# Architectural patterns for collaborative applications

Paris Avgeriou* and Peter Tandler

Fraunhofer Integrated Publication and
Information Systems Institute (IPSI),
Dolivostrasse 15, Darmstadt D-64293, Germany
Fax: +49-6151-869-963
E-mail: paris.avgeriou@ipsi.fraunhofer.de
E-mail: peter.tandler@ipsi.fraunhofer.de
*Corresponding author

**Abstract:** There is currently little reuse of either design or code in the development of collaborative applications. Though there are some application frameworks for this domain, they tend to be rather inflexible in the functionality they offer. This paper seeks to provide design reuse in the form of architectural patterns that focus on low-level horizontal issues: distribution, message exchange, functional decomposition, sharing data, concurrency and synchronisation. We base these patterns on a number of well-established patterns in the domain of distributed applications, concentrating on the specific issues that are encountered in the domain of collaborative applications. We also outline the relation between these low-level architectural patterns and the high-level functionality that collaborative applications offer. By codifying this knowledge and experience in the form of patterns, we hope for a wider support of low-level architectural design to the community of collaborative applications and thus a further advance of the field.

**Keywords:** pattern language; architectural patterns; software architecture; collaborative applications; CSCW; CSCL; distributed applications.

**Reference** to this paper should be made as follows: Avgeriou, P. and Tandler, P. (2006) 'Architectural patterns for collaborative applications', *Int. J. Computer Applications in Technology*, Vol. 25, Nos. 2/3, pp.86–101.

**Biographical notes:** Paris Avgeriou is a Senior Researcher at Fraunhofer IPSI under the Fellowship Programme of ERCIM. He received a Diploma (MSc) in Electrical and Computer Engineering (1999), as well as a PhD in Software Engineering (2003) from the National Technical University of Athens (NTUA), Greece. He has worked as a Senior Researcher at the Software Engineering Competence Center, University of Luxembourg (2004), as a Visiting Lecturer at the Department of Computer Science, University of Cyprus (2003) and as a research and teaching assistant at NTUA (1999–2002). He has participated in a number of EU research projects, has an extensive publication list and co-organises international workshops. He is a member of ERCIM, Hillside Europe and a founding member of the World-Wide Institute of Software Architects. His research interests concern the area of software engineering and particularly software architecture, patterns, evolution, business modelling and web engineering. His research is mainly applied in the domains of e-learning and CSCW.

Peter Tandler is a Senior Researcher at Fraunhofer IPSI, where he directs the Computer-Supported Cooperative Work (CSCW) research group of IPSI's Cooperative Environments and E-Learning research division. His research interests include CSCW, integration of virtual and physical environments, new forms of human–computer and team–computer interaction for roomware components (i.e. interactive chairs, tables and walls with integrated information technology), but also software architectures and frameworks, object and component technology and design of programming languages. He serves as programme committee member and reviewer for international conferences, workshops and journals in the areas of software architectures for ubiquitous computing environments, collaboration in ubiquitous computing environments and human–computer interaction. He received a Diploma (MSc) in Computer Science with Education and Psychology as additional subjects, and a PhD from the Technische Universität Darmstadt, Germany for his work on application models and software infrastructures for roomware environments.

## 1 Introduction

The field of collaborative applications is gaining prominence, as virtual organisations and virtual groups are growing and becoming the status quo of contemporary working and learning. Shared applications and desktops, synchronous and asynchronous text and multimedia communication, multi-user networked games, collaborative authoring and development environments are some of the potential future best-sellers in the domain.

Despite the increasing demand for collaborative applications and their already widespread use, development teams still do not receive much assistance in building applications in this domain (Schuckmann et al., 1999). In mature domains, such assistance is provided in the form of frameworks, patterns, reference architectures, modelling languages, reference models, CASE tools, etc. Of all these development aids, the only one that has advanced remarkably in the domain of collaborative applications is application frameworks. There are several frameworks in the market that offer reuse of design and code, for example Agilo (Guicking, 2005), Clock (Graham et al., 1996; Urnes and Graham, 1999), COAST (Schuckmann et al., 1996), Dream-Objects (Lukosch, 2003), DreamTeam (Roth, 2000a), GroupKit (Roseman and Greenberg, 1996), Rendezvous (Hill et al., 1994), Suite (Dewan and Choudhary, 1995) and TCD (Anderson et al., 2000). But the shortcoming of these frameworks is that they are characterised by limited scalability, extensibility and customisability.

There have also been some attempts to define patterns in the domain (Lukosch and Schümmer, 2004; Schümmer, 2003). This paper contributes to the effort of codifying the design experience of collaborative applications in the form of patterns. The level of abstraction that we are focusing on is low-level architectural patterns that address horizontal concerns in collaborative applications and satisfy basic functional and quality requirements. Specifically, these architectural patterns deal with six fundamental aspects of distributed applications: distribution, message exchange, functional decomposition, sharing data, concurrency and synchronisation.

It is emphasised that we regard collaborative applications as a special category of *distributed* applications that facilitate the collaboration between groups of people who share a common task. Consequently, we consider only *distributed* collaborative applications, and not *centralised* collaborative applications (e.g. Single Display Groupware, Stewart et al., 1999). Therefore, we reuse a number of the existing architectural patterns for distributed applications from the current literature (such as in Buschmann et al., 1996; Gamma et al., 1995; Schmidt et al., 2000), and especially build on a pattern language for distributed computing (Buschmann and Henney, 2002) that attempted to unify several of the former. We select and choose these patterns on the basis of the idiosyncrasy of collaborative applications and the domain-specific horizontal requirements.

We also note that, in the software architecture discipline, the architecture of a system is comprised of components, connectors, their configurations and constraints on all of them (Bass et al., 1999; Clements et al., 2002; Shaw and Garlan, 1996). Therefore, the architectural patterns that are described in this paper deal with these concepts in order to provide problem–solution pairs in a given context. Furthermore, the notion of component is perceived in the broad sense of the term, as a unit of run-time interaction (Shaw and Garlan, 1996), and not in the narrow sense of industrial standards, for example, Java Beans (Sun Microsystems, Inc., 2005).

The contribution of this paper is twofold. Firstly, we outline the 'big picture' of architectural design in collaborative applications in the form of a set of patterns. These are both high-level domain-specific patterns and low-level application-general patterns. Secondly, we specify six of the low-level patterns that tackle coarse-grained problems in the realm of application distribution. Development teams can navigate through the pattern language and use these patterns to design the software architecture (Avgeriou et al., 2004; Buschmann et al., 1996; Shaw and Garlan, 1996) of collaborative applications, in the sense that patterns generate architectures (Beck and Johnson, 1994). We provide experience and knowledge in this field, in digestible and inter-related chunks, and therefore help software architects, especially those who are inexperienced.

The structure of the rest of this paper is as follows: Section 2 presents the pattern language; Section 3 elaborates on six architectural patterns and Section 4 wraps up with conclusions derived from this work.

## 2 The pattern language

Figure 1 depicts a map of the pattern language as well as the main relationships between the proposed patterns. The arrows of the relationships denote the direction of the relationships, according to their labels. This pattern language attempts to tackle the complex problem of designing the software architecture of collaborative applications, focusing on the lower application-general layers. The *intended audience* for the language is software architects who are designing collaborative applications. This pattern language does not imply a waterfall-like up-front architecture design approach, but rather supports iterative and incremental approaches, where the patterns are applied iteratively in cycles, each time refining and fine-tuning.

There are two groups of patterns in this pattern language. The first group, which is comprised of the patterns at the top of Figure 1 with the striped background, are high-level architectural patterns that focus on the domain-specific functionality of collaborative applications. The remainder constitute the second group of patterns, which are low-level architectural patterns that deal with the distributed nature of collaborative applications.

The patterns of the first group are outside the scope of this paper and will be sketched only briefly, as follows:

- The COLLABORATIVE PROCESS (also known as SESSION MANAGEMENT or WORKFLOW MANAGEMENT) is the implementation of the

domain-specific functionality of a collaborative application. It represents the model of a collaborative process, such as instant messaging or application sharing, as well as the execution of this process during collaborative scenarios. It organises the structure of the collaborative groups, in terms of *actors* who play specific *roles*, in order to carry out specific collaboration scenarios and achieve particular goals. It provides USER AND ACTIVITY AWARENESS to the users according to the collaboration scenarios at hand. It uses COMMUNICATION AND COLLABORATION TOOLS to facilitate the synergy between the users. It appropriately manages the FLOOR CONTROL, to coordinate the interaction of users, as well as the COUPLING CONTROL, to control the degree of interaction data-sharing. A suitable DISTRIBUTION MODEL$_{3.1}$ can ensure that the low-level details of distribution and communication are separate from the actual FUNCTIONAL DECOMPOSITION MODEL$_{3.3}$ of the functionality specified in the COLLABORATIVE PROCESS.

- USER AND ACTIVITY AWARENESS helps users to coordinate their work by informing them about the activities of other users (Gutwin et al., 1996). This awareness can be performed either synchronously or asynchronously, depending on whether activities of other users take place the same time or have happened in the recent HISTORY. An appropriate CONCURRENCY MODEL$_{3.5}$ assists in coordinating concurrent threads or processes that are controlled by the different users.

- The HISTORY, also known as ELEPHANT'S BRAIN (Schümmer, 2003), of the users' activities is thoroughly maintained, and is comprised of all the relationships between the users and the tools or artefacts that they are currently using. It is used for synchronising the activities of the users that take place either at the same time or at different times. Such HISTORY information is stored using the SHARING MODEL$_{3.4}$ and accessed through a SYNCHRONISATION MODEL$_{3.6}$.

