
- **let these clients change their replica according to the information in the update message.**

Collaborations

Fig. 2 shows how the network communication works. A client modifies its replica and distributes an update message to all clients who also hold the replica. This update message may contain the new object state or the transformation that led from the old to the new state. Both cases are equivalent regarding update distribution but different regarding consistency management. After distributing the update message, the sending client collects acknowledgements from the receiving clients to ensure that each one received the update message.

To know who has to receive the update each client has to maintain a list of these clients. The time to which the update message is sent depends on the connectivity of the clients.

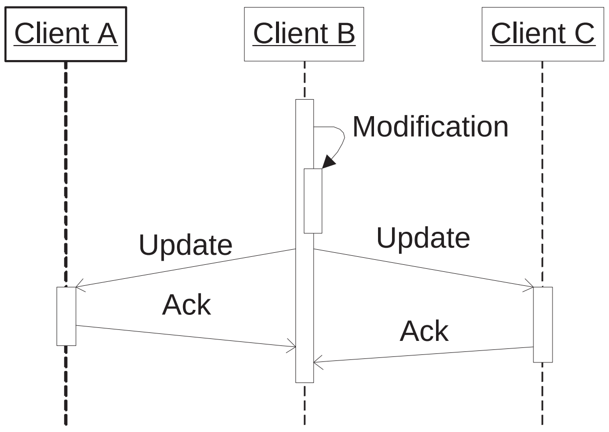


Figure 2: A client distributes update messages to all other clients

Rationale

Since all clients sooner or later receive an update message, they can change the state of their replica to the new state.

Danger Spots

- You might not know all your friends. In this case [9] proposes to use a flooding technique where the informed friends inform their friends. The latter is, e.g., done in Gnutella.
- Update messages might get lost and thus not all your friends get to know the update. One possibility to overcome this problem is to use sequence numbers in the update messages. Each client tracks the received sequence numbers and detects a missing update message by a gap in the received sequence numbers. In that case the client has to ask for the missed update message or request a new replica.
- If users change the state at the same time, there can be a conflict.
- If network bandwidth is an issue, distribute the actions describing the state change instead of the new object state to reduce the network load. Be careful as this might not be true in every case, e.g. actions may need arguments larger than the whole object state.
- If the execution of a state-changing action is more time-consuming than transmitting the whole object state to the other clients, distribute the new object state as update message.

– If clients are disconnected, use push- and pull-phases [9].

Known Uses

GroupKit [38] transparently replicates an a programming abstraction called environment. Whenever users change the value of a node in the shared environment, the runtime system automatically distributes the new content of the node to the other participating sites. Additionally, GroupKit supports a multicast RPC like described in [7]. By using the multicast RPC developers can distribute local changes to all other participating sites.

DreamObjects [29] supports a more sophisticated multicast RPC as described in [7] to achieve consistency. In DreamObjects each site maintains for each shared object a list of sites that hold a replica of the shared object. Based on these lists the runtime system of DreamObjects distributes the necessary update messages. These update messages describe the change and allow each receiving site to re-execute the change.

USENET uses a flooding mechanism to exchange messages between the different machines [21]. A message is posted on one machine to a list of newsgroups. This machine accepts it locally, as if applying a local change, and then forwards it to all its neighbors. The neighbors check if they are really interested in the new message and then apply the change. Furthermore, the neighbors forward the new message to all their neighbors. By using this technique, a site can apply a change without knowing all sites that are interested in the change. It is only necessary to now some sites. However, there are two further danger spots. First, to reduce network load loops must be avoided. Second, it has to be ensured that still all sites can be reached.

✣ ✣ ✣

Related Patterns

Replicate for Speed$_{\rightarrow 2.1.2}$: As users can locally change their replicas, it is necessary to inform the other participating sites to keep the shared data consistent. The Update Your Friends pattern can be used for this purpose.

Replicate for Freedom$_{\rightarrow 2.1.3}$: If users are allowed to modify their local replicas, the Update Your Friends pattern can be used to distribute the modifications and keep the shared data consistent.

Mediated Updates$_{\rightarrow 2.2.2}$: In Mediated Updates all changes are distributed by one site. Compared to this, the Update Your Friends pattern does not have a single-point of failure, is not a bottleneck, and the communication costs are lower, which reduces the response time of the application.

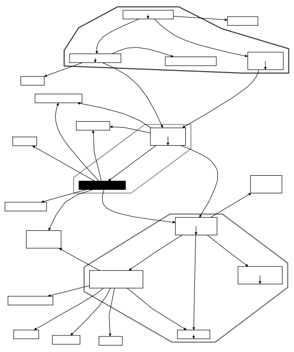
Believe in Your Group$_{\rightarrow 2.3.1}$ provides a way to optimistically ensure consistency when different users modify same parts of the shared data at the same time.

Acknowledgement [39]: This pattern can be used to model the acknowledgement interaction between the update receivers and the originator.

Roll Forward [39] discusses how to replay changes on replicated data. It uses a replication at two sites. One site performs the change and the other site replays the change after receiving an update message. If the first system crashes during the update, the second system will still be operational and thus not be affected by the illegal change of the first client. In the context of collaborative applications, this behavior is also desirable. If one user performs changes that cause inconsistent data, other clients will not replay these changes because the first client will no longer be able to update his friends (if it crashes before sending the updates).

2.2.2 Mediated Updates

Alternative names: Update dispatcher



Intent

Minimize the administrative load for updates by using a central instance that dispatches the updates.

Context

You have considered Update Your Friends$_{\rightarrow 2.2.1}$ to inform other interested clients about state changes to replicated objects. Now you want to reduce the administrative overhead for ensuring consistency and propagating update notifications.

✣ ✣ ✣

Problem

Clients want to propagate update messages to other clients who keep replicas of the same data. If they contact the other clients directly, they have to maintain information who those clients are and have to establish communication with these clients. This is complicated and error-prone. Especially if some clients may disconnect and reconnect in an unpredictable way (if the set of clients changes over time).

Scenario

Consider the German telephone network. Each year the German telecommunication company distributes phone books containing the actual telephone numbers of their customers.

Symptoms

You should consider to apply the pattern when …

- the state of the replicated objects often diverges because of missed update notifications.
- not every client knows who all the other clients are.
- it is hard to manage the set of interested clients; especially when the set of clients changes frequently.

Solution

Therefore: After changing a replicated object inform a mediator which will distribute an update message to all interested clients.

