Designing object-oriented synchronous groupware with COAST

Christian Schuckmann, Lutz Kirchner[i], Jan Schümmer, Jörg M. Haake
IPSI - Integrated Publication and Information Systems Institute
GMD - German National Research Center for Information Technology
Dolivostr. 15, D - 64293 Darmstadt, Germany
+49 6151 869 952
{schucki, kirchner, schimmi, haake}@darmstadt.gmd.de

ABSTRACT
This paper introduces COAST, an object-oriented toolkit for the development of synchronous groupware, which enhances the usability and simplifies the development of such applications. COAST offers basic and generic components for the design of synchronous groupware and is complemented by a methodology for groupware development. Basic features of the toolkit include transaction-controlled access to replicated shared objects, transparent replication management, and a fully optimistic concurrency control. Development support is provided by a session concept supporting the flexible coupling of shared objects' aspects between concurrent users and by a fully transparent updating concept for displays which is based on declarative programming.

Keywords
toolkit, synchronous collaboration, groupware, replicated objects, sessions, display updating, concurrency control

INTRODUCTION
Groupware allows several geographically distributed people to work together with the aid of a computerized environment.

Requirements on the kind of cooperation support to be provided by a specific groupware application arise, for example, from the cooperation situation at hand (e.g., cooperation style, procedures, organization), the task, and the setting in which the cooperation takes place (e.g., geographic distribution, hardware and software basis, network connections).

In case of interactive synchronous groupware, three abstract requirements can be stated from the user's point of view:

- Since direct manipulation user interfaces require rapid feedback, synchronous groupware must ensure non-delayed processing of user actions independent of the network connection (e.g. latency).
- Synchronous groupware requires by definition a fast propagation of other users' operations to enable optimal group awareness under given network conditions.
- Since different cooperative situations require different aspects of a shared workspace to be coupled among concurrent users (e.g., to achieve private, loosely-, or tightly-coupled work), synchronous groupware needs to support dynamic session behavior. By session behavior, we mean the extent to which the shared workspace provides group awareness and allows to share (or to keep private) changes of the workspace's content.

Seen from the developer's point of view, building groupware applications is still a complex task. Although some progress has been made over the last few years with respect to the development of toolkits, architectures and the basic components of groupware applications, designing and implementing groupware applications is still a difficult and error-prone task. This problem grows with the size of the software to be developed. An abstract goal with respect to the development process is to reduce the complexity of groupware development to be comparable to the complexity of the development of single user applications.

Existing toolkits neither meet all of the listed user requirements nor meet the above development goal.

In order to make the development of a wide range of cooperative applications easier and more effective, a toolkit should have the following characteristics:

(1) offer a general architecture for groupware applications that is adaptable to different situations, tasks, and settings in a flexible way.

(2) provide basic building blocks and generic components which can be reused and refined for different concrete situations. Especially provide components for groupware functionality (e.g. session management, shared object management) which is normally not addressed by single-user applications and therefore difficult to handle for application developers.

(3) provide a methodology with rules and conventions about how typical functionality of applications has to be realized in terms of the toolkit.

Computer Supported Cooperative Work '96 , Cambridge MA USA
© 1996 ACM 0-89791-765-0/96/11 ..$3.50

Through the use of such a toolkit, code with fewer errors can be expected, tending to be more understandable, reusable and extensible.

In the OCEAN department, we are developing hypermedia-based cooperation environments. Our experiences with developing two cooperative systems, SEPIA [4, 17] and DOLPHIN [16], led us to work on a toolkit for the development of cooperative applications. This toolkit aims at supporting the object-oriented design and implementation of synchronous groupware.

In this paper, we introduce the COAST toolkit supporting the design of object-oriented cooperative applications. COAST (COoperative Application Systems Toolkit) is a toolkit that offers developers an architecture for cooperative applications and corresponding classes that can be used to implement such applications. The implementation of the toolkit does not require any specific platform (such as an operating-system, user interface management system, or communication channel). User requirements are reflected in the following features:

- a replication concept combined with a fully optimistic concurrency control which enables immediate processing of user actions independent of network latency,
- a synchronization mechanism for replicated objects ensuring fast propagation of other users' actions,
- a session-based concept supporting dynamic session behavior.

Development requirements are addressed by:

- replication management which enables the developer to transparently treat replicated objects as shared objects,
- transaction management controlling access to shared objects and ensuring integrity and consistency,
- session-controlled coupling which allows the application developer to control the coupling and decoupling of object aspects (parts of the state of an object, also known as *attributes* or *instance variables*) from a central point, leaving the rest of the application code untouched,
- display management taking the task of updating displays off the developer.