- COMMUNICATION AND COLLABORATION TOOLS are well-established instruments in virtual organisations, for example, e-mail, chat, instant messaging, annotations, audio–video conferencing, discussion forums, application sharing, document sharing, etc. It is noted that these tools should be able to support synchronous or/and asynchronous collaboration (Edwards and Mynatt, 1997), depending on the application requirements. These tools rely on the low-level MESSAGE EXCHANGE MODEL$_{3.2}$ and SHARING MODEL$_{3.4}$, while their interactivity and layering is tackled by the FUNCTIONAL DECOMPOSITION MODEL$_{3.3}$.

- FLOOR CONTROL deals with which user is allowed or should do something at what times, according to the COLLABORATIVE PROCESS. A very restricted form of floor control is *turn-taking*, when only one user at a time can, for example, modify data, while others have to wait. The other extreme is to allow *full control* for each user, that is, all users can simultaneously modify the shared data (see SHARING MODEL$_{3.4}$). Depending on the application, different floor control strategies are appropriate. While *turn-taking* has been found in many cases to be too restrictive and to reduce efficiency, *full control* is more challenging with respect to the SYNCHRONISATION MODEL$_{3.6}$, as conflicts have to be avoided (if possible) or detected and resolved. Additionally, *full control* requires more USER AND ACTIVITY AWARENESS features in the user interface to aid coordination (Gutwin et al., 1996).

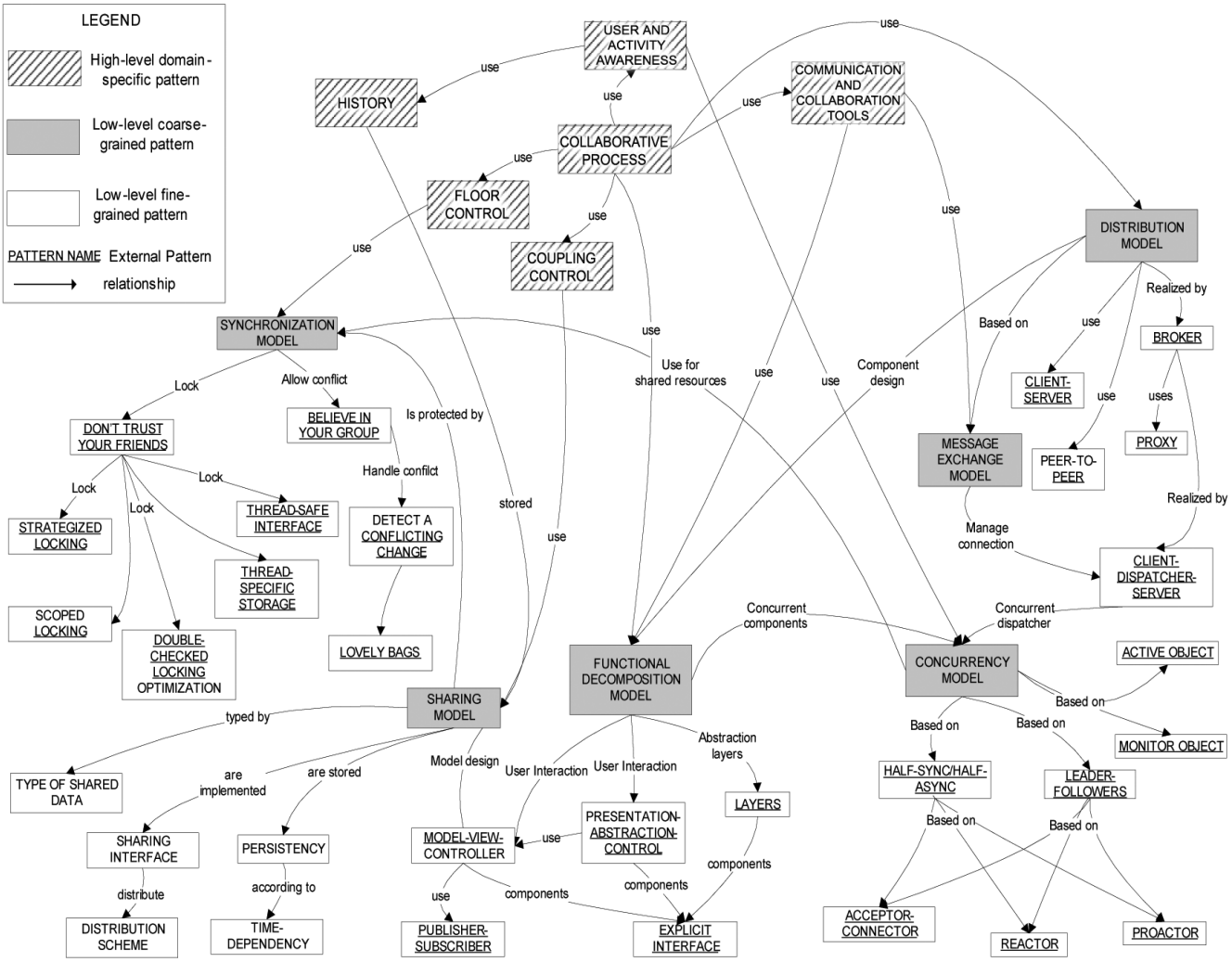
- COUPLING CONTROL regulates the *coupling mode*, that is, how tightly and to what extent users collaborate (Dewan and Choudhary, 1995). Users can thus collaborate in different modes, ranging from very loose coupling (e.g. single-user applications) to extremely tight coupling (Berlage and Genau, 1993; Haake and Wilson, 1992). The degree of coupling may be customisable according to the COLLABORATIVE PROCESS that is being enforced during a user's session. Technically, COUPLING CONTROL defines how tightly collaborative tools couple the application functionality among users and which part of the editing state needs to be designed as the shared *activity model* (see SHARING MODEL$_{3.5}$ and Schuckmann et al., 1999). For example, when working loosely coupled, navigation within a document is independent for different users, and no USER AND ACTIVITY AWARENESS needs to be provided. When working tightly coupled, users could, for example, couple navigation and require awareness of other users' activities that need to be coordinated. Whereas COUPLING CONTROL and FLOOR CONTROL are in general orthogonal to each other, tight collaboration can often be more effectively supported when FLOOR CONTROL is not very restricting.

The patterns in the second group constitute the focus of this paper: low-level architectural issues. These patterns can be further decomposed into two subcategories: those in white background in Figure 1 are fine-grained patterns that solve specialised architectural design problems; the ones in grey background are coarse-grained patterns that use the former in order to solve more general problems.

The coarse-grained patterns comprise the contribution of this paper as they tackle six major architectural design problems in the field of collaborative applications, helping groupware architects in designing the software architecture. These patterns will be elaborated in the remainder of this section, while their thumbnails follow:

- The DISTRIBUTION MODEL$_{3.1}$ takes care of the basic form of distribution in the collaborative application, in order to decouple distribution issues from the application functionality.

**Figure 1** Map of the pattern language and main relationships between the patterns




- The MESSAGE EXCHANGE MODEL$_{3.2}$ (also known as DATA TRANSFER) specifies the low-level communication mechanism between components in the context of a DISTRIBUTION MODEL$_{3.1}$, especially offering location transparency.
- The FUNCTIONAL DECOMPOSITION MODEL$_{3.3}$ mandates the basic architectural design of the application functionality itself, emphasising a layered structure and the high degree of interactivity that characterises collaborative applications.
- The SHARING MODEL$_{3.4}$ specifies the data that is common to the various distributed components of a collaborative application and needs to be communicated to them.
- The CONCURRENCY MODEL$_{3.5}$ deals with the design of multiple processes and threads that execute concurrently in a collaborative application.
- The SYNCHRONISATION MODEL$_{3.6}$, stems from the need to coordinate the concurrent access of multiple components to the shared data, as described by the CONCURRENCY MODEL$_{3.5}$.

The formation of these architectural patterns has been based on the categorisation in (Buschmann and Henney, 2002), where distributed applications are supported by patterns in five discrete areas: distribution infrastructure, application infrastructure, synchronisation, event handling and concurrency. We have considered that event-handling patterns should not receive particular focus in the domain of collaborative applications, but can be used within the scope of the CONCURRENCY MODEL$_{3.5}$. We have also added one more category of patterns that focus on the low-level architecturally significant issue of the SHARING MODEL$_{3.4}$. Naturally, other patterns and pattern languages can also be employed when the requirements of the collaborative applications demand specific solutions to corresponding problems.

As aforementioned, the coarse-grained patterns use the fine-grained patterns in the sense that the former solve coarse-grained problems by decomposing the solution into fine-grained solutions, resolved by the latter. Most of the fine-grained patterns have been documented elsewhere as object-oriented design patterns or architectural patterns (Buschmann et al., 1996; Gamma et al., 1995; Schmidt et al., 2000). Five of them constitute future work and are described by the following thumbnails:

- TYPE OF SHARED DATA$_{3.4}$ defines the characteristics of the distributed components' data (e.g. documents, user models, user activities, etc.) with respect to sharing.
- The SHARING INTERFACE$_{3.4}$ deals with the interface and the abstractions that developers use for the actual implementation of the shared data. It separates the definition of the data from the techniques used to implement sharing.
- DISTRIBUTION SCHEME$_{3.4}$ denotes the way that data are scattered among the distributed components.
- PERSISTENCY$_{3.4}$ deals with whether and how the shared data needs to be stored for future retrieval.
- TIME-DEPENDENCY$_{3.4}$ handles the duration of PERSISTENCY$_{3.4}$ according to the role that shared data plays in the collaborative application.

The next section presents the six coarse-grained architectural patterns in an analytical pattern format.