Collaborations

As shown in fig. 3, a user modifies her replica and sends an update message to the mediator. The update message may contain the new object state or the transformation that led from the old to the new state. Both cases are equivalent regarding update distribution but different regarding consistency management. The mediator maintains a list of all clients that are interested in the replica and redistributes the update message to these clients. As in the Update Your Friends$_{\rightarrow 2.2.1}$ pattern, the mediator collects the acknowledgements from the other clients and afterwards acknowledges the changing client. To reduce network communication, the interaction can also work without the acknowledgements of other clients (as shown in fig. 3). In this case, the mediator only propagates update messages if the update can be applied to the mediator's current state. At the same time, the mediator sends an acknowledgement to the initiating client.

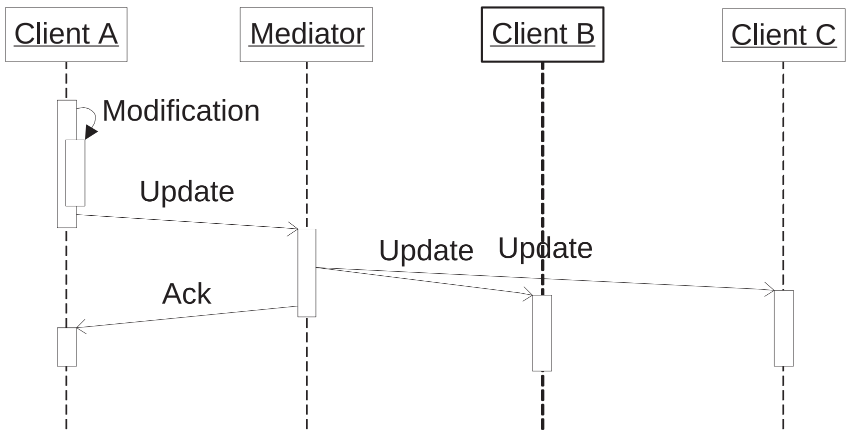


Figure 3: Distributing updates via a mediator

Rationale

As the mediator knows who has to receive the update and distributes the update to these clients, the clients do not have to deal with these issues any longer (cf. Update Your Friends$_{\rightarrow 2.2.1}$). The client list of the mediator is always valid, as the clients have to register themselves at the mediator for maintaining a replica.

Since the mediator is informed about all changes, it is also the preferred point for asking for the most recent version of each replicated object. Thus, clients that disconnect or enter a collaborative session after other clients already changed the data (these clients are called latecomers) are provided with a method for retrieving the most current state. In architectures that work without a central mediator, this is impossible (cf. the discussion in Update Your Friends$_{\rightarrow 2.2.1}$).

Danger Spots

- The mediator is a single-point of failure.
- The mediator is a bottleneck concerning the network communication.
- If users change the state at the same time, there can be a conflict, which should be resolved by the mediator.

– If clients disconnect, the mediator has to ensure that they receive all updates that were distributed during their absence.

Known Uses

COAST [41] uses possibly multiple mediators to ensure consistence on shared data. In COAST clients register at a mediator when they want to obtain replicas of shared data. The mediator keeps track of the clients that keep replicas and are connected to the network (if they get disconnected, they have to reconnect and obtain a fresh copy of the data).

Whenever the client changes replicated data (using transactions), a transaction log is sent to the mediator. The mediator then decides, whether or not this transaction can be incorporated with the current state. If this is possible, the master copy on the mediator is updated and the transaction log is broadcasted to all other connected clients that hold replicas of the object.

The mediator is also responsible for storing the shared objects.

DyCE [46] provides a central object manager which maintains the persistent object storage and handles shared objects. Developers have to define transactions to modify a shared data object. The transaction manager at the server acts as a mediator and is responsible for handling and distributing transactions.

Habanero [5] focuses on transforming Java applets into distributed applets (called *Hablet*). The state of such a hablet is replicated to every participating site. To keep the shared state consistent, Habanero intercepts user interface events and forwards them to a well-known server that acts as mediator. The server serializes all events and forwards them to all clients.

✣ ✣ ✣

Related Patterns

REPLICATE FOR SPEED$_{\rightarrow 2.1.2}$: As users can locally change their replicas, it is necessary to inform the other participating sites to keep the shared data consistent. The MEDIATED UPDATES pattern can be used for this purpose.

REPLICATE FOR FREEDOM$_{\rightarrow 2.1.3}$: If users are allowed to modify their local replicas, the UPDATE YOUR FRIENDS pattern can be used to distribute the modifications and keep the shared data consistent.

UPDATE YOUR FRIENDS$_{\rightarrow 2.2.1}$: The costs for maintaining the client lists are reduced. At a first glance the network load of UPDATE YOUR FRIENDS is lower than the load of MEDIATED UPDATES since communication with the mediator is not necessary. On the other hand if clients change frequently and therefore the client lists, additional network load is produced.

BELIEVE IN YOUR GROUP$_{\rightarrow 2.3.1}$ provides a way to optimistically ensure consistency when different users modify same parts of the shared data at the same time.

MEDIATOR [14]: The MEDIATED UPDATES pattern implements a distributed MEDIATOR. As in the MEDIATOR pattern, it decouples the single clients and transforms the mana-to.-many communication between the clients to an one-to-many/many-to-one communication between the mediator and the clients. It also helps to centralize control, which is very helpful when establishing consistency (e.g. with the DETECT A CONFLICTING CHANGE$_{\rightarrow 2.3.3}$ pattern). The main difference between the MEDIATOR] and the MEDIATED UPDATES is that the latter s more concretely focussing on the synchronization of replicated objects. It can thus be considered as a more concrete version of the MEDIATOR.

OBJECT MANAGER [3] describes how the use of an object can be decoupled from its life cycle management. The OBJECT MANAGER is responsible for retrieving the object, handing it out to clients, or for keeping it persistent. The mediator in the MEDIATED UPDATES pattern often plays the role of an OBJECT MANAGER.

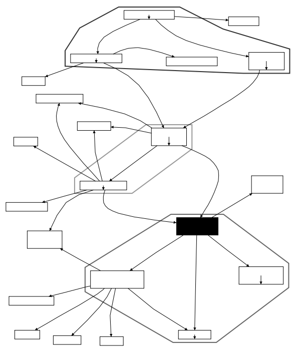
RECOVERABLE DISTRIBUTOR [22] describes how local views of global state can be kept consistent at different clients. It proposes a comparable interaction scheme. With this respect, it provides a combination of the MEDIATED UPDATES and DETECT A CONFLICTING CHANGE$_{\rightarrow 2.3.3}$ pattern. Such a combination makes the design of collaborative applications less flexible. Especially the focus on only one distribution scheme narrows the applicability.