The paper is organized as follows. In the next section, we present an overview of related research. The following five sections introduce the COAST-Framework by introducing and discussing the above features. Some details of the implementation follow. The paper finishes with a comparison to related work, some conclusions and our plans for future work.

RELATED WORK

Support for the development of cooperative applications can be found in three areas: programming languages for distributed applications, shared window systems, and toolkits for cooperative application development.

In general, programming languages (or environments) for the development of distributed applications provide basic constructs for their design and implementation. Such constructs include, a means for distributing objects (e.g., name and location services), message passing, etc. A prominent example are the CORBA-based implementations of object brokers that allow the implementation of distributed object-oriented applications. Services range from object creation, registration, message passing and data marshalling to network time and object traders [21]. Since CORBA is not tailored for cooperative application development, it does not offer higher-level constructs for data sharing, communication of group awareness, etc. Other programming languages aiming to support cooperative applications, like COOC [18], provide a means for synchronous and asynchronous message passing and concurrent object execution, but also do not support higher-level constructs for data sharing, communication of group awareness, etc. It is clear that the price for the generality of these general purpose environments is less extensive support for dedicated application areas.

Another possibility for implementing cooperative applications is to use a shared window system like Shared-X [2]. Here, the application programmer does not deal with cooperation at all, since the shared window system allows several users to interact with a single-user application transparently. However, this inevitably leads to a tight coupling of the shared windows, and no functionality for partial decoupling or communication of detailed group awareness can be provided.

The most promising approaches can be found in the area of toolkits for cooperative application development. Systems in this category offer higher-level building blocks for the design and implementation of cooperative applications. The COLA system [19] provides a shared object service that is based on distributed objects (e.g., by using CORBA). By using adaptor objects developers can couple/share data objects for specific users. This enables the implementation of flexible cooperation functionality.

The DistView toolkit [12] supports the development of synchronous groupware applications. It is based on object replication and provides a default sharing model.

Rendezvous [10] provides an architecture for synchronous multi-user applications. Using a declarative graphic system and a constraint maintenance system, synchronous cooperative applications can be implemented. Rendezvous uses a centralized architecture based on X-Windows.

Groupkit [13, 3] is a toolkit for building real-time, distributed and fully replicated applications. It provides different policies for concurrency control and application-defined undo-redo capabilities [3]. The inter-process

communication facilities offered by Groupkit can be extended by the application developer.

Suite [1] is an environment that allows the declarative coupling of user-interface elements by an application developer. It is based on a centralized architecture and offers a pessimistic concurrency control schema (based on locking). It is very well suited for implementing fine-grained collaboration support. However, it is not easy in Suite to cater for the dynamically changing session behaviors (i.e., providing different predefined coupling behavior of arbitrary shared view objects) required for transitions between different phases of collaboration.

In summary, one can state that programming languages provide only basic support for the development of cooperative applications. Shared window systems offer good performance and easy development but allow only limited cooperation support and no flexible and variable session behavior. Not surprisingly, toolkits provide the most specific support for developing cooperative applications. However, systems such as COLA and Groupkit require extra programming effort to define the cooperative user interface and dynamic session behavior. On the other hand, systems like Rendezvous, DistView and Suite support the easy design of cooperative applications (including their user interface). But the definition of and switching between different session behaviors is not explicitly supported. With respect to the user requirements, one can observe that none of the toolkits provide higher level concepts supporting different and dynamic session behaviors.

In the next section, we will therefore introduce our approach to a toolkit that supports easier development of synchronous cooperative applications, allowing the definition of different session behaviors (collaborative modes) and supporting transitions between them.

THE COAST ARCHITECTURE

This section gives an overview of the COAST architecture, its components, and their purpose.

COAST uses a replicated architecture approach, i.e., a cooperative application consists of several application processes (i.e., instances of the application program) running on (usually) different sites. Each application operates exactly on one document that is shared between its application processes. The document data is fully replicated. An application process is designed according to the general architecture depicted in figure 1.

Each cooperative application manipulates a shared document. A document consists of *document parts* which may employ an arbitrary object structure. It is completely replicated for each application process.

Access from users to a shared document (or its parts) is coordinated via a *session* object. Session objects provide group awareness and specific coupling of shared document aspects between concurrent users. Thus, they can be used to implement specific cooperative modes [4].