## 3 The patterns

Throughout the description of the patterns, in the example section, we demonstrate small parts of a case study that concerns an invented collaborative text editor named CoText, which can be used by multiple distributed[1] users to edit a document simultaneously. Few architectural designs of this case study will be given in the UML 2.0 notation, accompanied by explanatory text. We assume a few basic requirements in order to keep the example simple and comprehensible and, at the same time, provide insightful demonstrations of the various architectural patterns. We consider that the text editor handles simple ASCII text and the users can:

- create and delete documents
- create and delete, join and leave collaborative editing sessions
- insert, delete and format characters simultaneously
- change the position of their personal cursor and change their personal selection
- scroll their text view in order to make different parts of the document visible, independently of other users and
- get awareness of other users' actions (a remote view of cursor positions, selections and scroll positions is assumed here for simplicity).

### *3.1 Distribution model*

**Context:** You just took up the task of designing the architecture of a collaborative application. There are many facets in this task and you are trying to decide where to start.

**Problem: Distribution and application functionality are two inter-dependent facets of collaborative applications that need to be addressed by architects.** These two facets depend on each other and need to be efficiently integrated, in order to achieve a good trade-off of qualities for both facets. However, trying to tackle them simultaneously is overly intricate and problematic.

**Forces:**

- Collaborative applications are distributed software systems in their core, so that they facilitate multiple users to collaborate synchronously or asynchronously.
- The design of distributed applications is a tedious task because it requires the resolution of many issues that do not exist in centralised systems, such as communication problems (e.g. latency, failure, etc.), scalability and reliability.
- Designing the application functionality of a collaborative application is complex enough by itself, striving to satisfy the requirements of supporting groups of collaborating people, as well as other quality requirements.
- Designing the application functionality separately from distribution issues is problematic because they depend closely on each other.
- Designing the application functionality together with distribution issues can become a daunting task, because there are too many issues of different nature that need to be taken under account altogether.

**Solution: Decouple any distribution issues from the application functionality.** The first task in designing the architecture of the collaborative application should be to design the DISTRIBUTION MODEL$_{3.1}$, which mandates how components are distributed over a network, independently of their application functionality. This model addresses the quality requirements of the application at the distribution level, such as scalability and robustness.

At first, the only concern that should be addressed is the way the components of the application will be distributed. Domain-specific functionality of collaborative applications should not be discussed yet; instead, this functionality should be orthogonal to the DISTRIBUTION MODEL$_{3.1}$, and designed afterwards in the form of a FUNCTIONAL DECOMPOSITION MODEL$_{3.3}$ that can rely on the former.

The fundamental architectural patterns for application distribution are the well-established CLIENT-SERVER (Shaw and Clements, 1997) and PEER-TO-PEER (Clements et al., 2002) patterns. The former denotes an asymmetrical style of communication, while the latter considers a symmetrical relationship between components of equal functionality. These patterns are not always discrete, for example, a peer component can play the role of both a client and a server, according to the specific functionality it entails at a given moment.

Furthermore, in order to decouple distribution issues from domain-specific functionality of collaborative applications, a BROKER (Buschmann et al., 1996) can be used. It manages the communication of components in a

distributed application through a federation of BROKERS that forward requests and transmit back the results transparently, thus hiding the distribution. The BROKER maintains a Server Registry, so that components can *dynamically* register their services and clients can transparently look them up and subsequently invoke them. Such services in a collaborative application can range from generic distribution functionality to domain-specific groupware functionality, such as chatting, application sharing, etc.

In order to decouple the clients and the servers from the BROKER components in both the client and the server side, a PROXY (Gamma et al., 1995) can be used to mediate between the two parties. A PROXY provides location-independent access to services provided by other components through a transparent access mechanism. In this case, a PROXY on the client side can offer the interface of the server component, so that a client component communicates with a remote server component as if it was local, and moreover is not concerned with communicating with the client-side BROKER. Similarly, a PROXY on the server side will receive a request and call the appropriate service on behalf of the client. In both cases, PROXIES use their own mechanisms for communicating with the BROKERS, hiding them from the clients and server components. Furthermore, PROXIES can provide independency from the implementation language of the servers, in order to also include legacy code and ensure the heterogeneity of the distributed application. Finally, the PROXIES are an ideal place to marshal and un-marshal the client requests and the server responses. The process of *marshalling* takes care of transforming a message into machine-independent format, suitable for transmission through a network.

The BROKER needs to deal with low-level communication issues, such as the kind of Inter-Process Communication (IPC) or Remote Procedure Call (RPC) used or the deployment configuration of the components in the network. This can be further encapsulated into the MESSAGE EXCHANGE MODEL$_{3.2}$ that provides such services to application components. Therefore, the MESSAGE EXCHANGE MODEL$_{3.2}$ encapsulates the part of the BROKER functionality that tackles connection management and location transparency of the distributed components.

The DISTRIBUTION MODEL$_{3.1}$ addresses the quality requirements at the distribution level. At a later stage of the architectural design, further quality requirements at the domain-specific application functionality level will need to be addressed that contradict the former. A QUALITY TRADEOFF ANALYSIS (Avgeriou et al., 2004) must be performed in order to compromise the qualities according to the STAKEHOLDERS AND THEIR CONCERNS (Avgeriou et al., 2004).

**Example:** In our sample collaborative text editor, the CLIENT-SERVER architectural pattern was deemed more appropriate, where the Server persistently stores the shared documents and mediates communication among the Clients for concurrent editing. A PEER-TO-PEER scheme could also be possible but it would inflict far more complex synchronisation issues. In both client and server sides, except for the CoText Client Component and CoText Server Component that perform the domain-specific functionality of the collaborative editor, there are BROKERS to handle the communication, and PROXIES to provide location transparency to clients.

The servers register themselves to the server-side BROKERS, while the clients look up the servers through the client-side BROKERS, and as a result they fetch the corresponding client-side PROXIES. The clients can proceed and make requests through the client-side PROXY, by using the interface of the corresponding server component, as if the servers were local. The client-side PROXIES forward such request to the client BROKERS, which then undertake all the details of the MESSAGE EXCHANGE MODEL$_{3.2}$, in order to get the request to the BROKER on the server side. The latter similarly forwards the requests to the actual server components through the mediation of the server-side PROXIES. Though the BROKER pattern can potentially let clients and servers communicate directly after the link is established, in the CoText application, this link is always indirect through the BROKERS, in order to take advantage of the MESSAGE EXCHANGE MODEL$_{3.2}$ facilities of low-level communication details and handling of network problems.

It is obvious that the domain-specific functionality in terms of the client requests and the server responses are not tackled here, but will be discussed in the FUNCTIONAL DECOMPOSITION MODEL$_{3.3}$.

**Benefits:** The application of the pattern entails the following positive and negative consequences:

- The architectural design of the collaborative application commences from the right point: their distributed nature.
- All issues of distribution are decoupled from the application functionality.
- All issues of communication are based on the MESSAGE EXCHANGE MODEL$_{3.2}$.
- The collaborative application is scalable and offers location transparency of components.

**Liabilities:**

- There are certain issues of the distribution infrastructure that are not dealt with by merely enforcing the BROKER pattern. For example, the reliability and fault tolerance of the network are completely ignored at this point; this has to be handled by the MESSAGE EXCHANGE MODEL$_{3.2}$.
- The DISTRIBUTION MODEL$_{3.1}$ enforces a communication overhead to the collaborative application.

**Known Uses:**

**COAST** (Schuckmann et al., 1996) is a framework for synchronous groupware. It uses the CLIENT-SERVER pattern: all application-specific functionality is located in the clients, while the server (called ‘mediator’ in COAST) takes care of coordinating replication and synchronisation. It uses the BROKER pattern to encapsulate communication and PROXIES to access server functionality.

**Rendezvous** (Hill et al., 1994) uses the CLIENT-SERVER pattern as well. All model objects (called 'abstractions') are located on the central server and accessed via 'links' by 'views' local to each client.

**DreamTeam** (Roth, 2000ab) is built as a PEER-TO-PEER framework in order to improve communication efficiency and avoid the server as bottleneck. A 'connection manager' represents a BROKER responsible for communication among collaborating peers.

**Suite** (Dewan and Choudhary, 1995) is another example of a groupware framework that supports a PEER-TO-PEER distribution model.

**Wikis** and **BSCW** are both based on the CLIENT-SERVER pattern, while being web applications they use the standard internet infrastructure as networks of BROKERS and PROXIES.

Microsoft **Netmeeting** combines a PEER-TO-PEER structure for communication between nodes on intranets or the internet, as well as a CLIENT-SERVER mode that mainly implements a naming service in order for peers to locate each other.

### *3.2 Message exchange model*

**a.k.a.** DATA TRANSFER, COMMUNICATION INFRASTRUCTURE

**Context:** You have designed the DISTRIBUTION MODEL$_{3.1}$ of the collaborative application, which necessitates communication between distributed components.

**Problem: The implementation of communication increases the complexity of distributed components if the low-level details of data exchange have to be considered by all parts of the application that deal with remote components. Also, enclosing the communication details in the distributed components makes them inflexible to future evolution as it violates the 'separation of concerns' principle**.

**Forces:**

- In a collaborative application, the actual communication and collaboration is realised through data being transmitted between the various distributed components through a network.
- Distributed components can send and receive data by implementing IPC mechanisms, or even the more structured RPC mechanisms. Unfortunately, this substantially increases their complexity. To make matters worse, the components depend highly on the details of the specific IPC or RPC mechanism used and thus need to change when these mechanisms evolve over time.
- The topology of distributed components that follows the DISTRIBUTION MODEL$_{3.1}$ is hard-coded in the implementation of the communication mechanism. In other words, components depend on location-specific details of other components and need to update whenever the distributed topology changes.

**Solution: Encapsulate all the details of communication into a separate MESSAGE EXCHANGE MODEL$_{3.2}$ that operates at the lowest level of abstraction**. This model should take care of sending and receiving data in the form of messages between clients or servers or between peer components according to the DISTRIBUTION MODEL$_{3.1}$. The distributed components use this MESSAGE EXCHANGE MODEL$_{3.2}$ seamlessly and transparently without being aware of its internal details.

