

Review of the available CSCW Toolkits and frameworks

1. ABSTRACT

This document is intended to be a short review of the existing frameworks to develop collaborative applications. We will focus on synchronous collaboration in the field of mobile communications like wireless communication between small devices, PDAs, smart phones, etc...

We first speak rapidly about the different kinds of systems that we can find in CSCW (Computer Supported Cooperative Work) and then present some of these frameworks targeted at Java applications. Finally we try list the services that may be needed in a CSCW application and the techniques that may be used to achieve them.

2. CSCW - DIFFERENT KINDS OF SYSTEMS

The usual classification of CSCW systems is based on the place and the time of the use: the users can work in the same place but at different times, at the same time but in different places, in different places and at different times and of course at the same time and in the same place. In the following we mainly focus on the time aspect: synchronous or asynchronous work.

Some of the toolkits are designed for only one or two of these different kinds of collaborative work while others try to provide functionalities for all of them.

2.1 Asynchronous work

The users work at different times.

The toolkits that focus on asynchronous work are generally data-oriented: they provide functionalities to share data among users (usually in different places) and are usually based on data replication in order to allow the users to work in a fully disconnected mode and synchronize their data later. These systems thus use merging and synchronization policies to maintain the consistency. These policies may be fixed by the system or depend on the semantic of the application that use it.

Examples of data sharing systems: Bayou [1], Coda [2], Lotus Notes [3] documents bases,...

2.2 Synchronous work

The users work at the same time.

One of the most common, and simple, kind of synchronous systems are the Interface/Windows sharing systems. These allow sharing the graphical interface of an application among several users. The main advantage is that the application may not need to be designed for collaborative work. The user inputs are generally also duplicated so that each participant can use the application (which in fact runs only in one of the computers). This approach is fundamentally suited to WYSIWIS (What You See Is What I See) systems and does not support the separation of an object and its graphical representation. For example two different users cannot have different views depending on their devices or their

NOKIA Research Center
Pierre Rust

April 2001

status in the group (they may have different right policies, but only in a quite limited way). This approach also requires a quite large communication bandwidth.

Examples of Window/Interface sharing systems: XTV [4], DistView [5], ...

Other toolkits for synchronous collaborative work are operation-based: these automate group communication by using multicasted RPC. They provide functionalities that allow a procedure call to be automatically invoked on multiple hosts. Usually the methods that may change the state of an object are broadcasted to each participating application. Using this principle, if all the users start with an object in the same state, the consistency will be achieved (assuming that the broadcasted operations are atomic). Concurrency control (e.g. with lock-based systems) may also be needed, depending on the type of the shared objects, on the latency of the communication medium, on the desired level of consistency,...

However these systems cannot be used to restore consistency and are thereby are not suited for systems where the users may not work together all the time or where disconnections may happen quite often (like in the case of mobile networking for example).

Examples: Colab [6] (Xerox), GroupKit [7], Corona [8], ...

Some other toolkits take a more data-centric approach by providing functionalities to share the objects used by the application. These objects are generally system-defined abstractions (for example: dictionary-style data structure in GroupKit , glyphs in Manifold [10], Sync [11] replicated classes,...). When using these abstractions to implement the data used by an application, the system will automatically keep all the replicas consistent and the application does not need to implement any kind of collaborative services but will simply use the merging framework. Moreover, the data-centric approach also allows to separate the data from the way it is visualized on the screen, and thus helps to support the heterogeneity of the access devices.

However it could be difficult to cast a data structure into a predefined set of system-defined abstractions. There have been some try to allow this kind of services to work with arbitrary programmer-defined types but it seems bring to also other problems (e.g. overload) and some of the abstractions proposed in seems to be flexible enough to be used to encode virtually any data structure (cf. The XML-based glyphs in Manifold).