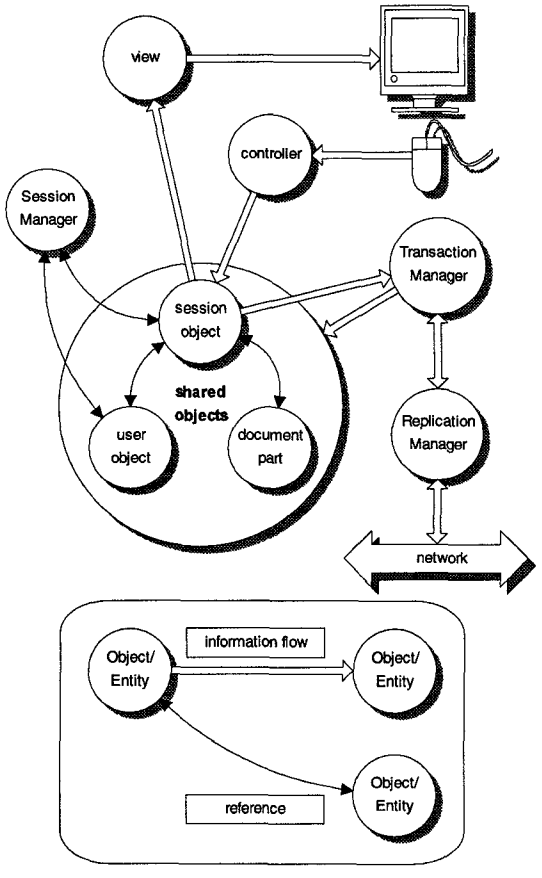

Figure 1: General architecture

Users interact with the shared document via *views* (visualizing the document in a window) and *controllers* (processing user input from the window). Views and controllers access the document by using the associated session object. This is an extension of the well known MVC-concept [9] to enable synchronous access to shared model objects (i.e., documents or their components) from several distributed view/controller pairs via replicated session objects. Thus, each application window (i.e., view/controller) displaying a part of the shared document is associated with a session coordinating access to its document part.

A *session manager* allows users to create, join, or leave sessions (thus, opening and closing of associated application windows).

User objects represent concurrent users of the document. They are used by session objects to maintain lists of their currently participating users.

All manipulations of the shared document, sessions, and user objects are encapsulated in transactions. A *transaction manager* ensures integrity of the shared objects

A *replication manager* is responsible for synchronizing the replicated objects (i.e., document, sessions, and user objects). By storing the replicated objects, documents can be made persistent.

In the sequel, we present the most important components provided by COAST: the session concept including user objects and session management, the view component, the transaction manager component, and the replication management.

THE COAST SESSION CONCEPT

The purpose of session objects is to model a group of users interacting with an application. Our understanding of 'session' does not cover the procedures of connecting to and disconnecting from the pool of shared objects. This functionality is provided by the replication manager.

In COAST,

- session objects maintain a list of user objects which represent users participating in the session.
- session objects refer to a part of the document on which the session currently focuses.
- session objects provide group awareness and define how participants of a session work together regarding the degree of coupling of the shared document's aspects. Therefore, session objects act as mediators between view-controller pairs and document parts.

User objects and session objects in COAST are persistent and are stored with the document.

Session Management

The administration of sessions (e.g. create, join, exit sessions) is handled by the COAST session management facilities. They automatically open application windows for each participant of a session on the specified part of the document.

When a user enters a session, the associated user object is added to the user list of the session. Leaving a session is done by removing the user-object from the session's user list. Opening and closing of the session's application window is done automatically.

A more sophisticated protocol for starting new sessions and joining running sessions is realized through coordination objects which are based on the simple mechanisms described above. For example, to invite a user U into a session S, the inviting user invokes some operation that puts the user object of U into the user list of an invitation object. This invitation object pops up a window on U's workstation and adds U's user object to the user list of S if U accepts the invitation.

Coordination objects can be used in various situations like voting or passing of notes between users.

Degrees of Coupling

The replication of objects results in an exact copy of all aspects of a document for each workstation. While this strict coupling meets the requirements of groups working on the same topic in a tightly-coupled manner, it tends to be hindering for many other cases. As a general cooperative application toolkit, COAST has to provide mechanisms for dynamically coupling and decoupling shared objects aspects'.

For example, in a cooperative text editor the position of the scrollbar needs to be coupled for all members of a tightly-coupled session [4], but it must not be coupled for other users working on that document. Similar problems arise for the document text itself. It is coupled between users in general, but private text elements such as annotations should be kept private for a specific user. During the analysis of meetings supported by DOLPHIN [16] we discovered situations – e.g. 'discussion' – in which the tightly-coupled mode for all group members best meets the needs of the group. In other situations, when meeting participants decide to work on their own or in subgroups, a loosely-coupled mode has to be established between the subgroups, while within each subgroup a higher level of coupling is needed.

