Why, What, Where, When:
Architectures for Cooperative Work on the World Wide Web

Category: **PAPER (final copy)**

Authors: **Devina Ramduny**
School of Computing, Staffordshire University
PO Box 334, Beaconside,
Stafford, ST18 0DG
D.Ramduny@soc.staffs.ac.uk
Tel: 01785 353255

Alan Dix
School of Computing, Staffordshire University
PO Box 334, Beaconside,
Stafford, ST18 0DG
A.J.Dix@soc.staffs.ac.uk
Tel: 01785 353428

Contact: **Devina Ramduny**

Why, What, Where, When:
Architectures for Cooperative Work on the World Wide Web

Devina Ramduny and Alan Dix

School of Computing, Staffordshire University
PO Box 334, Beaconside, Stafford, ST18 0DG

D.Ramduny@soc.staffs.ac.uk
A.J.Dix@soc.staffs.ac.uk
http://www.soc.staffs.ac.uk/~cmtajd/topics/webarch/

Abstract

The software architecture of a cooperative user interface determines *what* component is placed *where*. This paper examines some reasons determining *why* a particular placement should be chosen. Temporal interface behaviour is a key issue: *when* users receive feedback from their own actions and feedthrough about the actions of others. In a distributed system, data and code may be moved to achieve the desired behaviour — in particular, Java applets can be downloaded to give rapid local semantic feedback. Thus we must choose not only the physical location for each functional component but also when that component should reside in different places.

Keywords: software architecture, CSCW, Internet, caching, replication, applets, feedback, feedthrough, temporal problems, delays

1. Introduction

The world-wide web provides a ubiquitous infrastructure and platform independent interface for developing remote collaborative applications. Although such systems can be developed in an ad hoc fashion it is widely recognised that, for both single-user and multi-user interfaces, an appropriate software architecture is required as an aid for design, portability and maintenance [4,19,23]. In this paper we will investigate some architectural decisions for distributed collaborative applications focusing especially on collaboration over the web.

Software architecture is about dividing systems into components in order to perform certain functionalities — *what* the system can do. But in order to work as a complete system the components must be linked together in such a way that they can communicate effectively with each other. While all the components are running as part of the same program on the same machine these communications are easy. However, as soon as the system is distributed over a network, as is the case with many cooperative systems, components placed at different locations face higher communication costs and delays than those at the same location. Hence the choice of location — *where* the components are placed — has a significant effect on performance.

One of the principle effects of location decisions is on the pace of interaction. For many years temporal issues in interface design have been largely ignored with a few exceptions [10,11,13,17]. However, recently the importance of time and delays has become more widely recognised [18], due no doubt in part to experiences of Internet use. This *when* question is only of importance to the user when it becomes apparent to the user. Thus behavioural issues are the driving force that determine *why* one architectural solution is better than another.

In many systems data is moved about in order to improve interactive performance. Furthermore in the world-wide web Java applets allow code to move and execute on user's own machines. Thus placement decisions for the web are not just about what is placed where, but also about *when* the data and code is at a particular location.

In the next section we start by looking at mature architectures for single-user interfaces which will later be used as a pattern for looking at collaborative architectures. This is followed in section 3 by an analysis of important behavioural issues for collaborative work. We will then return to architectures, looking at what components are necessary in collaborative interfaces, modelled on those found in single-user interfaces and the requirements established in section 3. In section 5, we will look at different placement options, where to place different components in a distributed architecture. Section 6, examines the issue of mobility of data and code and we are able to plot different options in a when / where matrix for each. Finally, in section 7 we look at the way these two matrices interact focusing on the options available for world-wide web applications.

2. Background — single user architectures

Architectures for single-user interfaces such as Seeheim [23] and Arch / Slinky [29] support the partitioning of the application semantics and the user interface functionality. The dialogue component mediates the communication between them. Separation enhances portability, reusability, customisation and adaptability of an application [16]. However, a case against separability is the problem of rapid semantic feedback — modern direct-manipulation interfaces require information to be exchanged extensively between the user interface and the application.

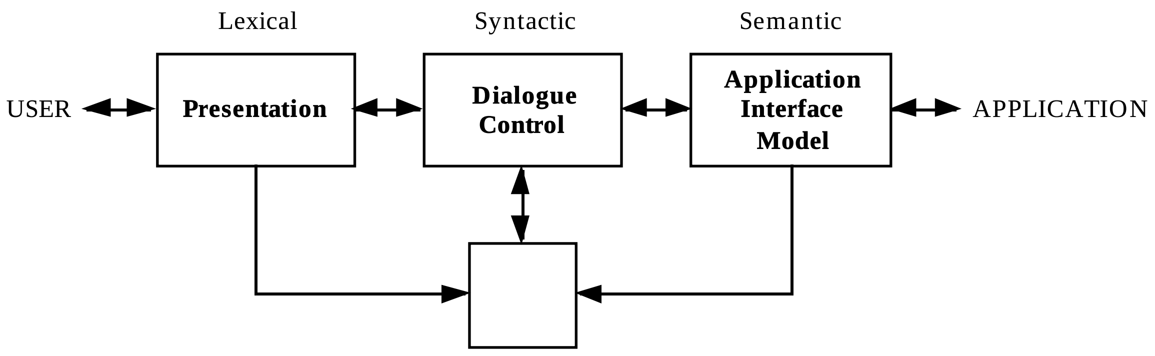

Figure 1. Seeheim model

Figure 1 shows the logical components of the Seeheim model. The *presentation* component is responsible for the external appearance of the user interface while the *application interface model* holds the data and defines the semantics of the application. The *dialogue control* component mediates the interaction between the user and the application to provide semantic feedback. Whilst Seeheim has been developed in various ways, most notably in the Arch / Slinky framework, most user interface architectures preserve some notion of layering between the surface output and input devices and the deep application semantics.