Another concern is the fact that the implementation choices made in the toolkit may interfere in the application and sometimes not be suited for it. Some toolkits address this problem by providing the possibility to the developer of the application to partially change the behavior and/or the implementation of the toolkit (for example Prospero [12] uses computational reflection for that.).

Examples: GroupKit shared environments, Sync [11], Prospero [12], Disciple [9], ...

Most of the toolkits we are going to study belong to this last family.

3. THE DISCIPLE & MANIFOLD FRAMEWORKS

3.1 The DISCIPLE Framework

The DISCIPLE (Distributed System for Collaborative Information Processing and Learning) framework is a communication infrastructure based on a replicated architecture for

NOKIA Research Center
Pierre Rust

April 2001

groupware. It provides functionalities for real-time groupware applications. It enables sharing of arbitrary Java applications in distributed group work. The system itself is based on Java (1.2 for the current version) and is thus platform-independent. DISCIPLÉ is application-centric in the sense that users share Java components.

DISCIPLÉ is free and the source-code is available.

3.1.1 Applications

The system focuses on sharing applications and data and does not provide group communication channel like audio or video conferencing tool. The applications targeted by DISCIPLÉ are Java Beans and applets and the type of collaboration is synchronous collaboration (also called *same-time-different-place*). DISCIPLÉ supports applications developed for it: the *CABs* (*Collaboration Aware Beans*), which are aware and take advantage of the feature of DISCIPLÉ like concurrency control and merging. However DISCIPLÉ also supports typical Java Components, called *CUABs* (*Collaboration Unaware Beans*), which operate as if used by a single user.

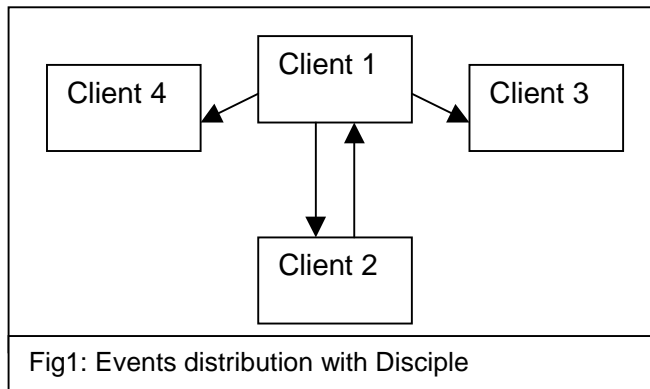
3.1.2 System architecture

The system architecture is a mixture of client/server and peer-to-peer architecture and is based on a replicated architecture for groupware. The coarse architecture also corresponds to the three-tiered architecture (*Presentation – Application Logic – Storage*). The system is divided into two software applications:

1. The *Virtual Desktop* that is run by each client;
2. The *Place Server* that run on a server and corresponds to the storage part. However it is desirable but NOT an essential component of collaboration in DISCIPLÉ (you can run an application without any central server). It serves as a "phone book" of all collaborators and historical record of the meetings. It also provides information to new/late comers so that they can join a meeting, and monitors the memberships. It may also be used for failure notification and recovery.

3.1.3 Consistency and Event management

To maintain the consistency across all copies of the shared application, the user events on each site should not be processed before all applications can consistently process the event. This issue is closely related to the event distribution model. In DISCIPLÉ, one of the clients acts as serialization point that distributes events for all other clients. On the figure 1 for example, the client 1 have the role of the *event merger*: it receives all the events occurring at the event posters (that is to say, at all the clients) manipulate them (this include reordering, concurrency control,...) and distribute them in the same order to every client. This approach allows supporting mobile team that may not have access to a distant centralized server (when working with a local network for example). This architecture also requires a lower network traffic than a fully distributed model in which each client distributes its own events to all the other clients.



DISCIPLER uses a *collaboration bus* for the event distribution, implemented on top of CORBA. This Bus corresponds to the application logic tiers and implements the concept of *nodes* and *places*. A *node* represents an organization and is used to maintain *users* and *places* information. A *place* corresponds to what is usually called a session and is *where* a collaborative session can occur.