We use the terms 'collaboration mode' and 'application context' to describe this kind of dynamic coupling. The *collaboration mode* defines the rules for coupling shared objects aspects' between users or groups of users. Examples of collaboration modes are 'tightly-coupled mode', 'loosely-coupled mode', 'individual mode', and 'subgroup mode'. A collaboration mode assigns each aspect to a level on which this aspect has to be coupled among users.

An *application context* summarizes properties of the actual situation in which users are working with an application. Properties of an application context c are:

- the group of users collaborating currently g
- the local user u

COAST uses deputy-object-modeling [11] to achieve a dynamic coupling behavior. Aspects of a coupled object O, which are not to be coupled within an application context c, are held in the deputy object O^c of O. O and O^c are instances of the same class and therefore have the same set of methods and possible aspects. In addition to O, O^c carries all the information needed to identify c and to distinguish between O^c and other deputy objects of O. Within the application context c, all requests for aspects of O are routed to the specific deputy object O^c. Requests for aspects that are coupled for c and thus not present in O^c are passed on to O. A deputy object itself can have a set of deputy objects. For example, deputy object $O^{g,u}$ holds specific aspects of object O for user u within the group of users g. Aspects that are common within the group of users g but that are not coupled with other users outside g are kept in O^g (c.f. figure 2).

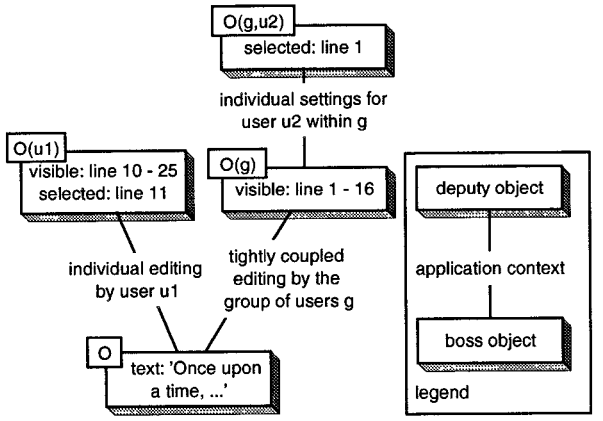

Figure 2: A tree of deputy objects for a cooperative editing situation. User u1 is editing in individual mode while user u2 together with other users is editing in tightly-coupled mode. Logically, all users see the same object O. Internally, user- and session-specific aspects are held in the deputy objects O(...). For example, the display of user u1 shows lines 10 to 25 and a selection of line 11. These aspects are private for each user and therefore stored in a deputy object O(u1) of boss object O. All requests for aspects of O made by user u1 are routed to O(u1). Requests which could not be handled by O(u1) are passed to O (e.g. for aspect 'text'). Members of g get the same value for the aspect 'text' user u1 gets, but they have scrolled to another position (lines 1 to 16, O(g)). The selection of text line 1 by u2 is kept private (using O(g,u)) even for other members of g.

Deputy objects are created on demand by the COAST system: An attempt to modify an aspect of object O which should be kept in O^c results in the creation of O^c, if O^c was not yet present.

Session Types

Deputy object modeling allows coupling behavior to be specified at an aspect-level granularity. For an application model with n different aspects and m possible levels of coupling, there are m^n possible collaboration modes of which only a handful are of practical use. For example, in the cooperative text editor it does not make sense to declare the text to be edited as private while setting the position of the scrollbars to be coupled.

For the end user who is usually unaware of the underlying data structure of an application, it is impossible to figure out this small set of useful configurations. As a solution, we modeled different collaboration modes as different types of sessions. While some generic default session types (such as loosely and tightly-coupled session) are already provided by COAST, application developers can extend this predefined set of session types by specifying the collaboration modes for new types. Session types can be generic (e.g. loosely-coupled) or tailored according to a concrete scenario in which a specific coupling behavior is needed (e.g. distributed code review).

More information about the COAST session concept can be found in [15].

THE COAST VIEW CONCEPT

COAST includes a powerful visualization framework which simplifies the implementation of an application's view objects.

Implementations of view objects in the standard MVC paradigm include two characteristic parts:

P1 A method or hierarchy of methods defining the visualization algorithm for a given model in a given graphical environment.

P2 A definition of update behavior, defining how the view reacts to its model's changes.

Given the characteristics of models in COAST, the P2 part becomes more complex than in a standard MVC approach, because changes of a model are not performed directly by the application developer's code. Changes are performed by the COAST transaction management facilities, which have the control over the point in time of the commitment of changes as well as the order in which they take place. In COAST, models can be any shared objects.