In order to design the MESSAGE EXCHANGE MODEL$_{3.2}$ the CLIENT-DISPATCHER-SERVER (Buschmann et al., 1996) pattern can be used. The Dispatcher undertakes the exchange of messages between clients and servers, or between peer components that act as both clients and servers. The Dispatcher provides location transparency and conceals the communication connections between the distributed components. All low-level communication details, and especially IPC mechanisms (e.g. shared memory, pipes and sockets) or RPC are hidden to the services that use this MESSAGE EXCHANGE MODEL$_{3.2}$. Furthermore, network problems, such as latency and failure of network nodes, are also taken care of, by this mechanism transparently. Finally, the naming service provided by the Dispatcher allows server components to register themselves and client components to look up the former, thus offering location transparency. Consequently, the Dispatcher receives a request from a client, finds the appropriate server component and establishes a communication link between them for transparent exchange of messages. In other words the Dispatcher implements the basic functionality of the broker, as prescribed in the DISTRIBUTION MODEL$_{3.1}$, because it implements the naming service, manages communication details and offers IPC.

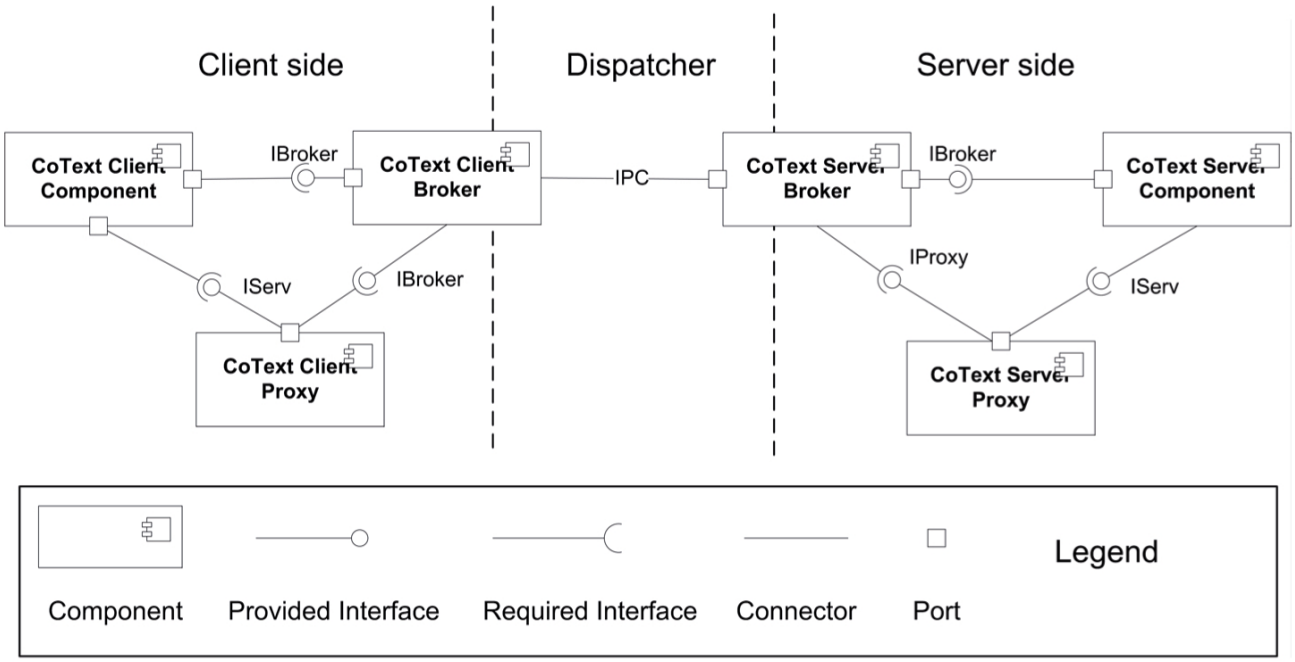
It is noted that the MESSAGE EXCHANGE MODEL$_{3.2}$ as a coarse-grained pattern is in fact a variation of the fine-grained pattern, CLIENT-DISPATCHER-SERVER, for collaborative applications. The added value of this variant is its description in the context of collaborative applications, as well as its relationships with the other patterns in this pattern language.

This pattern can be implemented by appropriate middleware technologies that are based on international standards, such as CORBA (Object Management Group, 2004), J2EE (Sun Microsystems, Inc., 2005) and .NET (Microsoft Corporation, 2005). These technologies offer location-independence of components, flexible component deployment, integration of heterogeneous components and especially legacy code.

Because the Dispatcher is a single point of connection management between multiple components it needs to be able to concurrently serve more than one component. It therefore needs to be designed according to an appropriate CONCURRENCY MODEL$_{3.5}$.

**Example:** In our example text editor, the MESSAGE EXCHANGE MODEL$_{3.2}$ is implemented through a Dispatcher. As already illustrated in Figure 2, the BROKER is implemented as a Dispatcher that takes care of registering server components, providing a naming service for them to be located by clients and establishing

**Figure 2** Application of the DISTRIBUTION MODEL and the MESSAGE EXCHANGE MODEL patterns for the collaborative text editor




communication between clients and servers through sockets. Because the application is simple and lightweight, we preferred not to use a heavy implementation of a BROKER, such as CORBA or J2EE, but to implement it from scratch. However, if features such as scalability or reliability are an issue, using a messaging framework would be a better choice.

**Benefits:** The application of the pattern entails the following positive and negative consequences:

- Low-level communication details of the MESSAGE EXCHANGE MODEL$_{3.2}$ are encapsulated in a separate layer. Any changes to the communication mechanisms do not affect the application components.
- Location transparency between the application components is achieved. The network topology can change without affecting the application functionality.
- Potential problems in the network can be independently dealt with, thus ameliorating fault tolerance.

**Liabilities:**

- An overhead is introduced due to the indirection layer of the dispatchers.
- The Dispatcher's quality of service might degrade if it attempts to sequentially serve multiple components. It needs to be designed according to a CONCURRENCY MODEL$_{3.5}$.

**Known Uses:** Most groupware systems and frameworks have a component responsible for message exchange.

**COAST** (Schuckmann et al., 1996) has a connection and communication layer that is responsible for sending and receiving messages between client and server.

**DreamTeam** (Roth, 2000a) has a message-based communication kernel and connection manager that handles event distribution.

**GroupKit** (Roseman and Greenberg, 1996) offers support for transparent RPC and shared events.

**Agilo** (Guicking, 2005) implements the CLIENT-DISPATCHER-SERVER pattern, providing location-transparent message exchange.

### *3.3 Functional decomposition model*

**a.k.a.** DOMAIN-SPECIFIC MODEL

**Context:** You have designed the DISTRIBUTION MODEL$_{3.1}$, which took care of the distribution issues, for example, decomposing the application to clients and servers or peers and PROXIES. You can continue to design the domain-specific issues of each of the distributed parts, for example, the client or the peer, focusing on their application functionality.

In software architecture terms, the domain-specific application functionality is designed as clearly separated components, which are the run-time units of computation or storage, and connectors, which are the units of interaction.

**Problem: In every application domain there are certain fundamental features that shape the domain-specific model of applications. The identification of these crucial features in collaborative applications, as in any immature application domain, is an error-prone task, especially for inexperienced architects.**

**Forces:**

- The application functionality of a software system is based on a few domain-specific features, which play a pivotal role. The selection of these features depends on the requirements of the particular application.

- The features that shape the architecture of an application concern mainly the following: how flow of data and control runs through the application; how processing of data takes place; how the components cooperate with each other; how the application interacts with the users or other systems and how the system evolves in the context of changing environment and requirements.
- Selecting the appropriate features in architectural design entails a vast impact on how the system will satisfy its functional and quality requirements.

**Solution: Design the FUNCTIONAL DECOMPOSITION MODEL$_{3.3}$ of the application, which organises its structure into clearly separated components and connectors, with respect to the two crucial features that characterise the domain of collaborative systems from an architectural viewpoint:** Firstly, the functionality of collaborative applications is highly structured, therefore the components can be grouped and use the services of other components that belong to other groups; secondly, collaborative applications are highly interactive systems that offer an extensive user interface that manipulates the shared data.

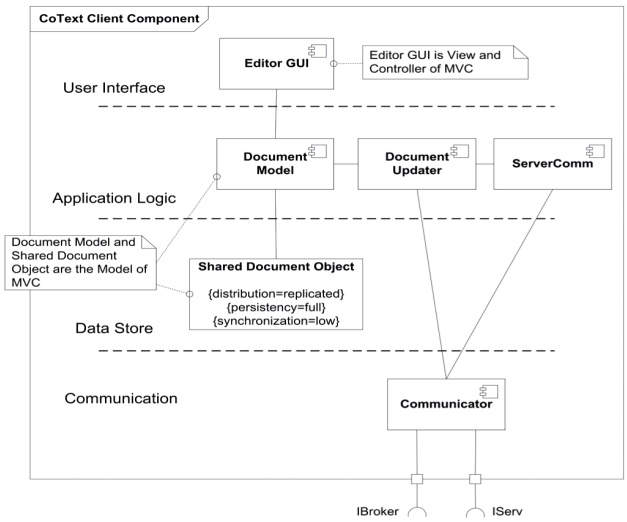
The first feature is addressed by the LAYERS pattern (Buschmann et al., 1996), which enforces an organisation of the system components into different levels of abstractions so that components in higher LAYERS reuse the services of lower LAYER components. The components in each LAYER are thus decoupled from the rest because a LAYER communicates with the lowest LAYERS through the EXPLICIT INTERFACE (Buschmann and Henney, 2003) of the latter. The EXPLICIT INTERFACE ensures that each LAYER has a discrete, unchanged interface, while the implementation of the functionality in each LAYER is hidden and may potentially change without consequences. The top LAYER should implement the user interface of the collaborative applications, while the bottom LAYER should be concerned with the MESSAGE EXCHANGE MODEL$_{3.2}$ (Dewan, 1999; Patterson, 1995).