The linear nature of communication between the components in the Seeheim model is often seen as a bottleneck for direct manipulation user interfaces. However the fast-switch represented by the lower box in Figure 1, allows the application to bypass the dialogue component when its state is not affected by output events. The application can then communicate directly with the presentation component and thus provide rapid feedback. But unlike the other functional components, the fast-switch is less well defined and correspondingly more difficult to implement as an architectural feature.

We can view the Seeheim model (and indeed other architectural models) in two ways: either as a conceptual (or logical) architecture, helping us to think about user interface development; or as a physical architecture demanding that there really are components of the system with the named roles and communicating along the paths specified in the architecture.

This is particularly obvious when all the components are placed on the same machine. As we have seen, the decoupling of the user interface functionality from the application functionality via the dialogue control is sometimes difficult to achieve, especially when rapid semantic feedback is required. Consequently aspects of the user interface may ‘leak’ into the application semantics component or vice versa. This is possible as there are typically no constraints to stop, for example, the application semantics from directly calling window toolkit functions. This fluidity of boundaries is recognised in the Arch/Slinky model [29] which still maintains the central role of layering and separability (in fact adding additional layers), but accepts that the precise placement of these layers into coded modules may vary between systems and even between parts of the same system.

When the software is no longer running on a single machine, as is the case in a distributed environment, the application and the user interface components are usually on different machines. One can no longer fudge the boundary and communications between these as they are enshrined in the physical location and network connectivity. As a result the issue of *where* the various components reside is decisive in order to achieve rapid feedback.

In addition to architectures which divide the entire system into a small number of large components, there are many agent-based or object-oriented user interface architectures. However, these either identify individual agents as belonging to one of the traditional layers or include a layering within each agent. For example, the PAC architecture [6] regards each agent as possessing a presentation, abstraction (roughly corresponding to application) and control component.

3. Why — behavioural issues

The reasons for determining *why* a particular placement should be chosen influence the behaviour of an application. The behavioural aspect affect the way users view the display on the screen (presentation). The behaviour of any application depends on its architecture. The most significant behavioural implication enforced by the architectural decisions is often the temporal impact. For instance, if one ignores the temporal issues then from the behavioural viewpoint, the location of the data is not important. However, for performance reasons, it is crucial that there is no perceived lag between any updates to the data and the subsequent changes being reflected on users' displays. Consequently, this may influence the selection of for example, a centralised or a replicated architecture [20]. The rest of this section describes the major behavioural issues which arise within web-based collaborative work.

Triggers and shared objects

A previous study [14,15] highlighted the importance and function of triggers, that is events which initiate activities. An important class of triggers are environmental cues [26], objects in the physical objects which by their presence remind users that activities need to be performed and help maintain the status of ongoing processes. Similarly, in an electronic cooperative setting, triggers can be associated with shared objects, reminding users that some actions have been carried out by others and/or some further actions need to be taken. Furthermore, the coordination of cooperative work can be mediated via shared objects. Although this form of coordination is less explicit than direct communication, it does play an important role. Indeed, in many cooperative processes there may be little direct communication. Instead coordination is mainly achieved by communicating implicitly through the artefact [8].

Feedback

Feedback manifests itself as a response from the display after a user’s actions. It is a common feature of direct manipulation interfaces where objects change their behaviour when they are manipulated by the user. For instance, a button is highlighted when it is clicked onto by the mouse. Feedback may depend on the underlying application semantics. Within the web environment, the feedback loop involves transmission over a network. If the network traffic is high, the delays associated with the feedback will be significant. Consequently, it may be difficult to achieve rapid semantic feedback and acceptable response times.

Feedthrough

Collaborative participants not only interact individually with the system but also with other group members via the shared objects. As a result, it is important to see both the user's own updates (feedback) and the effects of other users' actions (feedthrough). Feedthrough is the reflection of a user's actions on other users' screens [9]. For example, gIBIS [5] allows participants to be aware of any updates through a notification mechanism.

The requirements for feedthrough are not so stringent as for feedback [9]. Feedthrough depends on two major factors: the granularity of the updates and the propagation of those updates. In tightly-coupled cooperative activities, such as group drawing, the granularity of updates is small and rapid feedthrough is vital. In other words, the updates have to be broadcast to all the users after each action. On the other hand, in loosely-coupled applications, the update rates can be reduced significantly. The user who initiated the action still requires rapid feedback but the feedthrough to other users may be less frequent. Because some objects are more significant for obtaining a sense of engagement, concepts of quality-of-service [24] can be applied giving different levels of feedthrough on shared objects within a groupware architecture.