ACKNOWLEDGEMENT and ROLL FORWARD [39] relate to MEDIATED UPDATES as it was discussed in the UPDATE YOUR FRIENDS$_{\rightarrow 2.2.1}$ pattern.

2.3 Consistency

2.3.1 BELIEVE IN YOUR GROUP

Alternative names: Optimistic transactions [20]



Intent — Change data optimistically and undo the operations if users performed conflicting changes.

Context — Your system allows the users to work on replicated objects. You have managed to propagate state changes (c.f. UPDATE YOUR FRIENDS$_{\rightarrow 2.2.1}$ and MEDIATED UPDATES$_{\rightarrow 2.2.2}$). Now you are thinking about how to ensure consistency.

✣ ✣ ✣

Problem — **You want to ensure consistency but you do not have much time, until you can perform these changes.**

Scenario — Imagine the users of diagram editor where the diagram objects are replicated. Before a user can change a diagram object, she has to obtain a lock to ensure consistency. This is time-consuming and increases the response time of the diagram editor. As the response time is a critical issue in interactive applications, the diagram editor becomes unusable.

Symptoms — *You should consider to apply the pattern when ...*

- the system ensures consistency but is terribly slow.
- acquiring locks is too time-consuming.
- the application domain is suitable for controlling concurrent interaction by means of a social protocol.

Solution — **Therefore: Perform the change immediately. If another client performed a conflicting change earlier (cf.** DETECT A CONFLICTING CHANGE$_{\rightarrow 2.3.3}$**), undo or rearrange your change.**

Collaborations — A client changes a replicated object and informs the other clients by using the UPDATE YOUR FRIENDS pattern. The client continues its work and waits either for an acknowledgement or a reject. If its change is rejected by the other clients, it has to undo or rearrange its change.

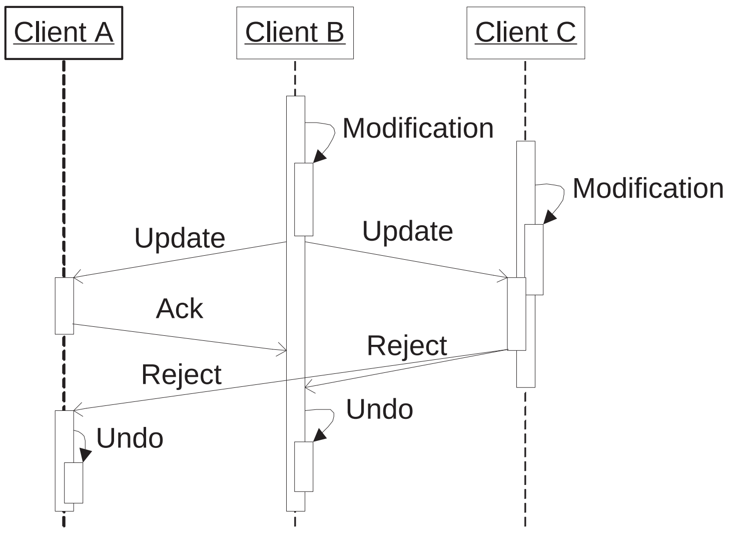


Figure 4: Distributing an update message resulting in an undo

Fig. 4 provides an example of three interacting clients. Client B and Client C perform modifications at almost the same time. After Client B UPDATED HIS FRIENDS, Client C finds out that B's modifications are conflicting. Now assume that Client C is authorized to reject B's changes. It informs all other clients that B's modification has to be rejected, which causes undo operations at the clients A and B. Note that client A also has to undo the change since it already integrated B's update with its replicas of the shared data.

Rationale

The client avoids waiting in two cases:

1. Before performing its change.
2. After performing the change and collecting the acknowledgements.

Danger Spots

- The optimistic approach can confuse a user by rolling back or transforming a modification.
- The implementation of a roll back mechanism is very memory intensive, as the state of an application has to be stored. Sometimes it can even be impossible or at least very expensive to define the rules that are needed for the transformation.
- Rejecting takes time.
- To reduce the amounts of rollbacks, you may use the local lag paradigm [30].

Known Uses

COAST [40] allows clients to perform their changes of shared objects directly on their replicas. If a client needs to change a replica, it starts a transaction and modifies the object using COASTs accessing methods. The transaction manager monitors each access and remembers it on a transaction log. When the transaction is finished, it is sent to the mediator, a server that decides, whether or not the transaction may survive. In the meantime, the client updates all dependent views. If the transaction was accepted by the server, the client has nothing more to do. Actually, the client will have been able to continue with the application while the server was still deciding whether or not to accept the transaction.

If the transaction was not accepted (i.e. if it failed the DETECT A CONFLICTING CHANGE$_{\rightarrow 2.3.3}$ test), then the client has to restore the change using the old values that were also recorded in the transaction log and updating all dependant views again.

DyCE [47] uses a comparable transaction mechanism with optimistic transactions as COAST.

DOORS [35] allows independent changes to model objects replicated between clients without requiring a central server. Whenever a client changes a model object, it remembers the change and propagates it to other clients as soon as it meets them. The decision whether or not changes are acceptable is made by each client.

GINA [1] enables synchronous collaboration on replicated objects. It understands the history of a document as operation tree and uses the tree for undo/redo mechanisms, optimistic concurrency control, and object merging.

✣ ✣ ✣

Related Patterns

DETECT A CONFLICTING CHANGE$_{\rightarrow 2.3.3}$: An old Russian saying[1] says: Trust, but control. This means that even with a high level of trust a group can produce conflicts. These conflicts have to be detected, which is the intent of the DETECT A CONFLICTING CHANGE pattern.

LOVELY BAGS$_{\rightarrow 2.3.4}$: It is crucial to model the shared objects in a way that allows many concurrent accesses. Otherwise, the BELIEVE IN YOUR GROUP pattern will result in too many changes that need to be undone. The LOVELY BAGS pattern provides one example of how to model the shared objects this way.

DON'T TRUST YOUR FRIENDS$_{\rightarrow 2.3.2}$: Even when most operations can be performed in an optimistic way, there may be some changes that are hard to undo. Especially changes with large side-effects can be expensive (or impossible) to undo. In these cases, you should perform these changes using locks.