The bus is made of three different kinds of communication channels: *posters*, *merger* and *announcer*. The *poster* and *merger* channels are used to distribute the user-generated events while the *announcer* channels deal with *node*-level information (status of the system, a user entering or leaving a place...).

3.1.4 User Interface and Applications

The entire interface use exclusively the Swing toolkit due to the problems with heavyweight Java AWT toolkit.

In the case of CUAB, to intercept all the event generated by the user, DISCIPLER uses a transparent component (GlassPane available with Swing).

For CAB, the Bean must comply with design guidelines. Communication bridges are used to provide communication between different Beans (loaded at run-time). The bytecode for these bridges is automatically generated and loaded at run-time by examining the bean properties using the introspection mechanism of Java.

DISCIPLER already provides some collaboration component such as telepointers, radaviews (for information about awareness tools, see [29]), concurrency control algorithms.

3.2 The Manifold framework

A key goal of the MANIFOLD framework, which is build on top of DISCIPLER, is to enable *collaboration in heterogeneous computing environments*. It should make possible to perform collaborative work among several different kinds of devices with possibly very different capabilities. This heterogeneity is most visible in network bandwidth and in display capabilities.

Display heterogeneity is addressed by using a data-centric approach: all the users share the same underlying model or data but different visualization methods are used to adapt the representation of the data to the capabilities of each devices used.

Network heterogeneity, from the bandwidth / capacity point of view, is addressed by applying transform methods to the data before the transmission. It could be methods for

NOKIA Research Center
Pierre Rust

April 2001

compressing information (primarily for audio and video) or simplifying it to meet the user display requirements.

3.2.1 Data structure

The key concept of the common data structure is a *glyph*, which represents all objects that have geometry and may be drawn. A glyph is essentially a container of *<property,value>* pairs. Adding or removing properties and hierarchical data abstraction can control the size of a glyph to suit the particular computing capabilities of a given device. Glyphs also implement the composite pattern [25] and thus may be an aggregate containing multiple leafs or aggregate glyphs.

There are only very few operations that apply on a glyph tree (delete, create vertex,...). For each of these operations the glyph produces an event that can be used for consistency and synchronization purpose, based on DISCIPLINE.

3.2.2 Dynamic adaptation to the context

As Beans do not allow dynamic adaptation, Manifold uses the dynamic loading of classes of Java to achieve this goal. It also uses XML [27] as the language to describe the composition of the applications: XML is used to represent the glyph tree and to "wire" glyph and behavior together.

The Model-View-Controller pattern [26] is used for the representation of data. A glyph may have a corresponding GlyphView and stylesheets in XLS are used to generate, from a glyph-tree, an GlyphView-tree suited for a particular device, view, ... A viewer can then be applied to this tree and generating the representation.

Since the glyph-tree may not be exactly the same on every site (especially in the case of heterogeneous domains), the glyph operations need also to be transformed. To make these transformations easier the operation are also represented as XML fragment on which XSL stylesheets may be applied. These transformations can occur either in a server or on each client.

3.3 CSCW-related services

We list here the CSCW-related services that we have identified in these frameworks.

3.3.1 Maintaining the consistency

Maintaining the consistency is the most obvious service that users need to be able to work on the same data. This usually includes synchronizing, merging and currency control.

3.3.2 Supporting heterogeneous environments

A system that supports heterogeneous environments allows to work with several different kinds of devices with possibly very different capabilities.

3.3.3 Awareness

The awareness [29] service includes all the tools and methods used to allow an user to see what the activity of the other users (e.g. availability and current work area).

NOKIA Research Center
Pierre Rust

April 2001

3.3.4 Work in a local network

Working in a local network means that the users do not need a connection to a central server or access provider to use the system. Thus they may work anywhere as long as they can connect one to the others.