The very nature of cooperative work introduces delays as users have to wait for feedback from their own actions and feedthrough of the actions of others. In addition, with the web, there are further delays and lags which are implicit in the network. Thus the provision of rapid feedback and feedthrough becomes more problematic. Current web-based collaborative applications often weakly support feedthrough even though it is essential in maintaining fluid collaboration.

Awareness

Traditionally, distributed systems have applied different types of transparency to hide information from the users. However, within the context of CSCW, users need precisely that information for effective cooperation. Awareness of individual and group activities is critical for establishing successful collaboration. Different kinds of awareness have been identified in the research literature.

The three major forms that enhance group work are:

a) awareness of the presence of group members and their availability for cooperative work,

b) awareness of the effects of group members actions (i.e. what changes have occurred) and

c) awareness of how changes happen.

Nevertheless, in remote cooperative work, users are often faced with unpredictable timing delays over the network due to remote site failures or network bottlenecks. This may lead to a complete breakdown in work. We therefore require an additional form of awareness:

d) awareness of the state of the communication channels.

An interesting observation which can be made is the fact that awareness of type (b) is basically conveying the notion of feedthrough [12]. Also the pace of feedthrough is directly proportional to the rate of providing awareness of type (c). In other words, we can infer the reasons changes happen by noticing the intermediate steps and the way changes happen. However, both awareness of type (b) and (c) will be negatively affected by network delays and lags.

Awareness of type(c) is not easy to manage especially when the web is used as an asynchronous environment. Some traditional groupware with shared workspaces record who has made the updates and when. Such temporal information is however hard to reconstruct at a distributed level. Even synchronous interaction will pose a similar problem in the event of delays over the communication channel.

Control

Due to the common focus on work, collaborative participants have to access the same data. Therefore some form of control is required to manage the shared data and the shared objects. This will determine the nature of the cooperation dealing with issues such as who can update what, where and when; who can see the changes and whether the changes can be noticed in a reasonable amount of time.

One of the most common control mechanisms is locking including explicit floor control policies [1,28] and implicit locking which is automatically applied when users attempt to access an object. In some systems additional protocols are built on top of the locking mechanisms, such as access rights or roles [21]. Users perform certain tasks depending on the roles they are assigned. Unlike access rights which normally impose a restriction on users functions, roles are more dynamic in nature.

Finally, certain applications do not provide any mechanism for locking, relying on participants using a social protocol to negotiate simultaneous access in a free-for-all situation. However such systems must usually include some form of mechanism to detect conflicts in order to automatically restore consistency or at least alert users.

4. What — architectural components of cooperative systems

The support for collaborative work has seen the development of architectures which present a number of interfaces for simultaneous interaction by multiple users. One of the main functions of cooperative architectures is the presentation and manipulation of shared information by a community of users. As we have discussed earlier in section 2, the separation between the application semantics and the user interface is acknowledged to be a desirable feature for a number of reasons.

Whereas in single-user applications, the logical separation is sometimes ignored to reduce the complexity and speed of development, in collaborative applications, logical separation is a necessity for supporting alternative views of the system for different participants. It is necessary to identify which elements of collaborative interfaces are shared between participants and which are different for each one. This logical separation is also essential when deciding where elements are placed in a networked environment. The web for instance, allows extensions or modifications at the server-end, client-end and the communication protocol, an issue we return to in section 6. Let us now consider how the Seeheim model can be mapped onto cooperative systems.

Presentation

The *presentation* component of collaborative applications must support alternative representations of the users' display. Shared information can be presented as a single view to all the participants (WYSIWIS systems) or it can be viewed differently by different users (multiple views). For instance, a user may view the data in tabular form while another may view it as a graph. Similarly, group members can have their own private view or they can also share views of the display. Some systems allow users to shift between a tightly coupled mode where they share the same presentation and a loosely coupled mode where users can view and scroll independently. In cases where the presentation or view is shared there must be some component of the systems to manage the shared information.

Shared data

The key element in any collaborative system is the shared application data. In the Seeheim model the *application interface* component manages the mapping between application data and the rest of the user interface. This is important as it emphasises the fact that the visualisation of information requires both the raw data and the semantics of the data — in a computational setting embedded in code. In the web setting this aspect is often embedded in CGI scripts which communicate with the user interface component (the web browser) using web pages and forms (dialogue level information). However, Java applets have opened up the possibility of including far more of the application semantics at the user interface itself.

Control

In section 3 we discussed why control mechanisms were necessary to avoid conflicts and maintain consistency. However, behavioural level control itself has be to driven by some lower level control that has to be maintained by the architecture, the most common mechanism for this being locking. In a single-user interface the *dialogue* component is responsible for determining the allowable order of user actions. The control component satisfies a similar role in that it controls the possible order of actions by different participants.

Traditional distributed systems view control as dealing with the problems of distribution and masking such problems from applications [25]. For instance, most distributed systems allow users to know who *can* access which objects but they do not allow users to know who *is* accessing a particular object at a particular moment. The control decisions are thus embedded into the system and hidden from the users. However, due to the dynamic requirements of CSCW applications, one of which is awareness, transparency is the wrong approach.