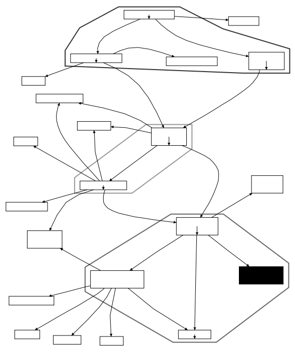
OPTIMISTIC TRANSACTION [20] explains how to optimistically perform concurrent changes. It proposes to perform the changes on a shared resource and test, whether other clients performed changes at the same time. For testing, this pattern uses a change counter that is compared to a change counter value expected for the case in

[1]Lenin is often referred to as the origin of this saying. However, he was not the author. The saying was just one of his favorite sentences.

which only the changes of the local client had appeared. The main difference to the BELIEVE IN YOUR GROUP pattern is that the OPTIMISTIC TRANSACTION pattern does not address the context of replicated objects. This makes the detection of a conflicting change through change counting simpler. If this pattern would be applied in a distributed setting, the change counters would have to be kept consistent which on its own results in the management of shared data in a replicated context.

2.3.2 Don't Trust Your Friends

Alternative names: Pessimistic locking



Intent

Allow only one user at a time to change a specific part of the shared state.

Context

You are working with replicated objects and users may change these objects at the same time. Now you are thinking about avoiding any inconsistencies, as they can appear when using Believe in Your Group$_{\rightarrow 2.3.1}$.

✣ ✣ ✣

Problem

You want to ensure that nobody messes up your changes on replicated data objects, but many users are working with the same objects at the same time. Each site performs its changes locally before informing the other sites. This can lead to different execution orders of the changes. If the changes are not commutative, i.e. changing their execution order does not lead to the same shared state, the shared state becomes inconsistent.

Scenario

Consider, e.g., a collaborative CAD application. Users collaboratively work in this application to design a new engine. They can drag and drop parts of the engine and change the specification of the single parts. Now imagine that one user decreases the cylinder capacity of the engine while another wants to move the cylinders into place. Depending on the execution order at the different sites, either the CAD application rejects the first or second modification. Some users might see an engine with a lower cylinder capacity and others an engine where the cylinders are in their place.

Symptoms

You should consider to apply the pattern when ...

– undoing inconsistent changes is hard, harder, or even impossible. An example is the situation where changes rely on external events.

– the lack of a social protocol causes synchronous and conflicting changes.

– you have made bad experiences with inconsistencies in your sensible shared data (e.g. shared objects used to control the position of fuel rods of a nuclear power plant).

Solution **Therefore: Let a site request and receive a distributed lock before it can change the shared state. After performing the change let the site release the lock so that other sites can request and receive it for changing the shared state.**

The lock can have different grain sizes. The grain size of a lock determines how much of a shared data object or all shared data objects can be modified after getting one lock.

Collaborations The different sites interact according to fig. 5. A site that wants to change the shared state, i.e. the content of the replicated objects, first has to request a distributed lock. One strategy for obtaining the lock is by means of a central lock manager. In this case, the lock manager grants the lock to the requesting client. After receiving the distributed lock, the site can perform its change and distribute the change to the other participating sites. Finally, the site releases the distributed lock. If other sites also requested a lock while another site held the lock, they have to wait until the lock is available again (client B in fig. 5).

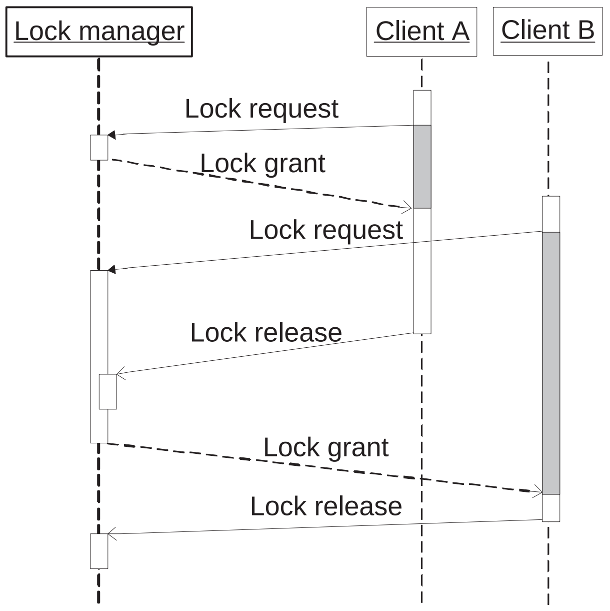


Figure 5: Requesting a lock with a central lock manager

There are many strategies to implement distributed locks [6]. The central lock manager described above is only one of these strategies. Selecting the distributed locking algorithm has an impact on the user interface. When the participating sites are, e.g., connected via TCP/IP, the use of a token-based algorithm, e.g. [45], reduces the number of exchanged messages and thereby also decreases the response time of an application. If the participating sites are, e.g., connected via a network that supports IP multicasting, an algorithm that bases on multicasting messages, e.g. [36] is the proper choice.

Rationale By requesting and receiving a distributed lock before performing a change to the shared state, only one site at a time can change the shared state. This prevents inconsistencies of the shared state.

Danger Spots

– Requesting and receiving a distributed lock before changing the shared state can increase the response time of an application. Interactive applications demand a low response time. Therefore, be careful when using distributed locks for concurrency control.

– It is a difficult issue to set the grain size of a lock. A coarse grain size, e.g. all shared objects, reduces the number of lock requests, but also the concurrency in the application. A fine grain size, e.g. for each modifying method, increases the number of lock requests and the network traffic, but improves the concurrency in the application. Greenberg et al. [17] discuss these issues concerning its effects on the user interface.

– If a user needs more than one lock to perform its changes, there is the danger of deadlocks. So, you have to use deadlock avoidance strategies, as, e.g., by totally ordering the locks that have to be requested [19].

Known Uses

DreamObjects [29] supports two different grain sizes for concurrency control. Developers can use one distributed lock per shared objects. Whenever a site wants to change a shared object, it first has to request and receive a distributed lock. Otherwise, the runtime system prevents changes to the object. To achieve more concurrency, developers can specify sets of methods that must be executed mutually exclusive, e.g. methods that change same parts of a shared object. For each set of methods DreamObjects uses one distributed lock that must be requested and received by a site before executing a method in the set. Requesting and receiving the locks are automatically handled by the runtime system.

DistView [34] divides an application into interface and application objects. Both are completely replicated. DistView intercepts all calls to these objects and broadcasts them to the respective replicas to synchronize the states of the shared windows. To ensure consistency DistView associates with each replicated application object one distributed lock. Developers explicitly have to request these locks before changing an application object.

✣ ✣ ✣

Related Patterns BELIEVE IN YOUR GROUP$_{\rightarrow 2.3.1}$: If the response time of the application is a critical issue, this pattern can be used to perform changes in an optimistical way. However, if a conflict occurs, inconsistencies can happen that can be hard to resolve.

