8

Software Infrastructures

PAUL DOURISH
Xerox PARC

ABSTRACT

Increasingly, personal computers and workstations come ready "out of the box" to participate as nodes of a distributed computing network. Elements of distributed computing infrastructure, from network file systems and shared printers to high-speed connection backbones, are part of our everyday experiences as users of computers. This chapter discusses software infrastructures for the design of CSCW applications. In particular, it is concerned with how developments in distributed computing and user interface architecture can be exploited in applications that support collaborative activity. The chapter considers a variety of currently-available infrastructure components and discusses how they can be used in collaboration, before going on to suggest a new approach which revises the nature of the relationship between infrastructure and applications.

8.1 INTRODUCTION

CSCW is a highly diverse discipline. From its very beginnings, it has drawn from psychology and sociology as much as from computer science. In turn, within computer science, issues from the areas of network communication and distributed systems have been as important as those from user interface design and usability.

The focus of this chapter is software infrastructure in the design of CSCW systems. By "infrastructure", I mean those elements which lie below the level of the collaborative systems themselves, but which can be exploited in the design of those systems. Explicitly collaborative infrastructures, or collaboration toolkits, are discussed in Chapter 6 of this book [Gre99]. So, many of the infrastructure components that will be discussed here have been (or are being) designed outside the CSCW domain itself. This chapter will take a CSCW perspective on these non-CSCW technologies, and discuss how they can be used in CSCW applications.

Computer Supported Cooperative Work, Edited by Beaudouin-Lafon

Later on, we will also consider implications both for the design of CSCW technologies and the future development of software infrastructures.

8.1.1 Overview

This chapter is organized into two main parts.

First, in Sections 8.3–8.5, we will consider software technologies that provide infrastructure services which can be used in CSCW systems. This will cover both the general use of particular types of infrastructure system, as well as discussing particular tools applicable to CSCW.

The second part (Section 8.6 onwards) will outline new work on CSCW support based on *computational reflection*. This approach provides a way for applications to become involved in aspects of infrastructure, so that the infrastructure can be tailored to the specific needs of particular applications. I have been developing this approach in the design of a prototype CSCW toolkit called Prospero.

8.2 INFRASTRUCTURE ELEMENTS IN CSCW

The nature of CSCW software lends itself to the appropriation of other technological bases as an infrastructure for collaboration. We will consider three particular areas here: first, distributed systems, which support network-wide computation; second, database systems and related concerns in storage and replication; and third, user interfaces to distributed, network-wide applications.

8.2.1 CSCW and Distributed Systems

CSCW software is inherently distributed, and so a variety of techniques and systems developed within the distributed systems community can be fruitfully adopted in CSCW. Aspects of distributed systems technology which are relevant include shared distributed objects, mobile services, replication mechanisms, global coordination, distributed naming schemes and architectural considerations in data and application distribution.

As with most other elements of infrastructure discussed in this chapter, distributed system technologies can be deployed as infrastructure at a variety of levels. On one level, distributed system components can be used to provide a basic set of services *on top of which* CSCW systems will be built. In these cases, the CSCW system is seen as an application of the distributed system infrastructure. Implemented at a different level, the CSCW system can be seen as *being itself a distributed system.* In this approach, the distributed system technologies can be directly incorporated into the collaborative application or environment.

8.2.1.1 Transparency in Distributed Systems

However, the observation that CSCW systems are distributed, and hence potentially amenable to distributed system solutions, can be a misleading one. Whether CSCW applications are implemented as distributed services or clients of those services, designers must take care not to confuse the goals of distribution with those of collaboration.

Many distributed systems set out to achieve some form of *transparency*. Typically, the goal

of transparency takes the form of attempting to hide from the user the consequence of some aspect of distribution, while still realizing the benefits. Consider some examples below:

- *Location transparency* refers to isolating the application or client from the effects introduced by the *location* of the computation.
- *Concurrency transparency* refers to isolating the application or client from the effects introduced by the fact that their computation might, in fact, consist of multiple concurrently-executing subprocedures which, together, can be regarded as a single computation.
- *Replication transparency* refers to attempts to hide the fact that what appears to be a single data item may, in fact, be copied and reproduced at different points in a network.
- *Failure transparency* refers to attempts to hide from applications the consequences of a potential failure at one point in the network, by attempting to recover using other resources available in the distributed system.

In different settings, different forms of transparency can be invaluable in providing users and applications with seamless access to an apparently unified large computational resource which is, in fact, made up of discrete, connected units. However, these same features can become problematic in the CSCW setting, since the goals of CSCW are different. For instance, issues such as location and replication, which might be hidden by a traditional distributed system, can often turn out to be significant for the ways in which a group will work, or even for the nature of the work which they attempt to perform. Greenberg and Marwood [Gre94] discuss the ways in which concurrency management, for example, can interfere with the smooth and natural flow of user interaction when a distributed systems layer makes concurrency control "transparent" to the CSCW application. They point out that the details which distributed systems hide (by making them transparent) are ones which are highly significant for the coordination of group tasks.