Because data is shared in collaborative applications, there is a clear distinction between the mechanisms for enabling distribution and sharing (e.g. ability to move an object) and the policies for managing those mechanisms (decisions about when and where the object should be moved to). Effective groupware systems therefore need separate low level control mechanisms to support those higher level control policies. Architectural level control may be either of a centralised or a peer-peer nature and can be supported by a separate server or be part of the shared data's infrastructure.

Notification

In collaborative work users operate simultaneously on the shared data — some users may view some part of the data while others may perform an update. MEAD [3] is an example groupware application which allows detailed level sharing of the scrollbars. Consequently, there is a need to maintain consistency between the users views and the underlying data. Without such consistency feedthrough is lost and users cease to have a common focus for collaborative activity.

In a single-user interface, similar issues arise whenever there are multiple views of the same underlying object. However, because there is ultimately a single locus of control (the user) this can be managed within the *dialogue* component. For example, the PAC architecture [6] has a hierarchy of PAC agents within the dialogue controller which manage consistency between views. In a distributed collaborative setting it is fundamentally more complex to maintain this consistency because of the multiple loci of control.

Notification mechanisms address this problem by informing the presentation component of various updates so that the latter can replicate the changes on the users' display. A low level notification mechanism is therefore required to maintain feedthrough by informing the application of changes to the data and by keeping track of the activities of collaborative participants to support awareness.

5. Where — placement decisions

In the previous section we looked at some of the components of the Seeheim model and how they parallel components required of cooperative systems. However, we did not consider the fast-switch which allowed the application to communicate directly with the presentation component. This is because the fast-switch is not really part of the conceptual architecture (which is perhaps why it is so often omitted in descriptions of Seeheim), but instead is there as an optimisation. In principle all feedback could be routed through the dialogue component with more or less translation and interpretation on the way. The problem with this is that the dialogue component introduces a computational delay between application and presentation thus reducing the pace of feedback.

Arguably this is not a great problem today for single-user single-machine systems as it is often possible to perform several levels of processing and still achieve acceptable interactive response. However, for collaborative systems it is likely that shared data will be stored remotely from the user's workstation — instead of a computational delay we have a network delay. Whenever data is stored remotely from the interface feedback delays are bound to occur.

Unfortunately, we cannot simply add an extra component similar to the fast-switch as it too would have to sit remote from the data or remote from the interface. You can bypass computational components, but not space! The location of data and code (where) inevitably effects the pace of feedback (when).

One solution to this problem is to accept that semantic feedback will be delayed. Instead one can adopt a paradigm of mediated interaction with instant local feedback that the user's action has been recognised followed by subsequent semantic feedback when the effect has occurred remotely [7]. However, this will not be acceptable where user's demand direct manipulation interfaces similar to those for local single-user applications, in which case alternative solutions must be found.

Replication and Caching

Most solutions which aim at providing rapid feedback and increasing the availability of data involve some form of replication or caching (Figure 2). The objective is to bring the shared data closer to users.

Caches are merely temporary repositories and hold an ephemeral copy of the data at any instant. Each user's workstation therefore uses local copies of the shared data (Figure 2b). The actual shared data is held in a central repository. Because there is centralised control over the data, consistency can be easily maintained as each cache only needs to communicate with the central repository.

Replicas on the other hand, are equally valid full copies of the data which are stored locally. Replicas are more persistent than caches as they are the real data. However, it is more difficult to maintain data and interface consistency as replicas have to communicate between each other on a peer-to-peer basis. Replicas are synchronised by sending input from each workstation to each replica (Figure 2a). Consequently the multiple points of updates may lead to race conditions and potential data inconsistency. For example, if a user deletes a selected object in a WYSIWIS group drawing program while another user is changing the selection to a different object, inconsistent interfaces can result due to events arriving in a different order at each workstation.