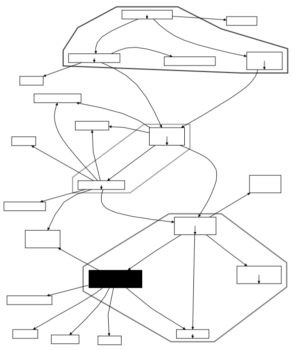
Selective Locking [31] is a pattern language for locking in parallel programs and examines constraints, e.g. memory latency or size, for this scenario.

Permit based Locking [44] is a design pattern for requesting and receiving locks which aims to reduce network traffic. For this purpose, it uses a central lock server that passes permits to lock to those clients which are guessed to need the lock next. If a client has a permit to lock, it need not request the lock and can perform its change immediately. Otherwise, it has to request the permit from the central lock server. If the server already distributed the requested permit, it revokes this permit. This pattern lacks from the central lock server which is a single-point of failure. Additionally, if the clients often need different locks requesting and revoking permits also produces network traffic.

Coordinator [23] uses a two-phase commit protocol to ensure that all clients perform a change. The Don't Trust Your Friends pattern uses a locking approach to ensure that only one client performs a change at a time.

Resource Lifecycle Manager [25]: The lock can be interpreted as a resource in the sense of the Resource Lifecycle Manager. It is a very simple resource that only controls the right for modifying shared objects. Thus, a lock does not need a resource environment as described in the Resource Lifecycle Manager.

2.3.3 Detect A Conflicting Change



Intent — Detect conflicting changes.

Context — You have managed to distribute state changes and still Believe in Your Group$_{\rightarrow 2.3.1}$. Now you are worried about conflicting changes.

✣ ✣ ✣

Problem — **If two or more users change the same data at the same time, changes interfere. This can lead to inconsistent data or contradict the users' intentions. If the users are not aware of this conflict, they will no longer have a common base for collaboration.**

Scenario — Remember the times when you were playing jigsaw puzzles. The goal of this game was to rearrange a scrambled picture by moving single pieces of the puzzle to the right position. This game can be played cooperatively by allowing multiple users to move the pieces.

Now consider an electronic version of the game: Since the puzzle is an interactive game, all players receive a replica of the shared objects that model the puzzle pieces (replicate for speed). Now the players start moving around the pieces. After each move, the player's client informs all other clients that the piece was moved. The other clients then update the position of the piece.

This is unproblematic, if the users perform the changes at different points in time so that the system finds time to receive other users' changes before the local user performs a new change. But now imagine that user u_1 moves piece p_1 to the lower right corner and at the same time user u_2 moves the same piece to the upper right corner.

After the client of u_1 has performed the change, it notifies the client of u_2. Since the client of u_2 has already performed its change (moving the piece to the upper right corner), it will replay the other change (moving the piece to the lower right corner) after it was informed by u_1's client. Thus, u_2 will see the piece positioned at the lower right corner.

Now switch perspectives and see what happened at the client of u_1. It has already performed the move to the lower right corner, when it receives the information from u_2's client that the piece should move to the upper right corner. Thus, it moves the piece to the upper right corner. u_1 will see the piece at the upper right corner.

Thus, both users will see the piece at different positions, which is obviously a problem, especially, when they are communicating about the pieces (e.g. when u_1 asks u_2 in a chat to move the piece at the upper right corner to the center, u_2 will take another piece than u_1 intended).

Symptoms *You should consider to apply the pattern when ...*

- users perform parallel changes that lead to inconsistent data.
- the social protocol does not ensure consistency.
- each user works on his own inconsistent view of the shared data. This means that users are talking at cross purposes.

Solution **Therefore: Let each client remember all changes that have not yet been replayed by all other clients. Whenever a change is received from another client check it against those changes that have not yet been replayed by the other client and affect the same shared object. If the performed operations will produce a conflict then undo one of these changes.**

Collaborations Each client that should detect conflicts has to maintain a history of all changes that it performed up to now and note for each change, whether or not this change was perceived by other clients. If the change has been perceived by the other clients then it can be removed from the change list, since no future conflict is possible regarding this change.

Clients perform changes locally first and UPDATE THEIR FRIENDS directly after applying the change. The other clients receive the update information and check their local change history, whether or not other activities (not perceived by the client that sent the update notification) have changed the same data.

Concurrent changes on the same data are potential candidates for conflicts. The easiest way of deciding, whether or not it was a conflict is to consider all changes that are modifying the same data as conflicting changes. In general, one can consider two changes as conflicting changes if they produce a different application state if they were performed in different order. This means that the changes are not commutative. The more detailed the test for commutativity is, the more concurrent changes can be allowed in the application.

Consider a linked list as an example for a shared object. One could decide to consider all adds to this list (`addFirst`, `addLast`) as conflicting changes. One could also define `addFirst` and `addLast` as pairwise commutative since an `addFirst` operation will not affect the position of the element added with the `addLast` operation. On the other hand, two concurrent `addLast` operations are never commutative since they both affect the end of the list.

In a final phase, the client decides to undo one of the conflicting changes or inform the user that the data is inconsistent (in some cases, undoing is an expensive operation and small inconsistencies can be accepted). Which change is undone depends on the applied algorithm. It only has to be ensured that each client undoes the same change. In a system with MEDIATED UPDATES$_{\rightarrow 2.2.2}$, it is often the second change that reached the mediator.

The way how the change is undone depends on the way how changes are represented. In cases where new state is propagated, the undo operation may be the replacement of the changed data with the state that was present before the change. This implies that the old state is remembered at each client. In cases where the change is a command, it may be undone by executing an inverse command.

Rationale

The change history allows each client to detect whether or not conflicting changes have been applied on the same artifact.

Danger Spots

The process of undoing a change is often complicated again. But can also be easy, if the change that was received from the other client needs to be undone since this change has not yet been replayed by the local client. The related pattern provides a link to the command pattern for more information regarding undo.

Remember to forget those changes in the change history that have been perceived by all other clients. To know what these changes are, a client can include information on its perceived changes in its change notifications sent to the other clients (using a *Piggybacking* scheme). The more up-to-date the list of perceived changes is, the less changes will be stored in the change history. This leads to shorter execution times for detecting a conflict and less conflicts.

If the number of clients gets very large, the management of history lists can become a major bottleneck of the whole system.

Ensure that no additional changes have been applied to an object before undoing a conflicting change. Otherwise, the undo operation may lead to an inconsistent state.