4. THE JAVA SHARED DATA TOOLKIT

The JSDT [13] is an official toolkit from Sun for developing collaborative application with Java. It provides a set of API to design run-time collaborative applications. JSDT is data centric (the applications can share data elements).

JSDT support full duplex multipoint communication among an arbitrary number of connected applications. It also provides support of multicast message communications, the ability to ensure uniformly sequenced message delivery and a token-based distributed synchronization mechanism. It also provides the ability to share byte arrays amongst the members of a session. Thus, using Java serialization any kind of object may be shared using this system.

However JSDT is not especially targeted at mobile devices/computing.

Since the version 1.5 JSDT is free, including commercial use (the current version is 2.0) but the implementation source-code is not available.

4.1 Communication in JSDT

The design of JSDT is independent of the underlying implementation. Nothing is specified on how the various components of JSDT applications communicate with each other. This is left upto the individual implementations. Three implementations are provided by Sun: sockets, http and LRMP (lightweight reliable multicast protocol [14]) package from INRIA to send packets of information between collaborating applications). It seems that there is also an implementation for iBus [15] (which exists for the Symbian platform!), and it should be possible to make (or find) one for Cobra.

4.2 Basics

JSDT use a centralized-server architecture.

The Registry maintains references to all JSDT objects. The main JSDT objects are Clients, Sessions, and Channels and Data.

A Client is an object that wishes to communicate with other Client objects;

A Session is a conceptual collection of Clients that may (or may not) communicate with each other;

A Channel is a line of communication between Clients. A Channel allows a Client to send data to all Clients (including itself), all other Clients (excluding itself), or to a particular individual Client. Clients can join more than one Channel at a time;

A Data is a discrete unit of data (array of bytes) that is sent by a Client over a Channel to all of the Clients which have currently registered an interest in receiving data on the given Channel.

NOKIA Research Center
Pierre Rust

April 2001

Although JSDT use a server based architecture, there is no Server in the JSDT toolkit. Servers are in fact Clients in JSDT terminology, but with different code as the other Client (in the classical meaning) to give them server functionality.

4.3 Related technologies from Sun

Some other API from SUN may be used to develop collaborative applications or at least to implement the communication part. These are the Java Messaging System (JMS [16], however, it is designed for asynchronous work) and JavaSpaces [17].

4.3.1 JavaSpaces

"**JavaSpaces technology** is a simple unified mechanism for dynamic communication, coordination, and sharing of objects between Java technology-based network resources like clients and servers. In a distributed application, JavaSpaces technology acts as a virtual space between providers and requesters of network resources or objects. This allows participants in a distributed solution to exchange tasks, requests and information in the form of Java technology-based objects. JavaSpaces technology provides developers with the ability to create and store objects with persistence, which allows for process integrity." SUN - [17]

JavaSpace is based on RMI [18] and is a JINI [19] application. It is targeted at small local networks with little bandwidth.

ANTS [20] may also be used, which is a facade API that is meant to simplify the access to publish/subscribe notification services. ANTS is a simple layer to different middleware services like JMS, Elvin [21] (another messaging service), and JSDT.

4.3.2 Javagroups

JavaGroups is a toolkit for reliable group communication that can be used to build collaborative applications. It mainly implements an abstraction layer of communication middleware and adds state replication. It is in this sense not really a collaboration framework but may usefully to implement the communication part of this type of applications.

JavaGroups is an OpenSource [29] project.

4.4 Communication & JChannel

JavaGroups has its own protocol stack (Jchannel [23]) which currently does not yet fully support reliability guarantees. It can also run over iBus and Ensemble [28], which should fully supports various types of reliability guarantees.

The **JChannel** protocol stack consists of a number of layers, each implemented as a Java class. When a new stack is created, each of the layer's name is the name of a corresponding Java class which will be loaded. An instance of that class will then be added to the linked list of protocol layers. Any message traveling up or down the stack has to pass through each layer, and each layer may perform some computation on a message, e.g. reordering it (for synchronized delivery), encryption/de-encryption, computing and appending a checksum etc.