The goal of the COAST view updating concept is to ensure correct and efficient update behavior of views and to free the application developer of this responsibility, so that only the P1 part of view programming remains for the application developer.

To understand the updating problem, we have to look at the interface and conventions between views and models. In the standard MVC paradigm, we find four important conventions:

A1 Views read information from their model. For this purpose, models declare public messages to provide this information. Such messages can simply access instance variables or they can contain complex computations based on the model's instance variables and information coming from other objects.

A2 Application developer-defined methods of models that manipulate the state of the model contain notifications of change, which are propagated to dependents. Such notifications usually contain information about the aspect of the object, that has changed.

A3 Views have to register as dependents of their model. The application developer typically ensures this within the views' initialization methods.

A4 Views implement update methods which are invoked by notifications of change coming from their model. Such methods determine how a view reacts when a certain aspect changes in its model. The updating of views has two objectives: the propagation of damaged areas to the window system and the updating of instance variables of views.

Virtual slots as substitutes for instance variables of views

According to A4, the task of updating includes not only the invalidation of a view's graphical area on a window but also

the updating of instance variables. Although views are functionally dependent of their models, extensive use of instance variables is made for efficiency reasons. Instance variables of views, which are defined by application developers, carry information whose computation is rather expensive and which does not need to be recomputed for every update of a view. This is a kind of caching which reduces the cost of updating but normally increases the complexity of implementing updating behavior.

COAST substitutes such instance variables of views by so called *virtual slots*. Virtual slots are declared together with a method for the computation of their value. They are readable like normal instance variables but not writable. COAST is responsible for the evaluation of the computation methods of virtual slots and for caching their value. Like normal instance variables, virtual slots may contain references to objects of any type.

Dynamic dependency relation

In COAST, dependencies between views and models are dynamically detected at run time and it is not the application developer's task to care for maintaining them (unlike A3). During view computations, the system observes read operations (see A1) and dependency tuples are created. The dependency relation in COAST has a finer granularity than in the MVC paradigm: dependency tuples refer to instance variables of model objects and not to entire model objects. The relation can be seen as a multi level directed acyclic graph whose nodes are instance variables and virtual slots and whose edges are dependencies (see figure 3). Root nodes are instance variables of model objects; leaf nodes are from a special kind of virtual slot and represent graphical areas of the display.

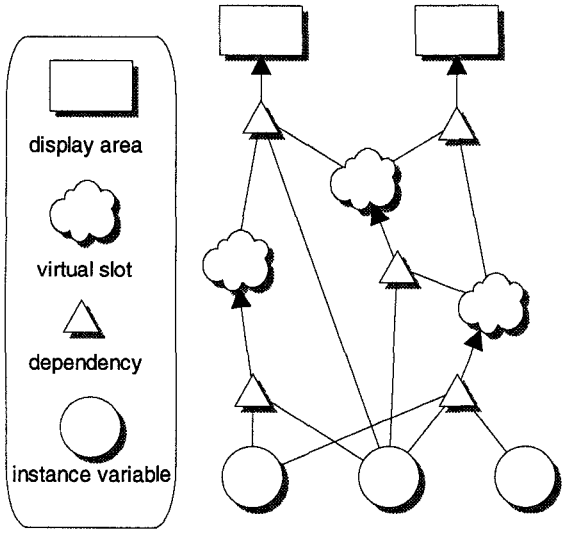

Figure 3: A sample dependency graph.

Generalization of model change

From the views' perspective, the transaction manager is seen as the source and the cause of change. Model change is generalized to consist of sequences of aspect manipulations and is regarded to be independent of the application method which originally initiated the change. In comparison, the standard MVC paradigm describes change by 'aspects' and attaches parameters to change notifications. Change notifications are embedded in the application methods that originally perform the change (see A2).

Propagation of change

In COAST, it is not the developers' task to identify change as such or to trigger any update. Instead, there is the convention between view, model and controller implementations, that every state of a model that can be produced after the completion of any manipulating transaction must be displayable. Change is automatically propagated along the dependency graph and the system ensures the triggering of updates.

Two phase protocol: damage collection and computation

In COAST, all computation of views is organized and triggered by the window management to overcome problems with temporarily inconsistent models and to avoid unnecessary computations. Creations and removals of views are also regarded as computations. The COAST window management mechanism triggers computation when either a new window is being opened or when damage to an existing window is being repaired. The concept of a distinction between damage collection and computation of display areas, which exists in many window systems, is extended to virtual slots. Not only display areas but also virtual slots can be marked as damaged in certain ways. During change propagation damage is only marked and no computation is triggered. During the repair phase of a window, computations are performed. The order of computations is partly pre-planned on the basis of the dependency graph and partly determined by the concept of lazy evaluation.