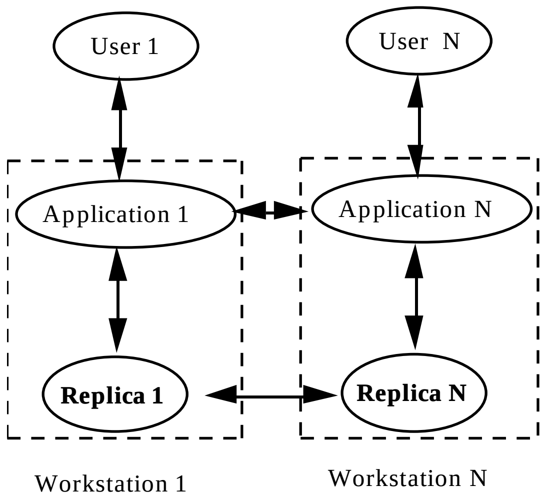

Figure 2(a) Replication

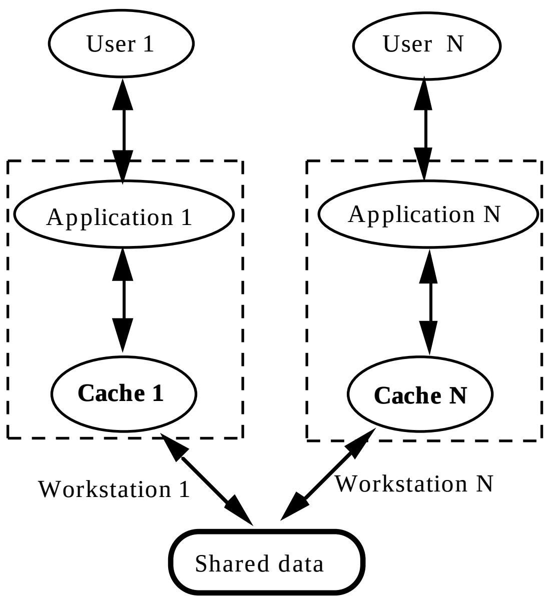

Figure 2(b) Caching

In distributed systems, the traditional approach to replication has been transparency. The system avoids race condition by maintaining consistency among the different copies of the data via some complex synchronisation algorithms. In the event of inconsistencies, the solution is to *rollback* [27] the replica(s) and re-execute the events in temporal order. However, this policy is unacceptable in collaborative interfaces as display screens would already have been updated and alternatives based on transforming updates to prevent rollback have been developed [4].

Control

When rapid feedback is not the major concern, concurrency control mechanisms, such as locking or floor control can be applied to prevent race conditions altogether or tolerate race condition only in situations where users obtain locks or other large scale events [9]. However, real-time synchronous update may demand special-purpose algorithms.

All such mechanisms require meta-data, for example recording who has the lock on which object. This meta-data must itself be maintained and has similar issues as the real data. It can be maintained in a replicated fashion using complex distributed algorithms, or more commonly be maintained using a central server. When the data is stored centrally the same server may deal with both data and meta-data as is usually the case with traditional databases. However, it is also possible to use a separate locking server. For example, the UNIX file system has no in-built locking mechanism, instead applications request locks on remotely stored files from a special process, the lock daemon. Obviously where no off-the-shelf locking is available, or where the locking supplied is unsuitable, application developers are forced to use their own ad hoc locking mechanisms. This is usually the case with web-based cooperative applications.

Notification

Although feedback causes problems, feedthrough is even more difficult — no amount of careful placement of components can change the fact that the user making a change is a long way from other users who see the effects of the change. Happily, we have seen that feedthrough can usually be of a lower pace than feedback, hence ordinary network delays are usually acceptable. What is not acceptable is if changes made by one user are never reflected on other users' interfaces or only do so after a long delay — hence the need for notification mechanisms as discussed in the previous section.

Similar issues arise as for locking. If no notification service is provided then an ad hoc mechanism will be necessary, for example, individual clients may poll one another for changes. Alternatively, a notification service may be incorporated within the data-management infrastructure, for example, Lotus NSP [22] offers a generic data storage and notification server and ALV [19] supports distributed constraint maintenance between shared data and user views. Finally, a stand-alone notification server can be used.

Whichever notification alternative is used there will be meta-data concerning this: what objects are being managed, who wants to know about which object. This meta-data must again be stored in either a replicated or centralised fashion.

Different kinds of remoteness

In a single-user system, when accessing remote data using traditional client-server techniques, it is clear what we mean by local and remote. In a cooperative application we need to be more careful as for each user the definition is different as their own machine is local, but data stored or updated on another user's machine is just as remote. So, if semantic feedback relies on data held at another user's machine, the feedback delays will be as great as for centrally held data, perhaps greater as central servers may have better networks response.

This gets even more complicated when using the web as an infrastructure. Each user may be accessing several web servers as well as other central servers such as databases. To an extent the web makes the physical location of data unimportant, except insofar as the location affects response time. However, the physical location is very important when using Java applets. The security mechanisms of Java only allow the applet to access Internet services lodged on the same machine as the web server that supplied the applet.

In summary for the web we have four kinds of 'remote' application:

- another user's client
- the web server for the current page
- a different server on the same machine as the current web server
- server on a different machine

Thus for the web placement decisions do not stop at local vs. remote, or even client vs. server. The decision about where server software is placed is intimately related to the techniques used to implement client software.

6. When — moving information and code

In a networked environment it is common for data to be dynamically moved or copied in order to improve performance. Also some distributed infrastructures support the migration of objects or code between machines. We will now discuss the various mobility aspects of data and code individually in preparation for considering their interaction in the next section.

Moving data

Consider caching — the 'golden' copy of the data is stored remotely, but a copy is made locally to speed feedback. The fact that data can be copied over networks means that in distributed collaborative applications, the place where shared data is permanently stored is not necessarily the same place as it (or a copy of it) is used. Using the simple local/remote distinction we can classify both the permanent storage place and the place of use giving rise to the matrix in Figure 3.