Known Uses

COAST detects changes at the mediator as well as at the client. The mediator (c.f. MEDIATED UPDATES$_{\rightarrow 2.2.2}$) detects a change when one client has based its change on data that has been changed by other clients in between. In this case, the mediator forgets the change that was received at a later point in time. The other clients will not be informed about the conflicting change. The client that initially performed the conflicting change will receive a change message for the same object from the mediator indicating that the change from the mediator is based on an old state. Thus, the client will undo all changes that were performed locally back to the state before the conflicting change. Since each client only has to detect changes from its local execution and the (valid) changes from the mediator, the management of the state only has to consider two parties, which makes it simple and fast.

✣ ✣ ✣

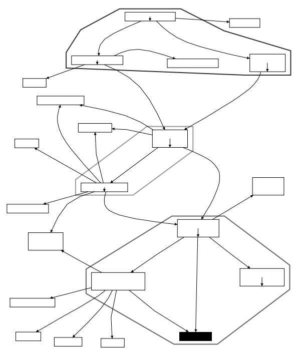
Related Patterns LOVELY BAGS$_{\rightarrow 2.3.4}$ can be used to reduce the probability of conflicting changes by modeling the data in a way that is suitable for concurrent changes.

ROLLBACK ([27], p. 163) discusses, how a client can return to a consistent state after a conflicting change has been detected. Basically, the old state of the system has to be recorded before the change (using the MEMENTO ([15], p. 283)) and is restored within the ROLLBACK. Besides the MEMENTO one can also utilize the COMMAND pattern with undoable actions for returning to an old state ([15, p. 233]). The ROLLBACK pattern has been transferred to the context of replicated systems [39].

RECOVERABLE DISTRIBUTOR [22] combines the current pattern with the MEDIATED UPDATES$_{\rightarrow 2.2.2}$ pattern. A discussion, how the RECOVERABLE DISTRIBUTOR relates to the current pattern is provided in the related Patterns section of MEDIATED UPDATES$_{\rightarrow 2.2.2}$.

CHANGE WARNING [42] reduces the risk of running into conflicting changes by visualizing another user's change as soon as it happens (without necessarily replaying the change).

2.3.4 Lovely Bags



Intent

Use bags for storing shared objects in a container because they provide the best concurrency behavior.

Context

You are using mechanisms to Detect A Conflicting Change$_{\rightarrow 2.3.3}$. Now you are thinking about reducing the number of conflicting changes.

✣ ✣ ✣

Problem

Access operations to shared container objects change the content of the container by adding or removing elements. Most of these operations are very bad regarding concurrency. Thus, synchronous collaboration on container objects often seems impossible.

Scenario

Imagine a chat system that was implemented on the basis of replicated objects. The easiest way of modelling a chat would be to have one long string to which the users append their contributions.

Now imagine that two users send a contribution at the same time. While the first client has appended its contribution (and sent out his update message to the second client), the second client already added its own contribution to its representation of the chat log. Thus, it will add the first client's contribution after its own contribution. And the first client will add the second client's contribution after its own contribution. Thus, both clients will see a different value for the chat log (which implies that one of the changes has to be undone to ensure consistency).

Would it be better, if the clients used a list where they stored the individual contributions? No, because again both clients would try to access the same position in the object (the end) at the same time. This would be true for all ordered objects such as Lists, Vectors, Arrays. Whenever two clients try to modify an ordered collection in a way that affects the positions of newly added elements, these accesses cannot be performed concurrently.

Symptoms

You should consider to apply the pattern when ...

- access operations to container objects are often rejected since two concurrent changes cannot be executed in different orders.

Solution **Therefore: Wherever a high level of concurrency is needed model your container objects by means of a bag. If the container's entries need to be ordered equip the data entries with an order criterium that can be uniquely assigned by each client (e.g. the current time stamp together with a unique client ID) but still store the entries in the bag.**

Collaborations The main participant is the bag. The bag is a shared container object that can hold references to other objects. It allows duplicates and does not care about the order of the contained elements (from a mathematical point of view, bags are often called multisets). Clients perform operations on the bag concurrently. These operations are in most cases commutative (because of the ignored order and allowed duplicates). In cases, where the bag may only grow, one does no longer have to check for consistency since clients will never perform operations that are not commutative. In cases, where the clients also remove elements from the bag ensure that the clients DETECT A CONFLICTING CHANGE$_{\rightarrow 2.3.3}$.

When order is needed while iterating over the collection locally, it will be converted to an ordered collection before iterating it. This conversion requires that the contained elements provide one or more attributes that can be used as a sorting criteria.

Rationale The simple explanation why this pattern works lies in the "lovely" nature of bags: A bag does not care about order and contained elements in the same way as other container classes do.

Compared to an ordered container object where the add operations are related to an insertion position (e.g. arrays or lists), bags produce the same result if two elements are added in different order.

Compared to container objects where the add operations depend on the current set of included objects (e.g. sets or dictionaries), bags produce the same results if an element was already present in the bag and two clients perform an add and a remove of this object concurrently.

This is illustrated in figure 6. In the left part, two users collaborate on a list. The list has the initial state `abc`. Then $User_1$ adds `d` while $User_2$ adds `e` at the same time. After both clients have performed their operation, they find time to UPDATE THEIR FRIENDS. But since the $User_1$'s state differs from $User_2$'s state, they will get different states if they perform the updates. Thus, one operation has to be undone.

Now consider the right part of figure 6: again, the users performed changes and UPDATED THEIR FRIENDS. But now, the *add*-operations can be performed since adds do not depend on the object's state or the (non-existent) order of its elements. Both users will reach a consistent state after the updates.

In cases where order is needed, one can often restore this order. Consider for instance a sorted collection: This class ensures that the elements stored in instances of sorted collections will be stored according to the sorting order (or iterated according to the sorting order). The main rea-

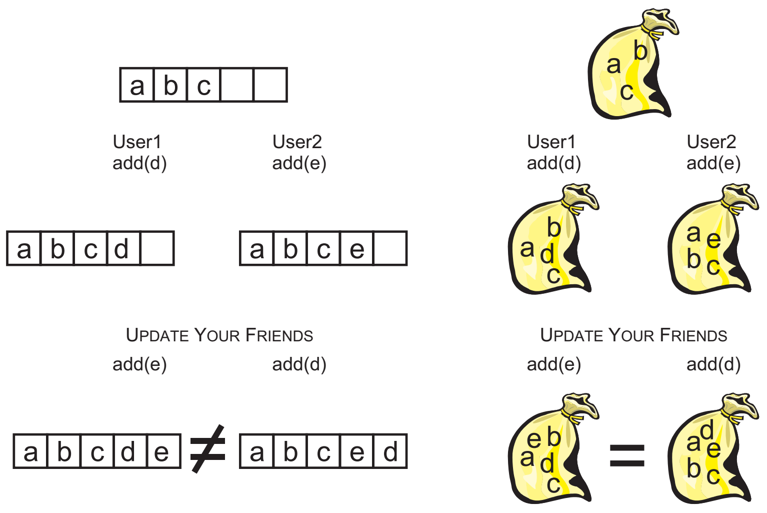


Figure 6: Concurrent accesses to lists and bags.

son for using such a sorted data structure is to speed up the process of iterating over the elements in a sorted way.