More information about the COAST view concept can be found in [7].

TRANSACTION MANAGEMENT IN COAST

A transaction manager keeps track of COAST transactions ensuring ACID properties [20]. COAST introduces three types of transactions:

- *User transactions* are used to encapsulate modifications of shared objects performed by controllers. This is the only difference to controllers in the original MVC concept. User transactions are performed immediately on the locally replicated objects (local commit). Afterwards they are broadcast by the replication manager to all other sites for a global commit or reject.
- *Display transactions* are used for encapsulating views' access to shared objects during view computations. Display transactions are automatically initiated by the system and do not modify shared objects.
- *Update transactions* are used by the replication manager to replay globally committed user transactions

that have been invoked on some other site on the affected replicated objects. Update transactions are also initiated by the system.

For user transactions durability is ensured after global commit which may be significantly later than local commit. The distributed concurrency control described in the next section globally commits transactions. After the local commit the user of a COAST application may continue invoking further user transactions even though this will increase the risk of a later reject.

REPLICATION MANAGEMENT IN COAST

The replication management of the COAST system provides an environment in which the application developer can handle application data (shared objects) in the same way as in conventional non-groupware applications. Technical details like replication, notification and concurrency control are hidden from the application developer.

Replication

To ensure optimal availability of the document and independency of application processes, documents in COAST are fully replicated for each site. Exactly one site of all sites holding a replicated document is responsible for storing this document. So, there is always a unique place for the persistent document.

Notification of Document Change

Modifications to the document can be made at any time by working on replicated objects using user transactions (see section about transaction management). To enable other sites to reflect these changes to a document, the changes have to be broadcast to all other sites. This process is called *notification* (c.f. figure 4).

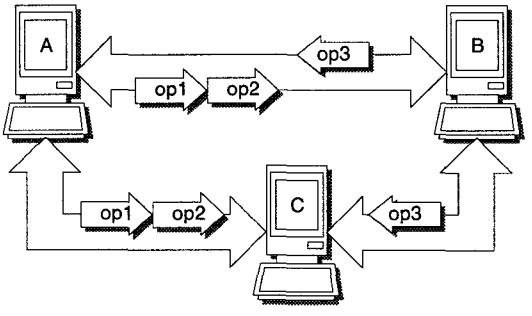

Figure 4: Three sites running a COAST application. At site A two operations have been invoked, at site B one operation. Operations are broadcast as notifications via network-communication links and are buffered until they can be processed at the receiver site.

Delivery of notifications may be delayed depending on network latency, but because of the fully optimistic concurrency control the COAST system does not have to wait for any results. This means that the user can continue to work even if there is a very unstable or slow network connection to the other sites.

Distributed Optimistic Concurrency Control

Replicates of a document at different sites have to be synchronized. This is done by notifications. Of course, there may be conflicting modifications made to the document that would result in inconsistent document states if each site simply executes the notifications it receives. Avoiding these inconsistencies is the task of the concurrency control.

One goal of the COAST design was to ensure non-delayed processing of user actions. Therefore the different application processes have to work independently from each other. A distributed optimistic concurrency control algorithm (DOCC) has been developed which fulfills this need. In the following paragraphs a short overview of the DOCC will be given (a detailed description can be found in [14]).

The DOCC belongs to the class of serialization algorithms that put all user transactions from all sites into one serial order. It is based on the Optimal Response Time (ORESTE) algorithm described in [8]. Similar to ORESTE, DOCC tries to minimize undo and redo of transactions when re-ordering transactions. However, unlike ORESTE it is possible that transactions may fail totally because of conflicting transactions that would destroy document consistency. A transaction will be globally committed when all sites have committed the transaction. This means that global commit is not granted explicitly like in conventional transaction systems. From the perspective of one site global commit can be detected implicitly as soon as the site has received at least one subsequent transaction from all other sites.

Transactions consist of basic operations that are defined by the transaction management. This has the advantage that undo and redo can be performed without involving the application developer and that commutativity of transactions can be detected by the replication manager.

IMPLEMENTATION

The COAST framework is implemented in VisualWorks Smalltalk from ParcPlace Systems. This provides a platform-independent development environment that even allows binary exchange of applications (Smalltalk images) between different platforms. For inter process communication the TCP/IP protocol is used which is also available on most platforms. The required network-throughput depends on the specific application data. For example, for a shared texteditor a telephone line would be sufficient.