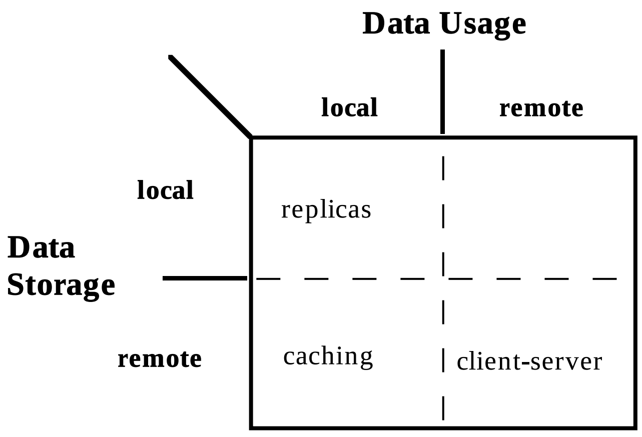

Figure 3. Data Usage v/s Data Storage

This matrix clearly shows the distinction between caching, where the local copy of the data is ephemeral and the 'real' data is central, compared with replication, where the local data is more persistent. Where data is held and used remotely we have traditional client-server interfaces. Only the information necessary to generate the interface presentation of the data is transmitted to the user's local machine. Notice the empty location. In a groupware context, it is highly unlikely to have a scenario where the data is held locally and yet is used or processed remotely. However, such a situation does exist for non-groupware solutions, for example super computers.

Moving code

In a collaborative distributed interface we must also decide where the code for different architectural components resides. In particular, for web-based systems application specific code may run at the server-end (CGI scripts or independently running servers) or at the client-end (applets and helpers or browser plug-ins [30]).

Notice again that in some cases (e.g. CGI scripts) the code is stored remotely, in others (e.g. helpers) it is stored locally. In the case of both CGI scripts and helpers the code executes in the same place as it is stored. However, in the case of Java applets remotely stored code is executed locally. This is a form of migration as is found in many object-based distributed systems.

Code execution and code storage are key architectural options. It is essential to decide where the code gets executed for efficiency reasons in order to provide rapid feedback. Similarly, the rate at which changes to the code occur and the ease of distributing those changes (a form of feedthrough) are affected by where the code is stored.

Using these two axes the matrix in Figure 4 classifies code options using a similar matrix as that we had for data.

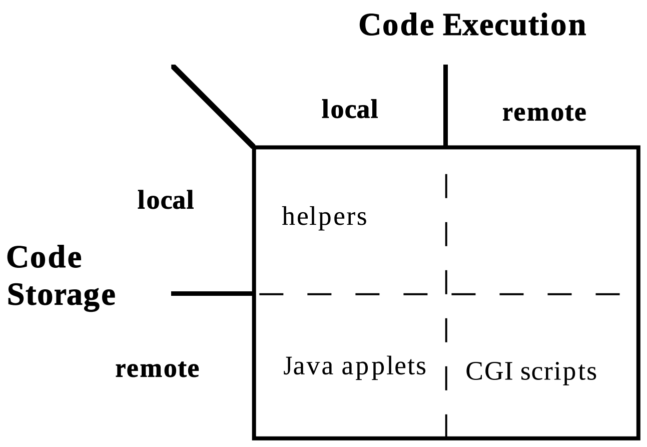

Figure 4. Code Usage v/s Code Storage

As with the data matrix we find a gap in Figure 4. The web does not cater for locally stored code to be executed at the server end and it seems an unlikely option for groupware systems in general. However, in some client-server database applications, quite complex SQL queries can be sent to the server and which may be regarded as a form of locally stored code with SQL queries being executed remotely.

7. World Wide Web — narrowing down the options

We saw that shared data can be stored and used either locally or remotely (Figure 3). Similarly, code can be stored at the client-end (locally) or at the server-end (remotely) and the same applies to code execution (Figure 4). So, for each component of a collaborative application we need to decide where in the respective matrices the code and data for that component resides.

At first this looks as though we have 16 different architectural options to consider for every component as there are 4 possibilities for both code and data. In fact it is not this bad! For general distributed collaborative applications and in particular for the web we can narrow down the potential architectural options.

From the matrices in Figures 3 and 4, we noted that in each there was a gap which appears an unreasonable option for any collaborative application. Thus there are only 3 real possibilities for code and data and at most 3×3=9 combinations.

If we also look at the combinations of code and data the possibilities further reduce. Although data and code can be stored in different places, the code must execute where the data is used. The data and code matrix must ‘agree’ in the location of execution and use (Figure 5). Therefore there is only one possibility for remote execution/use and 4 possibilities (2x2) for local execution/use. We’ll look at the former and latter in turn.

Remote execution and use

The only possibility for remote execution and use in a collaborative application is where both code and data are stored, used and executed remotely (although each could conceivably be stored at different remote sites only coming together for execution/use). A component of this kind could be implemented in several ways.