If a bag should be iterated in a sorted way, one can first convert it to a (local) sorted collection and then iterate over the local copy. The order (which drastically decreases possible concurrent operations on the data structure) is thus restored locally where needed. This makes the access to the ordered collection slower (and requires that each client sorts the elements) but it makes the data structure more robust for concurrent manipulations.

Another reason for storing objects in an ordered collection is the desire to access it by an index. The reason for this is often to speed up iteration again (by using a counter for the index when, e.g., iterating over an array in Java). As with the sorted access order, the iteration speed is less critical than the reduced concurrency. If the container object needs to be iterated frequently, you can perform the iterations on local copies of the replicated object, which are valid until the replicated object is changed again.

Danger Spots

Unfortunately, even the lovely bag is still vulnerable regarding remove operations. If the same element is removed and added at the same time and the bag did not include the element before, this will result in different states of the bag depending on the order of the operations. Thus, prohibiting removes could be an option.

In some cases, arrays can be as attractive (or even more attractive) as bags: when the number of elements in the bag does not change often and an entry of the array is accessed by its index without relating to the other elements, this change is concurrent to all changes at other array positions. But if the array should change its size (or have a shared index as a current index), these attributes will reduce the concurrency of the array.

Known Uses

Chat in FUB: FUB [18] is a system built on top of COAST for supporting brainstorming in the context of distributed collaborative learning. Thus, the users are provided with two different kinds of chats: a brainstorming chat and a discussion chat, where concepts are discussed. While the brainstorming chat does not require any order (and can thus be directly modelled using a bag), the discussion chat needs to ensure that all chat entries are shown in the same order for all users.

Thus, the chat is modelled as a set of chat entries. Each entry has a time stamp that represents the (synchronized) time when the entry was added at the client. For displaying the chat log, all entries are sorted with the time stamp as a primary and the contained text as a secondary key. This ensures that all entries are shown in the same order at each client.

Arrangement of Messages in Usenet [21]: Usenet newsgroups are semantically represented as trees of messages (modelling the reply-relations between messages). While these relations could have been explicitly modelled at the news server, the designers rather decided to hide the relations within the news entries.

Each entry has a unique id, which is determined by the client that generated the entry. The entry can relate to a parent message, while the parent message is not changed at all (it does not know about the child messages). All entries are then stored in an unspecified collection by the server. The important issue here is that the protocol does not demand for any order of the entries. When clients request entries they can ask for a sorted version (by time), which will then be generated (by inspecting the date fields of the messages).

The client is responsible for ordering the entries when displaying the message threads.

✣ ✣ ✣

Related Patterns

DON'T TRUST YOUR FRIENDS$_{\rightarrow 2.3.2}$: When the bag has to be reordered before the client can process it, an additional computation overhead is added. This may slow down the responsiveness of the application. If the reordering process takes more time than obtaining a lock as proposed in DON'T TRUST YOUR FRIENDS, you should prefer the locking mechanism.

DETECT A CONFLICTING CHANGE$_{\rightarrow 2.3.3}$: Even when using LOVELY BAGS inconsistencies can occur. DETECT A CONFLICTING CHANGE allows to detect these inconsistencies.

3 Conclusions

Among the main obstacles during groupware development is data sharing. Platforms that aim at simplifying the development process are often too prescriptive. When using a platform developers have to use a specific programming language or a specific class structure. Due to this, it is often hard to use a platform in a project that has different constraints.

This pattern language provides a set of proven solutions for the recurring issues during the development process. It allows developers to use these solutions in their intended context. The language concentrates on data sharing and keeping the shared data consistent. However, it only covers a small part of all low-level issues concerning groupware development. It, e.g., does not cover latecomer issues in synchronous groupware or shared data persistency. Therefore, there is still a lot of work to be done.

Acknowledgements: We would like to thank our shepherd Andreas Rüping for his excellent and challenging questions that helped to improve the patterns in this paper.

References

[1] Thomas Berlage and Andreas Genau. A framework for shared applications with a replicated architecture. In *Proceedings of the 6th annual ACM symposium on User interface software and technology*, pages 249–257. ACM Press, 1993.

[2] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[3] Frank Buschmann and Kevlin Henney. Explicit interface and object manager – two patterns from a pattern language for distributed computing. In K. Henney and D. Schütz, editors, *Proceedings of the Eighth European Conference on Pattern Languages of Programs (EuroPLoP'03)*, pages 207–220, Konstanz, Germany, 2004. UVK.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A system of patterns.* John Wiley and Sons, Chichester, West-Sussex, UK, 1996.

[5] Annie Chabert, Ed Grossman, Larry Jackson, Stephen Pietrovicz, and Chris Seguin. Java object-sharing in Habanero. *Communications of the ACM*, 41(6):69–76, June 1998.

[6] Ye-In Chang. A simulation study on distributed mutual exclusion. *Journal of Parallel and Distributed Computing*, 33:107–121, 1996.

[7] Eric C. Cooper. Replicated distributed programs. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 63–78, Orcas Island, Washington, USA, 1985. ACM.

[8] M. Crispin. INTERNET MESSAGE ACCESS PROTOCOL — VERSION 4rev1. Request for Comments 2060, IETF, December 1996.

[9] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *Proceedings of the 23rd International Conference on Distributed Computing Systems, ICDCS2003*, 2003.

[10] Prasun Dewan and Rajiv Choudhary. A high-level and flexible framework for implementing multiuser interfaces. *ACM Transactions on Information Systems*, 10(4):345–380, October 1992.

[11] C. Ellis, S. Gibbs, and G. Rein. Groupware - some issues and experiences. *Communications of the ACM*, 34(1):38–58, 1991.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. Request for Comments 2616, IETF, June 1999.

[13] Darin Fisher and Gagan Saksena. Link prefetching in mozilla: A server-driven approach. In *Eighth International Workshop on Web Content Caching and Distribution*, Hawthorne, NY USA, September 2003.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, pages 273–282. Addison-Wesley, Reading, MA, 1995.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[16] Hector Garcia-Molina. The future of data replication. In *Proceedings of the IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 13–19, Los Angeles, CA, USA, January 1986.

[17] Saul Greenberg and David Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work*, pages 207–217, Chapel Hill, NC, USA, 1994.