USE OF THE COAST FRAMEWORK

For application developers, COAST provides both generic components (i.e., session manager, replication manager, transaction manager) as well as abstract classes that can be refined to implement specific applications. COAST provides abstract classes for views, controllers, replicated objects, and sessions.

The application developer has to follow rules in building his application:

1. The application's document model has to be designed as subclasses of the abstract COAST replicated object class.
2. The appropriate views have to be build as subclasses of the abstract COAST view classes, following the rules of COAST view development.
3. The corresponding controller classes have to be defined following the COAST controller rules. The most important rule is to use transactions for model manipulation. This rule requires the most effort, because standard Smalltalk controllers can not be used.
4. The application developer may use the generic sessions provided within COAST (tightly coupled, loosely coupled, private). He may also subclass these session classes to implement customized behavior.

DISCUSSION AND FUTURE WORK

The COAST toolkit introduced in this paper supports the development of synchronous, document-based groupware by providing a general architecture (for such applications) and corresponding classes. This enables the construction of cooperative applications that:

- ensure non-delayed processing of user actions for rapid feedback independent of the network connection (e.g. latency),
- ensure fast propagation of other users' operations to enable optimal group awareness under given network conditions,
- enable the coexistence of private and coupled information in a shared document. Furthermore, different degrees of coupling aspects of a shared document are supported,
- can be developed relatively easy by using or specializing existing classes, thus leading to less development effort, more rapid development, and reduced complexity of the resulting code,
- are not dependent on specific hardware/software/ network platforms or properties.

Comparison with Related Research

Comparing COAST to related work, one can state that, like any toolkit, COAST provides better support for constructing special applications (in this case: synchronous, document-based groupware) than any of the general programming languages for distributed environments.

Unlike shared window systems, COAST enables the implementation of different degrees of sharing and group awareness. However, additional development effort is necessary if none of the predefined session classes can be used.

Compared to other toolkits, COAST explicitly addresses the problems of object sharing, session management, and view updating. Other toolkits only support some of these by providing higher level concepts. For example, systems such as COLA and GroupKit require extra programming effort to define the cooperative user interface and specific coupling behavior. Systems like Rendezvous, DistView and Suite support the easy design of cooperative applications (including their user interface). But the definition of and switching between different degrees of coupling among arbitrary view objects and group awareness or cooperation modes is not supported.

However, it should be clearly noted that COAST also shows certain deficits. One of them is that COAST does not offer good support for asynchronous modes of sharing. The longer a user works on a shared document without a network connection, the higher the risk of later conflicts (and thus the possibility of loss of work) when the connection is re-established. Unlike planned extensions of GroupKit, COAST provides exactly one concurrency control algorithm. Another issue is related to our decision for using a full replication approach. While this offers benefits in terms of independence, it also means that larger documents require longer initial set up time (for loading the document) and that this approach limits the size of shareable documents.

Plans for the Future

There are a number of avenues to follow from this starting point. One of our projects is the development of a concurrency control handling partial replication for the model sharing component. Another important issue is the provision of better means for supporting asynchronous work. Here, we are currently integrating the VerSE flexible versioning environment developed at IPSI [5, 6]. This will enable us to provide specific session classes supporting asynchronous collaboration.

Until now, COAST has only been used to implement the latest version of the DOLPHIN cooperative hypermedia-based meeting support system. In the future, we will test the general applicability of our toolkit by using it for other application development projects.

From a more global perspective, we are in the process of conceptualizing the notion of "situation-aware" cooperative applications. Such applications will include components that may use knowledge about the current task, setting and infrastructure to (semi-) automatically provide adequate sessions supporting the collaboration between users. Later versions of COAST will provide generic support for this kind of situation-awareness.

ACKNOWLEDGEMENTS

We thank Boris Bokowski, Weigang Wang, Ajit Bapat, Jörg Geißler, David Hicks, Wolfgang Schuler, Daniel Tietze, and Thomas Knopik for valuable comments and discussions. Special thanks are also due Till Schümmer, Holger Kleinsorgen, Torsten Holmer, Roland Schleser, and Wolfram Wille who helped us to implement COAST.

REFERENCES

1 Dewan, P., Choudhary, R. Flexible user interface coupling in collaborative systems. *Proceedings of ACM CHI'91 conference, (New Orleans, Louisiana, April 27 - May 2, 1991)*, 41-49.