The second feature is tackled by the MODEL-VIEW-CONTROLLER (MVC) pattern (Buschmann et al., 1996; Krasner and Pope, 1988), which decouples the user interface (both views and controllers of the views) of collaborative applications from their application logic and the stored data. Therefore, any modifications to the user interface will not affect the application logic or the data. On the other hand, changes to the data can be propagated to the user interfaces of the distributed applications. A propagation mechanism, such as PUBLISHER–SUBSCRIBER (Buschmann et al., 1996), is required, where a Publisher keeps a registry of Subscribers and notifies them when it changes state, so that they can retrieve the updated state. Furthermore, some of the data that belong to the Model (in terms of MVC) need to be shared by the distributed components. These data are therefore a subset of the Model's data and cover different aspects of collaborative applications, as explained in the SHARING MODEL$_{3.4}$: *documents* that users work with, *user models* and *activity models*. The Model can be designed according to the EXPLICIT INTERFACE pattern, to achieve its decoupling from views and controllers. As in collaborative applications client components implement the Views and Controllers that concurrently access a shared Model, an appropriate SYNCHRONISATION MODEL$_{3.6}$ needs to be established.

Furthermore, if a collaborative application is separated into several functional units, each having its own User Interface (e.g. a collaborative software development platform), the PRESENTATION-ABSTRACTION-CONTROL (PAC) pattern (Buschmann and Henney, 2002; Calvary et al., 1997) will be the best choice. Because the PAC pattern embodies MVC in every agent, the principles of applying MVC hold for PAC as well.

**Example:** In our running example of a collaborative text editor, we need to design the FUNCTIONAL DECOMPOSITION MODEL$_{3.3}$ of all the components that originate from the DISTRIBUTION MODEL$_{3.1}$. Owing to space limitations we will demonstrate here only the FUNCTIONAL DECOMPOSITION MODEL$_{3.3}$ of the CoText Client Component and the CoText Server Component. The CoText Client Component, as illustrated in Figure 3, implements both the MVC and the LAYERS patterns. The MVC pattern deals with the shared document and manages to decouple the User Interface from the data and the application logic that modifies it. The Editor GUI Component encapsulates both the View and the Controller because for a text editor it makes sense to keep them connected rather than discrete, as View and Controller require lots of information exchange. The Shared Document Object, on the other hand, contains the data of the Model, while the Document Model implements the application logic that modifies the data. The entire CoText Client Component is organised in four different LAYERS: the User Interface, which apparently contains the Editor GUI; the Application Logic, which contains the aforementioned Document Model, the Document Updater, which manages the synchronisation of the document modifications with the other users and the ServerComm, which looks up the server and retrieves its interface; the Data Store, which contains the shared document itself; and the Communicator, which handles the interaction with both the Client BROKER and the Client PROXY.

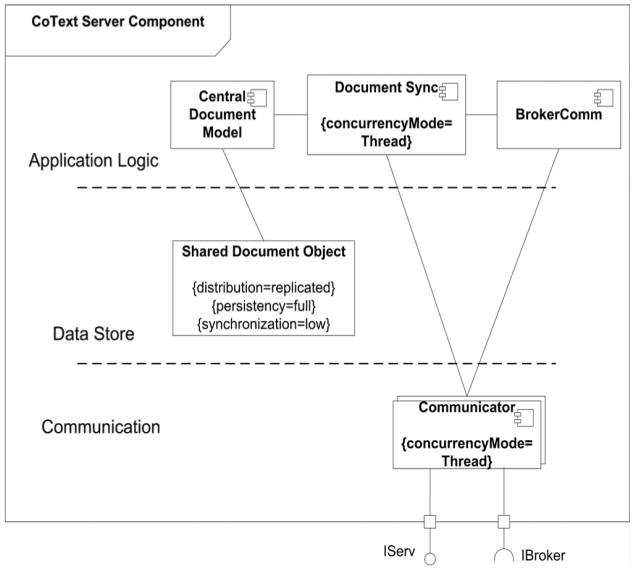
**Figure 3** Application of the FUNCTIONAL DECOMPOSITION MODEL pattern for the CoText Client Component



The CoText Server Component, as illustrated in Figure 4, implements the LAYERS pattern and not the MVC because the server does not include a user interface. In particular, the three different LAYERS of the server component are: the Application Logic, which contains the Central Document Model shared by all the clients, the Document Sync, which synchronises the clients and the BrokerComm, which registers the server with the BROKER; the Data Store, which contains the shared document itself and the Communicator, which handles the interaction with both the Server BROKER and the Server PROXY.

As illustrated in both the CoText Client and Server Components, as well as the PROXIES and the BROKERS (see Figure 2), each and every one of these components communicates through EXPLICIT INTERFACES. In particular, the Server Component offers the IServ interface, which is also provided to the Client component by the client-side PROXY. The server-side PROXY offers the IProxy interface to the server broker, while the BROKERS on both server and client side offer the IBroker interface.

**Figure 4** Application of the FUNCTIONAL DECOMPOSITION MODEL pattern for the CoText Server Component




**Benefits:** The application of the pattern entails the following positive and negative consequences:

- The collaborative application is organised into layers of abstraction that communicate through EXPLICIT INTERFACES.
- The SHARING MODEL$_{3.4}$ and the application logic that modifies it are decoupled from the User Interface.
- If the user interface is bound to the SHARING MODEL$_{3.4}$ via PUBLISHER–SUBSCRIBER, the user interface remains independent of the location and identity of the components that modify the shared data.
- Separation of concerns is achieved among several domain-specific semantic concepts, if the PAC pattern is applied.

**Liabilities:**

- The application of the three patterns, LAYERS, MVC and PAC introduce *indirections* that may potentially jeopardise the performance of collaborative applications.
- LAYERS, MVC and PAC satisfy only some of the requirements of collaborative applications. The development team needs to find more architectural patterns in the pattern catalogues in order to cover all the requirements and design the full software architecture.

**Known Uses:**

**DreamObjects** (Lukosch, 2003) separates the application, service, data and communication LAYERS.

**Beach** (Tandler, 2004) also defines four LAYERS, namely the task, generic, model and core LAYERS. In addition, it separates data, application logic, environment, user interface and interaction issues in different components. MVC is used to structure the interaction components.

**COAST** (Schuckmann et al., 1996, 1999) separates views, application model and domain model, extending MVC. The framework defines LAYERS for transaction handling, shared data and connection and communication.

The **Clock** (Graham et al., 1996; Urnes and Graham, 1999) architecture separates the application functionality from the DISTRIBUTION MODEL$_{3.1}$. It uses a layered MVC as architectural style.

**BSCW** follows the MVC pattern to decouple the application functionality (called kernel) and the user interfaces, also providing an interface to create or modify user interfaces.

Additionally, there are some general domain-specific decomposition models for groupware applications. **Dewan's generic architecture** (Dewan, 1999) for groupware and **Patterson groupware taxonomy** (Patterson, 1995) and the **C2 architecture** (Taylor et al., 1996) highlight that groupware applications should be structured in LAYERS.

## *3.4 Sharing model*

**a.k.a.** SHARED DATA

**Context:** You have designed the FUNCTIONAL DECOMPOSITION MODEL$_{3.3}$, which includes components that conform to the MVC (Buschmann et al., 1996; Krasner and Pope, 1988). You carry on refining the Model of those components that are shared between the users.

**Problem: Managing the data that is shared between the distributed components is a complex task.** What exactly qualifies as shared data and how it should be managed, needs to be precisely specified. Unnecessary sharing of too much information may also decrease performance.

**Forces:**

- A collaborative application bases its functionality on data that is shared among the users of the application.

- It is not always clear what qualifies as shared data and what should better be kept within individual components.
- Choosing the right resources for sharing is essential for designing collaborative applications, as it may burden performance. Especially, responsiveness of the user interface needs to be ensured for interactive applications.
- The type of shared data and how exactly the data should be managed by the application is another key design issue.
- Some shared data need to persist over time, while others have a limited lifetime.

**Solution: Design the SHARING MODEL$_{3.4}$ by identifying all data that is common to the distributed components of the collaborative application and analysing the TYPE OF SHARED DATA$_{3.4}$.** Depending on the application requirements choose an appropriate SHARING INTERFACE$_{3.4}$ that allows decoupling the SHARING MODEL$_{3.4}$ from the DISTRIBUTION SCHEME$_{3.4}$ as well as mechanisms for PERSISTENCY$_{3.4}$ and TIME-DEPENDENCY$_{3.4}$.

When specifying the SHARING MODEL$_{3.4}$, start with the TYPE OF SHARED DATA$_{3.4}$ that determines '*what*' needs to be shared and may vary according to the application requirements (Rubart and Dawabi, 2004). As a good starting point one can look at the components identified in the FUNCTIONAL DECOMPOSITION MODEL$_{3.3}$. Nearly all collaborative applications rely on shared *documents* that users work with, for instance the drawings on a shared whiteboard, the messages exchanged in a chat application or program source code that is collaboratively edited by two developers. Furthermore, *user models* (e.g. during a session, the COLLABORATIVE PROCESS$_2$ defines which users work together in which manner) as well as *activity models* (all the actions that users perform), can be shared when USER AND ACTIVITY AWARENESS$_2$ must be provided. Normally, also the HISTORY$_2$ is shared in collaborative applications in order to provide common context information to all cooperating users.