[18] Jörg M. Haake and Till Schümmer. Some experiences with collaborative exercises. In *Proceedings of CSCL'03*, Bergen, Norway, 2003. Kluwer Academic Publishers.

[19] J.W. Havender. Avoiding deadlock in multitasking systems. *IBM Systems Journal*, 7(2):74–84, 1968.

[20] K. Hendrikxs, E. Duval, and H. Oliv´e. Managing shared ressources. In Martine Devos and Andreas Rüping, editors, *Proceedings of the Fifth European Conference on Pattern Languages of Programs (EuroPLoP'2000)*, pages 411–430, Irsee, Germany, 2001. UVK.

[21] M. R. Horton and R. Adams. Standard for interchange of USENET messages. Request for Comments 1036, IETF, December 1987.

[22] Nayeem Islam and Murthy Devarakonda. An essential design pattern for fault-tolerant distributed state sharing. *Communications of the ACM*, 39(10):65–74, 1996.

[23] Prashant Jain. Coordinator. In Alan O'Callaghan, Jutta Eckstein, and Christa Schwanninger, editors, *Proceedings of the Seventh European Conference on Pattern Languages of Programs (EuroPLoP'02)*, pages 521–533, Irsee, Germany, 2003. UVK.

[24] Michael Kircher and Prashant Jain. Caching. In K. Henney and D. Schütz, editors, *Proceedings of the Eighth European Conference on Pattern Languages of Programs (EuroPLoP'03)*, pages 257–268, Konstanz, Germany, 2004. UVK.

[25] Michael Kircher and Prashant Jain. Resource lifecycle manager. In K. Henney and D. Schütz, editors, *Proceedings of the Eighth European Conference on Pattern Languages of Programs (EuroPLoP'03)*, pages 243–255, Konstanz, Germany, 2004. UVK.

[26] J. C. Lauwers and K. A. Lantz. Collaboration awareness in support of collaboration transparency: requirements for the next generation of shared window systems. In *CHI '90 Conference on Human Factors in Computing Systems, Special Issue of the SIGCHI Bulletin*, pages 303–311, Seattle, Washington, USA, April 1990.

[27] Doug Lea and Douglas Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[28] Bo Leuf and Ward Cunningham. *The Wiki Way.* Addison Wessley Addison Wesley, Longman, 2001.

[29] Stephan Lukosch. *Transparent and Flexible Data Sharing for Synchronous Groupware.* Schriften zu Kooperations- und Mediensystemen - Band 2. JOSEF EUL VERLAG GmbH, Lohmar - Köln, August 2003.

[30] Martin Mauve. Consistency in replicated continuous interactive media. In *Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work*, pages 181–190, Philadelphia, PA, USA, December 2000. ACM.

[31] Paul E. McKenney. Selecting locking primitives for parallel programming. *Communications of the ACM*, 39(10):75–82, 1996.

[32] Jonathan P. Munson and Prasun Dewan. Sync: A java framework for mobile collaborative applications. *IEEE Computer*, 30(6):59–66, June 1997.

[33] John F. Patterson, Mark Day, and Jakov Kucan. Notification servers for synchronous groupware. In *Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work*, pages 122–129, Boston, Massachusetts, USA, 1996.

[34] Atul Prakash and Hyong Sop Shim. Distview: Support for building efficient collaborative applications using replicated objects. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work*, pages 153–164, Chapel Hill, NC, USA, 1994.

[35] Nuno Preguica, J. Legatheaux Martins, Henrique Domingos, and Sergio Duarte. Data management support for asynchronous groupware. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 69–78. ACM Press, 2000.

[36] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communication of the ACM*, 24(1):9–17, January 1981.

[37] Mark Roseman. When is an object not an object? In *Proceedings of the Third Annual Tcl/Tk Workshop*, pages 197–204, Toronto, Canada, July 1995. Usenix Press.

[38] Mark Roseman and Saul Greenberg. Building real-time groupware with groupkit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction*, 3(1):66–106, March 1996.

[39] Titos Saridakis. A system of patterns for fault tolerance. In Alan O'Callaghan, Jutta Eckstein, and Christa Schwanninger, editors, *Proceedings of the Seventh European Conference on Pattern Languages of Programs (EuroPLoP'02)*, pages 535–582, Irsee, Germany, 2003. UVK.

[40] Christian Schuckmann, Lutz Kirchner, Jan Schümmer, and Jörg M. Haake. Designing object-oriented synchronous groupware with coast. In *Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work*, pages 30–38, Boston, Massachusetts, USA, July 1996.

[41] Christian Schuckmann, Jan Schümmer, and Till Schümmer. Coast - ein anwendungsframework für synchrone groupware. In *Proceedings of the net.objectDays*, Erfurt, 2000.

[42] Till Schümmer. Gama – a pattern language for computer supported dynamic collaboration. In K. Henney and D. Schütz, editors, *Proceedings of the Eighth European Conference on Pattern Languages of Programs (EuroPLoP'03)*, Konstanz, Germany, 2004. UVK.

[43] Till Schümmer and Robert Slagter. The oregon software development process. In *Proceedings of XP2004*, 2004.

[44] Dietmar Schütz. Permit based locking. In Andreas Rüping, Jutta Eckstein, and Christa Schwanninger, editors, *Proceedings of the Sixth European Conference on Pattern Languages of Programs (EuroPLoP'01)*, pages 347–359, Irsee, Germany, 2001.

[45] Mukesh Singhal. A heuristically-aided algorithm for mutual exclusion in distributed systems. *IEEE Transactions on Computers*, 38(5):651–662, May 1989.

[46] Daniel A. Tietze. *A Framework for Developing Component-based Co-operative Applications.* PhD thesis, Technische Universität Darmstadt, 2001.

[47] Daniel A. Tietze and Ralf Steinmetz. Ein framework zur entwicklung komponentenbasierter groupware. In R. Reichwald and J. Schlichter, editors, *Verteiltes Arbeiten - Arbeit der Zukunft (Proceedings der Fachtagung D-CSCW 2000)*, pages 49–62, Munich, Germany, September 2000. B. G. Teubner Stuttgart, Leipzig.

[48] Ouri Wolfson and Sushil Jajodia. An algorithm for dynamic data distribution. In *Proceedings of the 2nd Workshop on the Management of Replicated Data (WMRD-II)*, Monterey, CA, USA, November 1992.

[49] Ouri Wolfson and Sushil Jajodia. Distributed algorithms for dynamic replication of data. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS'92)*, pages 149–163, San Diego, CA, USA, June 1992.

[50] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, June 1997.