Each layer has to subclass a *protocol interface*, which governs how messages are passed up and down the stack. Methods of the interface (such as `Up()` or `Down()`) are called

whenever a message passes through the layer. An implementor may override these methods (and others) to perform layer-specific chores. (The documentation on writing protocol layers will follow shortly)

A collection of layers determines the properties of the protocol stack: a multimedia conferencing application might require a *control channel* (e.g. for floor control) which has to provide total order and loss-less delivery of all messages, and one or more *data channels* to disseminate video and audio to all participants. With the data channels, loss of some video frames/audio packets is tolerable; however, delay (e.g. because of message retransmission) would not be tolerable. The benefit of **JChannel** is that it can be configured according to the properties required by the application using it.

4.5 Functionalities

Its functionality includes sending messages to all members of a group, ensuring that each member receives the same sequence of messages in the same well-defined order. Its basic abstraction is a *channel*, which is similar to BSD sockets: clients can connect to the channel giving the name of the group they want to join, send and receive messages, retrieve all members currently joined, and receive notifications when members join or leave. All channels with the same name 'find' each other and a message sent via a channel will be received by all channels that have joined the same group (i.e. have the same name).

5. SYNC

Sync [11] is a Java-Based framework for developing collaborative applications for wireless mobile systems. Sync is based on replication and offers synchronization-aware classes based on existing Java classes, using these classes the application developed will automatically have facilities like synchronization, merge,...

Programmers may also extend the Sync-provided classes to create new replicated classes; either to add functionality or modify a class's merge policy. They also can define the notion of conflict and specify the conflict resolution on the basis of the application's structure and semantics.

Sync supports fully disconnected operation and employs centralized, asynchronous synchronization. However it seems to be possible to also use it as a basis for synchronous application as we have seen an example of synchronous collaborative editor based on Sync.

Sync is a Java 1.1 application. *The last version seems to have been released in 1998 but, although we found several articles about it, we were unable to find neither source nor the binary!* However, the model seems to be quite interesting.

5.1 CSCW-related services

5.1.1 Disconnected work

A system that allows fully disconnected work can be very useful in the case of low quality network with frequent disconnection or high connection price. It usually requires data replication.

6. OTHERS

6.1 Flexible JAMM

Flexible JAMM [24] supports sharing of single-user applets in synchronous collaboration. It is also based on replication and event broadcasting.

However it relies on a custom modified version of the JDK 1.1 (use of a customized version of some base classes of AWT, especially for the events management) that make it non-easily portable. We will thus not investigate it further.

7. CONCLUSION

After having found, and investigated several frameworks for collaborative computing in Java, it seems that only two of them are really advanced usable tools : JSDT and Disciple. Disciple is the only one that has a real extension to support mobile computing, however the only application presented by the authors works on palm devices but relies on a pc-based server. Thus it may still be quite heavy for devices with little memory. JSDT is supposed to be relatively lightweight but we have seen no realization on mobile devices.

Some principles, from the conceptual and from the design point of view (like for example event broadcasting, replication, non strict WYSIWIS, ...) are stressed by most of these frameworks. Thus, even if we do not rely on these frameworks, many ideas, principles and/or architecture parts may be reused when working on a small demonstration application.

7.1 CSCW services

We try to list here the services that may be needed in a CSCW system, and the corresponding techniques that **may** be used to achieve them.