The SHARING INTERFACE$_{3.4}$ defines the mechanism and abstractions to be used to share data. Depending on the sharing technology, this interface can vary: Some groupware frameworks require shared data to inherit from some Shared Object base class; other systems define a protocol and interface that shared data need to conform to, in order to get shared, while still others merely define a component responsible for transmitting information between nodes that all shared data have to use. So, in addition to the '*what*' specified by the TYPE OF SHARED DATA$_{3.4}$, the SHARING INTERFACE defines '*how*' the data are shared from the developers' point of view. This is influenced by the choice of the appropriate DISTRIBUTION SCHEME$_{3.4}$ (or DISTRIBUTION ARCHITECTURE, Phillips, 1999): shared data can be distributed in one of the following schemes: *central, asymmetric, semi-replicated and replicated* (Lukosch, 2002; Phillips, 1999). By introducing a SHARING INTERFACE$_{3.4}$ in between the SHARING MODEL$_{3.4}$ and DISTRIBUTION SCHEME$_{3.4}$, the distribution scheme is decoupled from the data itself, allowing a flexible adaptation of distribution schemes (Graham et al., 1996). Compared with the DISTRIBUTION MODEL$_{3.1}$, which describes how the *components* are distributed, the DISTRIBUTION SCHEME$_{3.4}$ is concerned about the distribution of the *shared data*.

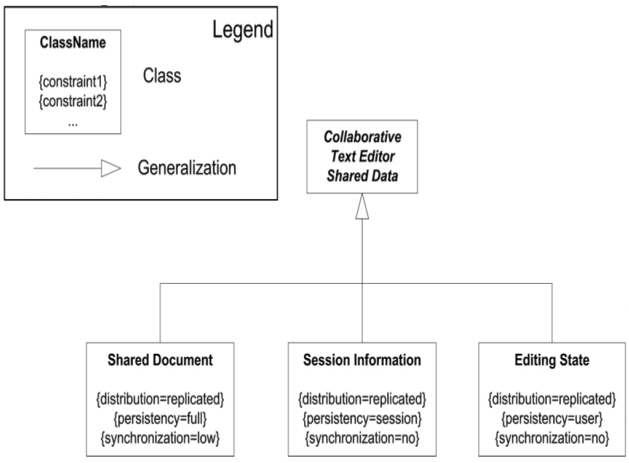
The PERSISTENCY$_{3.4}$ of shared data are also a key issue: some data need to be stored and retrieved later, whereas others may be discarded. Furthermore, data can be characterised by TIME-DEPENDENCY$_{3.4}$, if they persist for only a limited period of time, before they become obsolete and are no longer stored.

Shared data need to be protected from concurrent access, according to their individual properties. Some data will be modified simultaneously with a high (or low) chance of possible conflicts. Other data are always modified by a single user having write access, whereas others get notified of changes without being able to change the data themselves. For some data it is crucial that all collaborating users get a synchronised representation as quickly as possible; for other data it might be acceptable that the representation can temporarily differ for users, as long as consistency is ensured at some point in time. These characteristics strongly influence which CONCURRENCY MODEL$_{3.5}$ and which SYNCHRONISATION MODEL$_{3.6}$ are appropriate for which data, which SHARING INTERFACE$_{3.4}$ should be chosen and which data needs to be kept persistent for how long.

Using the MVC pattern (Buschmann et al., 1996; Krasner and Pope, 1988) for DISTRIBUTION MODEL$_{3.1}$ components, as a rule of thumb the Model part of MVC triplets needs often to be shared data (Graham et al., 1996; Hill et al., 1994; Schuckmann et al., 1999). Changes to the Model are propagated to the Views and the Controllers through notification mechanisms, such as a PUBLISHER–SUBSCRIBER (Buschmann et al., 1996).

**Example:** In the collaborative text editor case study, there are three TYPES OF SHARED DATA$_{3.4}$ (Figure 5): the shared document, the session information (i.e. the *user model*, which represents which users have joined the session and their location) and the editing state (i.e. the *activity model*, which represents each user's cursor position, selection and scrollbar position in this example).

**Figure 5** Application of the SHARING MODEL pattern for the collaborative text editor



As far as the SHARING INTERFACE$_{3.4}$, all shared objects inherit from an abstract class that declares shared status. Furthermore, the DISTRIBUTION SCHEME$_{3.4}$ of the editor is fully replicated: the document is small enough that full replication is not too costly, and replication improves the responsiveness of the user interface, which is crucial for interactive applications.

The PERSISTENCY$_{3.4}$ of the editor application is as follows: the document will remain persistent until explicitly deleted, the session information is persistent only during that session and the editing state remains persistent until the corresponding user leaves the session.

On the issue of synchronisation, the document may have simultaneous modifications with low chance of conflicts, and timely synchronisation is desirable to support tight collaboration. On the other hand, session and awareness information is always changed by a single user only, so no conflicts are possible there.

**Benefits:** The application of the pattern entails the following positive and negative consequences:

- The understanding of the nature of shared data allows defining the requirements for sharing technology, synchronisation, distribution, persistency and time-dependency.
- The SHARING INTERFACE$_{3.4}$ decouples the shared data from the sharing technology and the DISTRIBUTION SCHEME$_{3.4}$.
- The application components that are designed according to the MVC architectural pattern have a clearly defined Model, which contains documents, user models and activities.

**Liabilities:**

- There is always a possibility that some important data have not been included in our SHARING MODEL$_{3.4}$.
- The problem of change propagation from the model to the Views and Controllers needs to be tackled.

**Known Uses:** All groupware systems are fundamentally dependent on one form or another of shared data.

**COAST** (Schuckmann et al., 1996), **Rendezvous** (Hill et al., 1994), **Clock** (Graham et al., 1996; Urnes and Graham, 1999), **PAC*** (Calvary et al., 1997), **Suite** (Dewan and Choudhary, 1995) and later versions of **GroupKit** (Roseman and Greenberg, 1996) provide support for sharing the Model (in terms of MVC).

More recent systems, such as **DreamObjects** (Lukosch, 2003) or **Dragonfly** (Anderson et al., 2000) define a SHARING INTERFACE in order to provide flexibility in adapting the DISTRIBUTION SCHEME. **Wikis** do not prescribe a specific SHARING INTERFACE but they almost always have a centralised DISTRIBUTION SCHEME. Furthermore, all the shared data of **Wikis** are persistent and do not depend on time. **CVS** provides a semi-replicated DISTRIBUTION SCHEME and, of course, unlimited persistence of files and metadata. **BSCW** offers a SHARING INTERFACE that allows for the persistent storage of new objects without modifying the existing storage mechanisms.

To realise PERSISTENCY of SHARED DATA, **DreamTeam** (Roth, 2000a) has a 'Persistence Kernel' and 'Archive Manager'. Using the **COAST** framework (Schuckmann et al., 1996), all shared objects are persistent. The TIME-DEPENDENCY of shared objects can be controlled by the application by 'naming' objects: while shared objects are garbage collected when no longer referenced, named objects remain persistent as long as they have a name assigned.

### *3.5 Concurrency model*

**Context:** You have designed the FUNCTIONAL DECOMPOSITION MODEL$_{3.3}$ of the collaborative application and you have identified several distributed components that run concurrently. Also, the Dispatcher of the CLIENT-DISPATCHER-SERVER (Buschmann et al., 1996) setting must be concurrent to facilitate the communication between multiple clients and servers.

**Problem: The complexity of collaborative applications makes it hard to deal with concurrency issues:** the satisfaction of non-functional (quality) requirements and especially performance, the combination of synchronous and asynchronous components, heterogeneous networking, etc.

**Forces:**

- In a collaborative application several client components may concurrently require access to the same service. Furthermore, multiple threads on the server side may be concurrently serving the client components.
- The non-functional requirements (also known as quality requirements), especially performance, robustness and scalability, need to be maintained in conditions of concurrency.
- A collaborative application usually combines synchronous and asynchronous components that potentially depend on each other.
- The messages that are exchanged through distributed components usually represent events, in essence, even if the MESSAGE EXCHANGE MODEL$_{3.2}$ hides the details of event handling from developers of application level components. Handling of events in a concurrent environment poses another layer, which needs to be tackled in the design of collaborative applications.
- The components that need to be executed concurrently may run on heterogeneous hardware nodes or be implemented with diverse software technologies.
- Concurrent access to components or shared data may cause conflicts and inconsistencies, thus raising issues of synchronisation.

**Solution: Design the CONCURRENCY MODEL$_{3.5}$ of the collaborative application in order to ensure that the quality requirements of the collaborative application are satisfied, focusing especially on its performance.**

This model should specify how the application components may run as concurrent processes and/or threads through the appropriate process and thread management mechanisms.

In collaborative applications, there are usually both synchronous and asynchronous components in the FUNCTIONAL DECOMPOSITION $MODEL_{3.3}$. For example, in an instant messaging application, file transfer is performed synchronously whereas messages are asynchronously sent and received between users. The interweaving and cooperation between synchronous and asynchronous components can be successfully achieved through the HALF-SYNC/HALF-ASYNC pattern (Schmidt et al., 2000), which decouples the two different kinds of component. Thus, asynchronous services are not slowed down by synchronous ones, and the latter do not deal with the complexity of the former.

In the CLIENT-DISPATCHER-SERVER setting, multiple threads are used in the Dispatcher to process the incoming events. These threads can be organised according to the LEADER/FOLLOWERS (Schmidt et al., 2000) pattern, in order to ensure the efficiency of the application. It shifts the paradigm from one thread per event, to multiple threads that handle a shared set of source events. As far as the event handling per se, it can be managed according to the REACTOR and PROACTOR patterns (Schmidt et al., 2000), in both HALF-SYNC/HALF-ASYNC and LEADER/FOLLOWERS. These patterns hide the complexity of event demultiplexing and dispatching to clients, while especially taking care of handling multiple simultaneous or asynchronous events. Depending on whether we need simplicity of implementing the event-driven mechanism or high performance, we should choose the REACTOR or the PROACTOR, respectively. Finally, for peer-to-peer systems, the ACCEPTOR-CONNECTOR (Schmidt et al., 2000) can help to establish connections between peers independently of their domain-specific functionality.