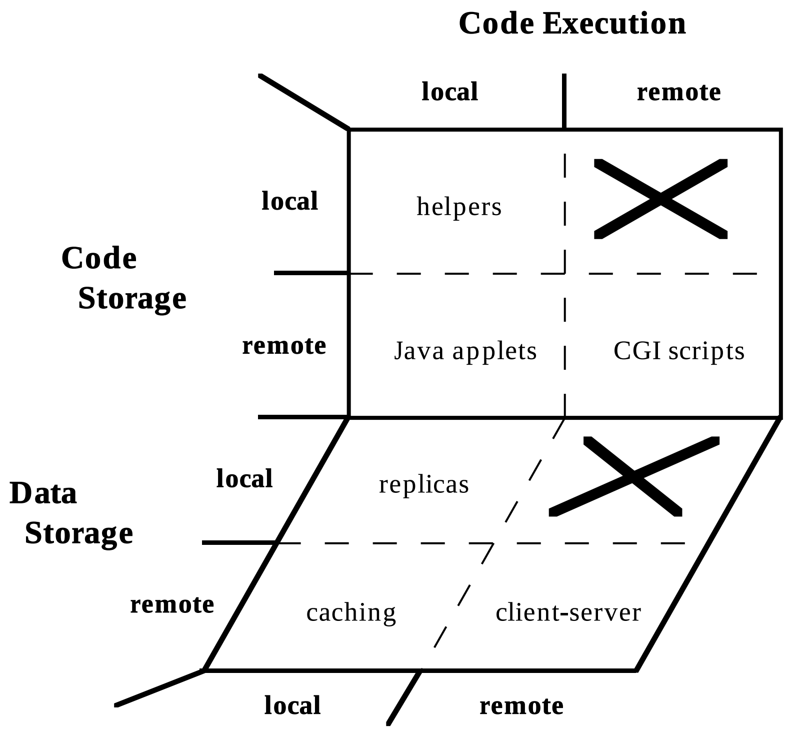

Figure 5. Linked matrices

It may be a traditional transaction-based client/server application using CGI scripts for central processing of transactions. In fact, many web-based repositories are of this form, for example BSCW [2]. Alternatively it may be achieved using a specialised central server as is the case with most chat-based web applications [30]. Note that these two implementation options differ principally in the pace of cooperative interaction they enable.

Local execution and use – applet restrictions

We had 4 storage options for code and data which is locally executed:

a) code local – data local

b) code local – data remote

c) code remote – data local

d) code remote – data remote

In both (a) and (b) we have a helper or stand-alone application using caching or replication to handle shared data. Given the limited ability of most web servers to allow uploading of documents, it is likely that (b) will use a non-web based data base or bespoke server. In both cases the web may act as a way of locating shared resources and initiating a specialised collaborative application, but is not intrinsic to the running application.

For cases (c) and (d) we are principally considering code in the form of Java applets (although other forms of downloaded scripts are available). The security limitations of Java applets mean that they can not access files stored on the user's local machine. This means they cannot operate in mode (c) with permanently locally stored data. Furthermore as they can only connect to a server on the same machine as the web server they were downloaded from, they cannot enter into peer–peer communication (except by using a central switchboard server). Thus they cannot even operate using locally held replicas. All feedthrough must be through a central server at the same site as the web server.

This effectively leaves only case (d) as a truly web-based option and even then only when using a data repository situated at the same location as the applet is stored.

8. Summary

Architectures have been influential within single-user interface design for both construction and conceptualisation. An examination of significant behavioural issues for cooperative interfaces allowed us to identify key components which roughly correspond to those in traditional single-user models. However, the placement of these components within a distributed system leads to conflicts between feedback and consistency. This is commonly dealt with by using caching or replication, both of which bring the shared data 'closer' to the user.

It is now common for web applications to use Java applets to download code to users' own machines. That is code and data may each have a permanent location where they are stored and an ephemeral location where they are executed or used. The resulting storage / use matrix for data and storage / execution matrix for code can be used to examine the placement of each part of a cooperative system.

At first there appear to be many possible combinations of data and code placement within these matrices, but an examination of their interaction within distributed environments in general and the web in particular narrows this down considerably leaving only 2 'real' web-based placement options.

The behavioural and component analysis brought out the importance of various kinds of meta-data, for locking, consistency maintenance and feedthrough. Of particular importance are the notification mechanisms which enable an appropriate pace of feedthrough. The issues surrounding the design options for notification services are too complex to deal with in this paper and are the focus of on-going work.

References

1. Begeman, M., Cook, P., Ellis, C., Graf, M., Rein, G. and Smith, T. (1986) Project Nick: meetings augmentation and analysis. In *Proceedings of CSCW'86* (Austin, Texas), ACM Press.

2. Bentley, R., Horstmann, T., Sikkel, K. and Trevor, J. (1996) The BSCW Shared Workspace System. In *ERCIM workshop on CSCW and the Web* (Sankt Augustin, Germany), GMD / FIT.

3. Bentley, R. (1994) Supporting Multi-User Interface Development for Cooperative Systems. *Ph.D. Thesis, University of Lancaster, UK.*

4. Bentley, R., Rodden, T., Sawyer, P. and Sommerville, I. (1994) Architectural support for cooperative multi-user interfaces. In *IEEE COMPUTER special issue on CSCW*, **27**(5), pp 37-46.

5. Conklin, J. and Bergman, L.M. (1989) gIBIS: A Tool for Exploratory Policy Discussion. In *Journal of American Society for Information Science* (May), pp 200-213.