Distributed system techniques *are* important elements of CSCW infrastructure, and extremely valuable. Data replication allows fast, concurrent access in cases where it would otherwise be impossible, and location transparency allows users to interact in mobile or fluid settings. However, before these techniques are applied directly to collaborative systems, the designer must develop a more detailed understanding of potential interactions between the behavior of users and the action of the system. Certainly, collaborative activity is often distributed; but this does not imply that collaborative applications and distributed applications are one and the same.

8.2.2 CSCW and Databases

Many features of CSCW applications make database technology an attractive candidate for infrastructure. Most programs are data-based, of course, but in particular CSCW systems often involve sets of computations over an explicit data store (or collaborative workspace). Similarly, database technologies have evolved to provide the means to coordinate and share data across time and space. As such, many collaborative systems can benefit from techniques developed in database management, and the persistence which databases offer may be exploited in supporting asynchonous working styles.

Most database systems support multiple users, but mapping the needs of collaborating groups onto the multi-user facilities of an existing database technology can be problematic. Multi-user databases are generally constructed so that they hide the activities of multiple users. Database systems erect walls between simultaneous users, in order to render each user imper-

vious to the actions (or even the presence) of others. The goal is to present each user with the illusion of a dedicated system. This is not simply an issue in how their interfaces are constructed, but reaches down to the basic conceptual model. Even the transaction execution model, for example, is explicitly designed to shield users from the effects of each other's actions, and to maintain the idea of a dedicated resource for each user.

The activities of others, then, are hidden and may become visible only through activity within the data store itself; and that activity is organized as essentially single-user, so that database consistency constraints can be maintained. However, a wide range of research studies in CSCW (typically going under the general term "awareness") have emphasized the importance of the *visibility of others' work* as a resource for coordination. In Heath and Luff's seminal study of the activities in the control rooms of the London Underground, for instance, they uncover a range of practices by which the controllers not only monitor each other's actions in order to coordinate the work as a whole, but also ways that they explicitly *make* their work visible to each other [Hea92]. Dourish and Bellotti [Dou92] observe similar issues at work in experimental collaborative design tasks. This sort of mutual visibility of action is hard to achieve in traditional databases. So while the database model might *enable* cooperative work by allowing multi-user data access, it generally doesn't *support* a collaborative model of data management.

However, some database research work has focused on extending the database model in ways which extend to collaborative settings. Extended transaction models such as nested transactions (originally introduced by Davies [Dav73]) or multiple granularity concurrency control [Gra75] have been developed. These extended models were driven by the requirements of domains such as computer-aided design or software development environments, where transactions may last much longer, involve multiple participants, or be transferred from one participant to another before being committed. At the same time, new techniques for semantics-based concurrency control in database applications (such as those of Herlihy [Her90] or Farran and Ozsu [Far89]) allow for greater parallelism in transaction execution, and hence more flexibility in mapping collaborative actions onto a database kernel. In the same way, aspects of database infrastructure may have to be extended for collaborative settings. (A semantics-based technique, similar to those cited above but specifically designed for CSCW applications, will be described in Section 8.6.4.2.) Barghouti and Kaiser [Bar91] provide a comprehensive overview of these developments, which hold considerable promise for the future role of database technologies in CSCW.

8.2.3 CSCW and User Interfaces

CSCW systems are generally interactive, and so the design of the user interface is critical to their acceptibility and use. However, as in the domains discussed above, CSCW introduces new challenges for user interface design.

In a single-user system, the user interface is responsible for presenting representations of the system's activity. For instance, the "hourglass" cursor indicates that the system is currently performing some time-consuming operation in response to a user request; dialog boxes may appear, asking for confirmation for requested actions (especially ones with potentially severe consequences); user-initiated changes in system state are reflected in changes to the display state of user interface objects (e.g. reversing black and white to indicate object selection).

Although these same mechanisms can be exploited in collaborative systems, we must, once again, consider the implications of moving into a multi-user setting. There's an important

piece of context which allows these kinds of behaviors to make sense in traditional interactive systems; the fact that there's *only one user*. This is particularly important because it implies that there is a straightforward relationship between the user's request and the system's response. Objects do not highlight themselves, but do so because they have been selected; dialog boxes asking for action confirmations do not appear at random, but in response to specific user actions. By and large, the system need not explain why (for example) a dialog box has appeared, because the user knows that it is in response to their recent activity. If something happens in the interface, it must be as a result of either the user's action or the system's.