2 Garfinkel, D., Gust, P., Lemon, M., Lowder, S. The SharedX multi-user interface user's guide, version 2.0. *Hewlett-Packard Laboratories, Technical report STL-TM-89-07*, March 1989.

3 Greenberg, S., Marwood, D. Real time groupware as a distributed system: Concurrency control and its effect on the interface. *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW'94) (Chapel Hill, N.C., USA, October 22-26, 1994)*, 207-218.

4 Haake, J., Wilson, B. Supporting collaborative writing of hyperdocuments in SEPIA. Proceedings of the ACM Conference on Computer–Supported Cooperative Work (CSCW '92) Toronto, Canada (October 31 - November 4, 1992), 138 - 146.

5 Haake, A., Haake, J. Take CoVer: Exploiting versioning support in cooperative systems. *Proceedings of the InterCHI'93 Conference, Amsterdam (April 24-29, 1993)*, 406 - 413.

6 Haake, A., Hicks, D. VerSE: Towards hypertext versioning styles. Proceedings of the 7th ACM Conference on Hypertext (HT `96) (Washington, D.C., USA, March 16-20, 1996), in press.

7 Kirchner, L. Anzeigenaktualisierung in dem kooperativen Hypermediasystem DOLPHIN. Diploma thesis at Technische Hochschule Darmstadt, May 1995.

8 Karsenty, A., Beaudouin-Lafon, M. An algorithm for distributed groupware applications. Proceedings of the 13th International Conference on Distributed Computing Systems ICDCS'93, (Pittsburgh, May 25-28).

9 Krasner, G. E., Pope S. T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk. *Journal on Object Oriented Programming*, August/ September 1988.

10 Patterson, J. F., Hill, R. D., Rohall, S. L., Meeks, W. S. Rendezvous: An architecture for synchronous multi-user applications. *Proceedings of the Third Conference on Computer Supported Cooperative Work (Los Angeles, CA., USA, October 1990)*, 317-328.

11 Peng, Z., Kambayashi, Y. Deputy Mechanisms for Object Oriented Databases. *Proceedings of the IEEE 11th Conference on Data Engineering, (March 1995).*

12 Prakash, A., Shim, H. S. DistView: Support for building efficient collaborative applications using replicated objects. Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW'94) (Chapel Hill, N.C., USA, October 22-26, 1994), 153-164.

13 Roseman, M., Greenberg, S. GroupKit: A groupware toolkit for building real-time conferencing applications. Proceedings of the ACM 1992 Conference on Computer-Supported Cooperative Work (CSCW'92) (Toronto, Canada, October 31 - November 4, 1992), 43-50.

14 Schuckmann, C. Variable Synchronisations-mechanismen in dem kooperativen Hypemediasystem DOLPHIN. Diploma thesis at Technische Hochschule Darmstadt, May 1995.

15 Schümmer, J. Sessions zur Unterstützung von Gruppenaktivitat im kooperativen Hypermediasystem DOLPHIN. Diploma thesis at Technische Hochschule Darmstadt, May 1995.

16 Streitz, N., Geißler, J., Haake, J. M., Hol, J. DOLPHIN: Integrated meeting support across local and remote desktop environments and Liveboards. *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW'94) (Chapel Hill, N.C., USA, October 22-26, 1994)*, 345-358.

17 Streitz, N., Haake, J., Hannemann, J., Lemke, A., Schuler, W., Schütt, W., Thüring, M. SEPIA: A cooperative hypermedia authoring environment. *Proceedings of the 4th ACM European Conference on Hypertext (ECHT `92) (Milan, Italy, November 30 – December 4, 1992)*, 11-22

18 Trehan, R., Sawashima, N., Morishita, A., Tomoda, I., Inoue, A., Maeda, K. Concurrent Object Oriented C (COOC). *7th. Programming Language, Fundamentals and Practise Workshop, 1992.* Information Processing Society of Japan.

19 Trevor, J., Rodden, T., Mariani, J. The use of adapters to support cooperative sharing. Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW'94) (Chapel Hill, N.C., USA, October 22-26, 1994), 219-230.

20 Ullman, J. *Principles of Database Systems.* Computer Science Press, 1982.

21 Vinoski, S. Distributed object computing with CORBA. *C++ Report Magazine*, July/August, 1993

---

[i] Lutz Kirchner can now be contacted at:
TU-Dresden - Fakultät Informatik
Institut für Sofwaretechnik II
Lehrstuhl für Multimediatechnik
D-01062 Dresden, Dürerstraße 24
e-mail: Lutz_Kirchner@inf.tu-dresden.de
Tel.: (+49) 351 / 45 79 260 Fax: (+49) 351 / 45 93 056