6. Coutaz, J. (1987) PAC, An Object Oriented Model For Dialog Design. In *Human-Computer Interaction - INTERACT '87*, Eds. H.J. Bullinger and B. Shackel, pp 431-436.

7. Dix, A.J. (1995) Cooperation without (reliable) Communication: Interfaces for Mobile Applications. In *Distributed Systems Engineering*, **2**(3), pp 171–181.

8. Dix, A.J. (1994) Computer-supported cooperative work — a framework. In *Design Issues in CSCW*, Eds. D. Rosenburg and C. Hutchison, Springer-Verlag, pp 9-26.

9. Dix, A.J., Finlay, J., Abowd, G., Beale, R. (1993) *Human-Computer Interaction*, Prentice Hall.

10. Dix, A.J. (1992) Pace and interaction. In *Proceedings of HCI'92: People and Computers VII*, (Sept. York) Cambridge University Press, pp 193-208.

11. Dix, A.J. (1987) The Myth of the Infinitely Fast Machine. In *Proceedings of the Third Conference of the BCS HCI SIG: People and Computers III*, Cambridge University Press, pp 215-228.

12. Dix, A. (1996) Challenges and Perspectives for Cooperative Work on the Web. In *ERCIM workshop on CSCW and the Web* (Sankt Augustin, Germany), GMD/FIT.

13. Dix, A. (1994) Que sera sera — The problem of the future perfect in open and cooperative systems. In *Proceedings of HCI'94: People and Computers IX*, (Glasgow) Cambridge University Press, pp 397-408.

14. Dix, A., Ramduny, D., & Wilkinson, J. (1996) Long-Term Interaction: Learning the 4Rs. In *CHI'96 Conference Companion Proceedings: Human Factors In computing Systems* (Apr. Vancouver, British Columbia), ACM Press, pp 169-170.

15. Dix, A., Ramduny, D., & Wilkinson, J. (1995) Interruptions, Deadlines and Reminders: Investigations into the Flow of Cooperative Work. *RR9509, University of Huddersfield*, available as: <http://www.hud.ac.uk/schools/comp+maths/research/reports/RR9509.html>.

16. Gram, C. and Cockton, G. editors (1996) Design Principles for Interactive Software, *Chapman & Hall, UK.*

17. Gray, P., England, D. and McGowan, S. (1994) XUAN: Enhancing UAN to Capture Temporal Relationships among Actions. In *Proceedings of HCI'94: People and Computers IX*, (Glasgow) Cambridge University Press, pp 301-312.

18. Johnson, C. and Gray, P. editors (1995) Workshop on Temporal Aspects of Usability. In *SIGCHI Bulletin*, **28**(2).

19. Hill, R.D., Brinck, T., Rohall, S.L., Patterson, J.F. and Wilner, W. (1994) The Rendezvous architecture and language for constructing multi-user applications. In *ACM Transactions on Computer-Human Interaction*, **1**(2), pp 81-125.

20. Lauwers, J.C. and Lantz, K.A. (1990) Collaboration Awareness in support of Collaboration Transparency: Requirements for the next generation of shared window systems. In *CHI'90 Conference Proceedings: Human Factors In computing Systems* (Apr. Seattle, Washington), ACM Press, pp 303-311.

21. Leland, M.D.P., Fish, R.S. and Kraut, R.E. (1988) Collaborative document production using quilt. In *Proceedings of CSCW'88* (Sept. Portland, Oregon), ACM Press, New York, pp 206-215.

22. Patterson, J.F. Day, M. and Kucan, J. (1996) Notification Servers for Synchronous Groupware. In *Proceedings of CSCW'96* (Nov. Boston, Massachusetts), ACM Press, pp 122-129.

23. Pfaff, G. and Hagen P.J.W., editors (1985) Seeheim Workshop on User Interface Management Systems, *Springer-Verlag, Berlin*.

24. Rada, R. (1995) Interactive Media, *Springer-Verlag, New York.*

25. Rodden, T. and Blair, B. (1991) CSCW and Distributed Systems: The problem of Control. In *Proceedings of the second European Conference on CSCW,* (Bannon, L. Robinson, M. and Schmidt, K. eds).

26. Rouncefield, M., Hughes, J.A., Rodden, T., & Viller S. (1994) Working with "Constant Interruption" CSCW and the Small Office. In *Proceedings of CSCW'94* (Oct. Chapel Hill, North Carolina), ACM Press, pp 275-287.

27. Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H. and Steere, D.C. (1990) Coda: a highly available file system for a distributed workstation environment. In *IEEE Transactions Computers*, **39**(4), pp 447-459.

28. Stefik, M., Bobrow, D.G., Foster, G., Lanning S. and Tatar, D. (1987) WYSIWIS revisited" early experiences with multiuser interfaces. In *ACM Transactions on Office Information Systems*, **5**(2), pp 147-167.

29. UIMS (1992) The UIMS tool developers workshop: A metamodel for the runtime architecture of an interactive system. In *SIGCHI Bulletin*, **24**(1), pp 32-37.

30. Welie, V.M. and Eliëns, A. (1996) Chatting on the Web. In ERCIM workshop on CSCW and the Web (Sankt Augustin, Germany), GMD/FIT.