However, in collaborative systems, this assumption may no longer hold. There are now multiple users to be considered, and actions which are observable in the interface may well be the result of *someone else's* activity, which may or may not be visible to other users. The direct connection between the user's activity and the system's has been broken, and with it, many of the assumptions on which user-interface design rests. So, as in the previous cases, the needs of CSCW applications often force us to re-think the elements and functionality of the traditional user interface.

That said, there *have* been cases where elements of current user interface systems have been fruitfully exploited in collaborative systems. One particular line of work has been with network-based interface architectures such as the X Window System and NeWS. These systems separate window clients (programs which use the window system to display results) from window servers (which provide windowing functionality for particular screens or displays), potentially across a network, using a hardware-independent protocol for drawing and windowing actions. This network independence immediately leads to the potential for multiplexing the windowing protocol, and hence sharing a single client between a number of displays. A number of systems of this sort have been developed, of which the best-known is probably Shared X [Gar89]. Application replication via window sharing allows previously single-user applications to be operated in a multi-user environment, albeit with certain restrictions to manage input streams. This is an extremely powerful approach, especially since it allows users to carry on working with familiar, everyday applications.[1]

Other user interface toolkits, widgets and mechanisms have been extended to support collaborative working. This work has typically been done in groupware toolkits, which are discussed in Chapter 6 of this book [Gre99] and so will not be discussed further here.

We will now go on to look at some particular technologies which can be valuably exploited as infrastructure for CSCW systems. For clarity, they will be addressed in three different areas: communication; coordination; and storage.

8.3 COMMUNICATION

Most CSCW technologies depend critically on digital communication infrastructures. Indeed, there have been claims that the most successful CSCW products are those which we might think of as simply being communication systems (such as electronic mail, networked file services or the World Wide Web). This section will explore the communication facilities which underpin CSCW applications development, and recent advances in communication facilities which are particularly relevant to collaboration.

[1] The sad truth about many collaborative editors which have been developed by CSCW researchers is that, while they might well be *collaborative*, they are rarely very good *editors*. This is another reason to value application-sharing approaches.

8.3.1 Internet Multicast and the MBone

One infrastructure advance of the past few years which is particularly relevant for CSCW is the development and widespread deployment of Multicast Internet Protocols, and the emergence of the multicast backbone or "MBone", a virtual Internet backbone for the distribution of multicast data.

The original Internet Protocol (IP) [Pos81] is a unicast protocol. That is, it supports one-to-one communication; each packet identifies a single receiver, and IP routes it precisely to that host. Receivers are named by IP addresses, which identify particular hosts. (Actually, IP addresses identify particular network connections, so that "multi-homed" machines with multiple network connections will actually have multiple addresses, but the fiction that IP addresses name hosts will be convenient here.)

In his thesis work at Stanford, Steve Deering developed mechanisms for IP multicast which could be layered on top of the existing unicast internet architecture [Dee88]. In his model, a set of addresses are recognized as naming "multicast groups" rather than single hosts. Using a low-level protocol called the Internet Group Multicast Protocol (IGMP), hosts can add themselves to multicast groups, essentially declaring an interest in the data sent to that group.[2] Any packets sent to a group (by using the group address as the packet destination address) will be routed to all hosts which have added themselves to the group. The IP multicast implementation is responsible for finding efficient distribution patterns for multicast data, so that packets sent to multicast groups will traverse any particular network connection at most once.

Multicast IP is managed by extending the routing mechanism of the traditional IP mechanism. IP packets sent to unicast addresses are handled normally, but packets sent to the multicast addresses will be processed specially. However, existing IP routing software and hardware were developed without support for Deering's new multicast model. The solution to this bootstrap problem was to develop, along with the new multicast routing mechanism, a way for multicast-aware routers to communicate with each other over traditional unicast channels. This approach — called IP tunnelling — treats unicast connections (the "tunnels") as simple network links between multicast routers. The unicast channels that distribute multicast data between multicast routers form a virtual internet over the existing Internet infrastructure. This is the so-called MBone, and it allows experiments with internet-wide multicasting to proceed before support for multicast protocols has migrated into the standard internet routing hardware and software.

Deering's original work was based on a multicast routing mechanism called DVMRP (Distance Vector Multicast Reverse Path). More recently, new routing mechanisms, such as MOSPF (Multicast Open Shortest Path First) [Moy94] and CBT (Core Based Trees) [Bal93] have emerged as possible internet-wide routing mechanisms. However, the choice of routing protocol does not affect the basic multicast service model.