When there is concurrent access to individual *shared components* by multiple clients, these components can be implemented as ACTIVE OBJECTS or MONITOR OBJECTS (Schmidt et al., 2000), either running within their own threads of control or being shared between multiple client threads, respectively. Client components are not able to tell the difference between shared components. In the case of a MONITOR OBJECT, the different instances of a shared component need to synchronise themselves with the help of a SYNCHRONISATION $MODEL_{3.6}$, to ensure consistency when accessed concurrently.

**Example:** In our running example of a collaborative text editor, an important issue of concurrency arises when the server needs to respond to multiple client requests in order to modify the shared document. We use two patterns in the design of the CoText Server Component to tackle this issue: firstly, in order not to block the clients waiting for a request to modify the shared document, we apply the ACTIVE OBJECT pattern; secondly, in order to improve the efficiency of the server we apply the LEADER/FOLLOWERS pattern.

Both of the applied patterns are indicated in Figure 4, which illustrates the FUNCTIONAL DECOMPOSITION $MODEL_{3.3}$ of the CoText server component. As the ACTIVE OBJECT pattern mandates, the Communicator component, which accepts the client requests, is separate from the Document Sync component, which processes the requests. Also, according to the LEADER/FOLLOWERS pattern, the Communicator component is comprised of multiple threads that take turns in order to accept the client requests and dispatch them to the Document Sync component.

**Benefits:** The application of the pattern entails the following positive and negative consequences:

- Concurrency issues of the collaborative application are resolved.
- Asynchronous and synchronous components cooperate with each other, without the former degrading their performance, or the latter becoming more complex.
- Performance and robustness of multi-thread applications are ensured, even in cases of a heterogeneous network.
- Event handling in concurrency settings is inherently designed.

**Liabilities:**

- The access to shared objects and resources needs to be synchronised through a specialised SYNCHRONISATION $MODEL_{3.6}$.
- The combination of the various patterns for concurrency and event-handling entail an inherent complexity in the implementation of the collaborative application.

**Known Uses:**

**DreamTeam** (Roth, 2000a) uses the HALF-SYNC/HALF-ASYNC pattern by providing a ‘connection manager’ responsible for synchronous communication and a ‘transfer manager’ that takes care of large asynchronous transfer of data.

**COAST**’s server (Schuckmann et al., 1996) uses an ACTIVE OBJECT to process messages received from clients.

### *3.6 Synchronisation model*

**a.k.a.** Concurrency Control
**Context:** You have designed the CONCURRENCY $MODEL_{3.5}$ of the collaborative application, which has entailed concurrent access to the shared data.

**Problem: The unrestricted concurrent access to the shared data may lead to corruption or inconsistent representation of the data.** The protection of this data through a conflict handling or prevention mechanism is imperative but not easy, especially as it should not introduce unsolicited overhead.

**Forces:**

- A collaborative application bases its functionality on a set of persistent data that is shared among all the users of the application.
- When a CONCURRENCY MODEL$_{3.5}$ is enforced, there may be cases where more than one user concurrently attempts to modify shared data. In such cases, the shared data must be protected to avoid corruption. Conflicts must be detected and resolved to ensure consistency and integrity of shared data.
- There are low-level locking mechanisms that can be applied, but more sophisticated strategies must be devised.
- Locking mechanisms may cause performance overhead.

**Solution: Design the SYNCHRONISATION MODEL$_{3.6}$ of the collaborative application in order to regulate concurrent access to shared data and safeguard this data from corruption.** Introduce the appropriate locking and conflict resolution mechanisms to ensure consistency of the data but also make sure that the robustness and performance of the application are not compromised.

There are several ways to ensure the consistency of data. A common approach is to *lock* data that will be changed by an atomic operation (e.g. DON'T TRUST YOUR FRIENDS (Lukosch and Schümmer, 2004)). Another possibility is to allow conflicts to occur, but *detect and handle* them afterwards (BELIEVE IN YOUR GROUP and DETECT A CONFLICTING CHANGE (Lukosch and Schümmer, 2004)). While locking can slow down applications if the same data is modified frequently by different clients, conflict handling is possible only if conflicts can be gracefully resolved or operations can be rolled back. Orthogonally to the synchronisation mechanism, *transactions* can be used to implement atomic operations.

Common locking mechanisms are mutexes and semaphores. The SCOPED LOCKING (Schmidt et al., 2000) provides a simple but effective way to automatically acquire and release locks within a scoped area of code, independently of the return path. The efficiency and consistency of the locking mechanisms can be implemented through DOUBLE-CHECKED LOCKING OPTIMISATION (Schmidt et al., 2000), which prevents protected areas of code from being unnecessarily accessed and avoids race conditions. Furthermore, THREAD-SAFE INTERFACE (Schmidt et al., 2000) minimises the overhead caused by locking and prevents the application from self-deadlocks. Finally, THREAD-SPECIFIC STORAGE (Schmidt et al., 2000) suggests an alternative locking mechanism, by providing access to shared data that is local to a thread, through a unique access point. If the optimal locking mechanism can change, STRATEGISED LOCKING (Schmidt et al., 2000) can dynamically bind any synchronisation mechanism (e.g. mutexes and semaphores) to a component, thus providing flexibility, customisability and reusability.

It is a trade-off for architects to decide whether the overall cost for pessimistic execution of all operations or the cost to handle conflicts of optimistic operations is higher. Optimistic transactions can ensure consistency while increasing the responsiveness of the user interface (Schuckmann et al., 1996). This is critical for interactive applications and can be used when conflicts are unlikely or harmless. Sometimes it is also possible to adjust shared data structures to use data types that minimise or avoid conflicts, such as LOVELY BAGS (Lukosch and Schümmer, 2004). If, for instance, a chat application uses a shared array to store all messages, it has a high chance of conflicts; if modelled as a set and a timestamp is assigned to each message, inserting messages can be done without risking conflicts, while still being able to compute an unambiguous order of messages.

**Example:** In our sample collaborative text editor, the issue of synchronisation arises in the case of the shared document, where there is chance of conflicts when users edit the same part of the document. For some operations, conflicts can be easily handled: if a character is inserted after another character that has concurrently been deleted, the operation can be adjusted to insert the character after the predecessor of the deleted character. For other operations no conflict resolution is possible, but the effect of cancelling the operation is harmless: if a character is formatted that has been deleted concurrently, the operation can simply be ignored, as it has become obsolete. Therefore, we can use an optimistic synchronisation mechanism for the shared document. To implement optimistic operations, the client's Document Updater components have to remember all locally executed operations until they receive the acknowledge from the server's Document Sync component (see Figures 3 and 4).

For the user and activity models, synchronisation is quite straightforward: these models have discrete parts that are modified by only a single user at any given time, thus conflicts cannot occur. It must only be ensured that all clients are informed about updates to the shared data (UPDATE YOUR FRIENDS and MEDIATED UPDATES (Lukosch and Schümmer, 2004)).

**Benefits:** The application of the pattern entails the following positive and negative consequences:

- Advanced locking mechanisms are designed that ensure shared data are always consistent when accessed concurrently.
- The performance of the system is not compromised by the locking mechanisms, and phenomena such as starvation, deadlocks, race conditions, etc. are avoided to the best possible extent.

**Liabilities:**

- When using locks as synchronisation mechanisms there is the danger of causing deadlocks.
- Potential misuse of synchronisation strategies can lead to excessive overhead.

**Known Uses:** While some groupware frameworks provide support for synchronisation, others leave it up to application developers to ensure consistency.

**COAST** (Schuckmann et al., 1996) supports both optimistic and pessimistic transactions. When optimistic transactions are used, the server takes care of detecting and resolving conflicts. For pessimistic transactions, different locking strategies can be used.

**Clock** (Urnes and Graham, 1999) provides two predefined concurrency control schemes, pessimistic locking and optimistic conflict detection with transparent rollback.

In **Dragonfly/TCD** (Anderson et al., 2000) each component has a facet called 'sequencer' and 'concurrency controller' that handles synchronisation of the component's view and model.

**Wikis** usually allow conflicts to happen and of course detect conflicting changes, but they also cater for locking individual web pages.

**CVS** is a typical example of pessimistic transactions, which enforces various locking mechanisms but also offers conflict prevention with unreserved checkout. Similarly, **BSCW** allows version management through locking but may also allow for conflicts to take place.

## 4 Conclusions

The domain of collaborative application is beginning to mature and, naturally, attempts to reuse design and code are emerging. This paper presents such an approach for reusing design in the form of architectural patterns, focused on low-level issues of application distribution. These patterns are intended for software architects who design collaborative applications, and heavily rely on related patterns from the distributed computing domain. The architects of a collaborative application are meant to use some of the patterns presented in this paper by customising them to their specific requirements and discard the rest. The benefits of applying the patterns can be summarised as follows:

- Patterns are indeed quite flexible and can be customised and applied in the vast majority of collaborative applications. This is due to the nature of patterns: they are generic enough, 'timeless' in nature, so that they provide wide coverage in architectural design issues, but on the other hand, they are flexible enough to be parameterised and solve individual problems.
- The whole pattern language serves as a roadmap that architects can follow in order to solve a number of architectural design problems. Each problem can be resolved through the pattern, while its interdependencies with other problems and their solutions are explicitly portrayed.
- The low-level fine-grained patterns serve as the documentation of the major architectural design decisions that the development team took. These decisions play a pivotal role in communicating the architecture during the application development and, even more so, during evolution.