What (services)	How (techniques)
Maintaining the consistency of the data	Concurrency control (lock, ordering, ...) Synchronization Merging Events multicasting / broadcasting systems
Heterogeneous computing	Data-based systems – data / view separation Transformations techniques
Local networks	Non-centralized architecture (at least without distant access provider)
Disconnected work	Data replication Merging & conflict resolution
Synchronized work	Concurrency control

What (services)	How (techniques)
Weak WYSIWIS	Data-based systems
Awareness	Central server (to maintain status information)

8. REFERENCES

- [1] Bayou: <http://www.parc.xerox.com/csl/projects/bayou/default.html>
- [2] Coda: <http://www.coda.cs.cmu.edu/>
- [3] Lotus Notes: <http://www.lotus.com/>
- [4] XTV: H. M. Abdel-Wahab and Mark A. Feit. XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration. In Proceedings of Tricomm `91, April 1991.
- [5] DistView: A. Prakash and H. Shim. DistView: Support for building efficient collaborative applications using replicated objects. Proc. ACM 1994 Conf. Computer Supported Cooperative Work, pages 153--164, Oct. 1994.
- [6] Colab: Boley, H., Hanschke, P., Hinkelmann, K., and Meyer, M. CoLab: A Hybrid Knowledge Representation and Compilation Laboratory. Tech. Rep. RR-9308, DFKI, Jan. 1993.
- [7] GroupKit: Roseman, M. and Greenberg, S., "GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications", Proc. ACM CSCW 92, Nov 1992.
- [8] Corona: R. W. Hall, A. Mathur, F. Jahanian, and A. Prakash and C. Rasmussen. Corona: A communication service for scalable, reliable group collaboration systems. In ACM Conference on Computer-Supported Cooperative Work. ACM Press, 1996.
- [9] Disciple: W. Wang, B. Dorohonceanu, and I. Marsic. Design of the DISCIPLINE synchronous collaboration framework. In Proceedings of the 3rd IASTED International Conference on Internet, Multimedia Systems and Applications (IMS'A'99), Nassau, The Bahamas, pp.316-324, October 1999.
- [10] Manifold: I. Marsic. DISCIPLINE: A Framework for Multimodal Collaboration in Heterogeneous Environments. To appear in ACM Computing Surveys, 1999.
- [11] Sync: J. P. Munson and P. Dewan. Sync: A Java framework for mobile collaborative applications. Computer, 30(6):59--66, June 1997.
- [12] Prospero: <http://www.parc.xerox.com/csl/members/dourish/thesis.html>
- [13] JSMT: <http://java.sun.com/products/java-media/jsmt>
- [14] LRMP: <http://webcanal.inria.fr/lrmp/>
- [15] iBus: <http://www.softwired-inc.com/products/mobile/mobile.html>
- [16] JMS: <http://java.sun.com/products/jms>
- [17] JavaSpace: <http://java.sun.com/products/javaspaces/>
- [18] RMI: <http://java.sun.com/products/jdk/rmi/>

NOKIA Research Center
Pierre Rust

April 2001

- [19] JINI: <http://java.sun.com/products/jini>
- [20] ANTS: <http://ants.etsi.es>
- [21] Elvin: <http://elvin.dstc.edu.au/>
- [22] JavaGroups: <http://www.cs.cornell.edu/home/bba/javagroups.html>
- [23] Jchannel: <http://www.cs.cornell.edu/Info/People/bba/user/node77.html>
- [24] Flexible JAMM: <http://simon.cs.vt.edu/JAMM/>
- [25] Composite pattern: E. Gamma, R. Helm, R. Johnson, and J. Vlissides. ***Design Patterns: Elements of Reusable ObjectOriented Software***. Addison-Wesley, Reading, Massachusetts, 1995.
- [26] Model-View-Controller pattern: E. Gamma, R. Helm, R. Johnson, and J. Vlissides. ***Design Patterns: Elements of Reusable ObjectOriented Software***. Addison-Wesley, Reading, Massachusetts, 1995.
- [27] XML: <http://www.w3.org/TR/REC-xml>
- [28] Ensemble: <http://www.cs.cornell.edu/Info/Projects/Ensemble/>
- [29] OpenSource: <http://www.opensource.org/>
- [30] Awareness:
<http://www.cpsc.ucalgary.ca/projects/grouplab/people/carl/research/awareness.html>