In the future, we intend to explore the space of high-level architectural patterns that concern domain-specific functionality as identified in Figure 1.

## References

Anderson, G.E., Graham, T.N. and Wright, T.N. (2000) 'Dragonfly: linking conceptual and implementation architectures of multiuser interactive systems', *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, New York, NY: ACM Press, pp.252–261.

Avgeriou, P., Guelfi, N. and Razavi, R. (2004) 'Patterns for documenting software architectures', *proceedings of the Ninth European Pattern Languages of Programming Conference (EuroPLOP 2004),* UVK.

Bass, L., Clements, P. and Kazman, R. (1999) *Software Architecture in Practice*, Addison Wesley.

Beck, K. and Johnson, R. (1994) 'Patterns generate architectures', *Proceedings of European Conference on Object-Oriented Programming (ECOOP'94)*, *Lecture Notes in Computer Science*, Vol. 821, Springer-Verlag, pp.139–149, Available at: http://st-www.cs.uiuc.edu/users/patterns/patterns.html and http://citeseer.nj.nec.com/27318 .html.

Berlage, T. and Genau, A. (1993) 'A framework for shared applications with a replicated architecture', *Proceedings of the Sixth Annual ACM Symposium on User Interface Software and Technology (UIST'93)*, New York, NY: ACM Press, pp.249–257.

Buschmann, F. and Henney, K. (2002) 'A distributed computing pattern language', *Proceedings of the Seventh European Pattern Languages of Programming Conference (EuroPLoP 2002)*, UVK.

Buschmann, F. and Henney, K. (2003) 'Explicit interface and object manager', *Proceedings of the Eighth European Pattern Languages of Programming Conference (EuroPLoP 2003)*,UVK.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996) *Pattern-Oriented Software Architecture, A System of Patterns*, Vol. 1, John Wiley & Sons.

Calvary, G., Coutaz, J. and Nigay, L. (1997) 'From single-user architectural design to PAC*: a generic software architecture model for CSCW', *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'97)*, New York, NY: ACM Press, pp.242–249.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J. (2002) *Documenting Software Architectures: Views and Beyond*, Addison Wesley.

Dewan, P. (1999) 'Architectures for collaborative applications', M. Beaudouin-Lafon (Ed). *Computer Supported Co-operative Work*, *Trends in Software*, Chapter 7, New York, NY: John Wiley & Sons, pp.169–193.

Dewan, P. and Choudhary, R. (1995) 'Coupling the user interfaces of a multiuser program', *ACM Transactions on Computer–Human Interaction*, Vol. 2, No. 1, pp.1–39.

Edwards, W.K. and Mynatt, E.D. (1997) 'Timewarp: techniques for autonomous collaboration', *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'97)*, New York, NY: ACM Press, pp.218–225, Available at: http://doi.acm.org/10.1145/258549.258710.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley.

Graham, T.N., Urnes, T. and Nejabi, R. (1996) 'Efficient distributed implementation of semi-replicated synchronous groupware', *Proceedings of the Ninth Annual ACM Symposium on User Interface Software and Technology (UIST'96)*, New York, NY: ACM Press, pp.1–10.

Guicking, A., Tandler, P. and Avgeriou, P. (2005) 'Agilo: a highly flexible groupware framework', *Groupware: Design, Implementation, and Use. Proceedings of the 11th International Workshop on Groupware, CRIWG 2005*, Porto de Galinhas, Brazil, LNCS 3706, Berlin, Heidelberg: Springer-Verlag, pp.49–56, Available at: http://www.ipsi.fraunhofer.de/concert/projects/agilo.

Gutwin, C., Roseman, M. and Greenberg, S. (1996) 'A usability study of awareness widgets in a shared workspace groupware system', *Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work (CSCW'96)*, New York, NY: ACM Press, pp.258–267.

Haake, J.M. and Wilson, B. (1992) 'Supporting collaborative writing of hyperdocuments in SEPIA', *Proceedings of the Conference on Computer-Supported Cooperative Work*, ACM Press, pp.138–146, Available at: http://doi.acm.org/10.1145/143457.143472.

Hill, R.D., Brinck, T., Rohall, S.L., Patterson, J.F. and Wilner, W. (1994) 'The rendezvous architecture and language for constructing multiuser applications', *ACM Transactions on Computer–Human Interaction*, Vol. 1, No. 2, pp.81–125, Available at: http://doi.acm.org/10.1145/180171.180172.

Krasner, G.E. and Pope, S.T. (1988) 'A cookbook for using the model-view-controller user interface paradigms in smalltalk-80', *Journal of Object Oriented Programming*, Vol. 1, No. 3, pp.26–49.

Lukosch, S. (2002) 'Adaptive and transparent data distribution support for synchronous groupware', *Proceedings of CRIWG'02*, Heidelberg, New York: Springer, pp.255–274.

Lukosch, S. (2003) 'Transparent and Flexible Data Sharing for Synchronous Groupware', Band 2 in Schriften zu Kooperations- und Mediensystemen, Josef Eul Verlag GmbH, Lohmar–Köln.

Lukosch, S. and Schümmer, T. (2004) 'Patterns for managing shared objects in groupware systems', *Proceedings of the Ninth European Pattern Languages of Programming Conference (EuroPLoP 2004)*, UVK.

Microsoft Corporation (2005) '.NET platform', Available at: http://www.microsoft.com/net.

Object Management Group (2004) *Common Object Request Broker Architecture (CORBA) Specification*, Available at: http://www.omg.org/technology/documents/formal/corba_iiop.htm.

Patterson, J.F. (1995) 'A taxonomy of architectures for synchronous groupware applications, workshop on Software architectures for cooperative systems CSCW'94', *ACM SIGOIS Bulletin Special Issue Papers of the CSCW'94 Workshops*, Vol. 15, No. 3.

Phillips, W.G. (1999) 'Architectures for synchronous groupware', *Technical Report 1999-425*, Queen's University, Kingston, Ontario K7L 3N6, Available at: http://phillips.rmc.ca/greg/pub/.

Roseman, M. and Greenberg, S. (1996) 'Building real time groupware with GroupKit, a groupware toolkit', *ACM Transactions on Computer–Human Interaction*, Vol. 3, No. 1, pp.66–106.

Roth, J. (2000a) 'DreamTeam': a platform for synchronous collaborative applications', *AI and Society*, Vol. 14, No. 1, pp.98–119.

Roth, J. (2000b) 'A taxonomy for synchronous groupware architectures', *CSCW 2000 Workshop "Which Architectureor What" Toward a Property-Based Selection of Software Architectures for Cooperative Systems*, Available at: http://www.cs.queensu.ca/~cscw2000/positionPapers.html.

Rubart, J. and Dawabi, P. (2004) 'Shared data modeling with UML-G', *International Journal of Computer Applications in Technology*, Vol. 19, Nos. 3/4, pp.231–243.

Schmidt, D.C., Stal, M., Rohnert, H. and Buschmann, F. (2000) *Pattern-Oriented Software Architecture, Patterns for Concurrent and Distributed Objects*, Vol. 2, John Wiley & Sons.

Schuckmann, C., Kirchner, L., Schümmer, J. and Haake, J.M. (1996) 'Designing object-oriented synchronous groupware with COAST', *Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work (CSCW'96)*, New York, NY: ACM Press, pp.30–38, Available at: http://doi.acm.org/10.1145/240080.240186.

Schuckmann, C., Schümmer, J. and Seitz, P. (1999) 'Modeling collaboration using shared objects', *Proceedings of International ACM SIGGROUP Conference on Supporting Group Work (GROUP'99)*, New York, NY: ACM Press, pp.189–198. Available at: http://www.opencoast.org.

Schümmer, T. (2003) 'Gama – a pattern language for computer supported dynamic collaboration', in K. Henney and D. Schütz (Eds). *EuroPLoP 2003, Proceedings of the Eighth European Conference on Pattern Languages of Programs*, UVK.

Shaw, M. and Clements, P. (1997) 'A field guide to boxology: preliminary classification of architectural styles for software systems', *Proceedings of the 21st International Computer Soft-Ware and Applications Conference (COMPSAC)*, pp.6–13.

Shaw, M. and Garlan, D. (1996) *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.

Stewart, J., Bederson, B.B. and Druin, A. (1999) 'Single display groupware: a model for co-present collaboration', *Proceeding of the CHI 99 Conference on Human Factors in Computing Systems (CHI'99)*, New York, NY: ACM Press, pp.286–293. Available at: http://www.cs.umd.edu/hcil.

Sun Microsystems, Inc. (2005) 'Java 2 Platform, Enterprise Edition (J2EE)', Available at: http://java.sun.com/j2ee.

Tandler, P. (2004) 'The BEACH application model and software framework for synchronous collaboration in ubiquitous computing environments', *Journal of Systems and Software*, Special Issue: *Ubiquitous Computing*, Vol. 69, No. 3, pp.267–296. Available at: http://ipsi.fraunhofer.de/ambiente/publications/ and http://authors.elsevier.com/sd/article/S0164121203000554.

Taylor, R.N., Medvidovic, N., et al. (1996) 'A component- and message-based architectural style for GUI software', *IEEE Transactions on Software Engineering*, Vol. 22, No. 6, pp.390–406.

Urnes, T. and Graham, T.N. (1999) 'Flexibly mapping synchronous groupware architectures to distributed implementations', *Proceedings of Design, Specification and Verification of Interactive Systems (DSV-IS'99)*, Heidelberg, New York: Springer, pp.133–147, Available at: http://dundee.cs.queensu.ca/~graham/stl/pubs.

## Note

[1]It is reminded that we do not address the specific issues of Single Display Groupware (Stewart et al., 1999) here.