Multicast extends the one-to-one model of unicast routing to a many-to-many model. Any member of a group can send data to the group, and any data sent to the group will be distributed to all participants. A multicast group can be thought of as a "software bus" allowing arbitrary communication between all connected components (group members). Multicast IP, then, provides a natural model for group communication in CSCW applications, and a number of widely-used multicast applications — the so-called "MBone Tools" — are collaborative applications.

---

[2] IGMP occupies roughly the same place in the IP multicast stack as ICMP (the Internet Control Message Protocol) plays for unicast IP.

8.3.1.1 Audio and Video Communication: vat, rat, nv and vic

The best known MBone tools are those which support the most common MBone activity — videoconferencing. While videoconferencing is rarely classed as a collaborative technology in itself, the long tradition of research in media spaces and video-mediated interaction (e.g. [Bly93] and Chapter 3 in this book [Mac99]) mean that it could certainly be regarded as a CSCW infrastructure component in its own right; but more pertinently here, it illustrates the use of multicast mechanisms in supporting cooperative work.

Early MBone tools, *vat* and *nv*, support audio- and videoconferencing respectively using multicast protocols. Audio and video sessions are made available as multicast groups, so that any MBone-connected host can subscribe to the group and "tune in". Since multicast is a many-to-many (rather than one-to-many) distribution model, this allows any member of the group to send multimedia data to all others.

However, the current Internet is a harsh environment for reliably delivering real-time data such as audio and video. Different participants may be connected by different means, have different levels of bandwidth available to them, and different latencies; and activity elsewhere on the network can introduce congestion at different points in the network. Factors like these make it difficult to provide continuous, timely streams of multimedia data uniformally across a multicast group. To address these problems, many MBone tools support a model called *lightweight sessions* [Flo95].

In TCP, reliable delivery is the responsibility of the sender. However, this approach does not work in multicast situations for a variety of reasons. One of these is the scaling problem; in a sender-based approach, the sender would be responsible for the different timeouts and resends for hundreds or thousands of receivers. Another is the danger of "ACK implosion", as all the receivers acknowledge receipt of a packet. Instead, in the *lightweight sessions* approach, receivers are made responsible for managing reliable streams. In addition to the data components, "session messages" are used to maintain a view of session membership, as well as to provide other checkpointing mechanisms around which the data protocols can operate. This approach to managing multicast sessions applies not only to the audio and video tools, but also to artifact-based collaborative tools described in the next section.

In addition to the problem of reliability in multicast streams, there is also a need to ensure timely delivery of temporal streams such as audio and video. The network itself provides no support for timely delivery. Instead, in the lightweight sessions model, incoming data is buffered in the receiver, which then attempts to deliver it to the user in a timely manner. The "playback point", corresponding to buffering delay, is continually adapted to current network conditions; closer to packet arrival time in the case of good network connectivity and performance, and further from packet arrival time if network response is poor (thus allowing more time for misordered packets to arrive and fill holes in the buffer).

Two newer tools, *vic* [McC95] and *rat* [Har95], are improved tools for video and audio respectively, incorporating lessons gleaned from the widespread deployment and use of tools like *vat* and *nv* over the MBone since 1990. They reflect greater understandings of network-friendly approaches to compression and encoding, architectures for real-time streams management on the Internet, and the integration of user interface and network level concerns.

8.3.1.2 Collaboration Tools: wb and nte

The first widespread MBone tools, discussed above, were for audio- and videoconferencing. More recently, tools directly supporting artifact-based collaborative work have appeared.

Wb [Flo95] is a shared whiteboard application from Lawrence Berkeley Labs (where vat and vic were developed). *Wb* is commonly used not only for collaborative interaction, but also as a presentation medium for Internet-broadcast talks. It presents a collaborative whiteboard with multiple pages. Any multicast group member can create a page, and any can draw on any page. *Wb* has been designed with a concern for scalability which is somewhat unusual in real-time CSCW design, with the result that it can support hundreds of receivers distributed across the Internet in a single session.

One particularly interesting aspect of *wb*, which emphasizes the way in which it combines networking and CSCW technologies, is the mechanism used for late joining (allowing clients to join a session which is already in progress). In general, *wb* uses a retransmission request mechanism for *wb* clients (or trees of clients) to ask for lost packets to be delivered again. *Wb* uses this same retransmission request mechanism to allow clients which join sessions in progress to catch up with the session state. Essentially, a client which joins a session in progress can be thought of as a client which has not successfully received *any* packets in the session so far. So the standard retransmission request mechanism provides a way for late arrivals to be brought up-to-date.

Wb provides collaborative access to drawings, and while text can be added to pages, it does not provide a way to collaboratively edit that text. *Nte* [Han97] is a collaborative text editor which uses multicast to support group collaboration over the Internet and MBone. Like *wb*, *nte* employs the techniques of lightweight sessions and Application Layer Framing [Cla90] to provide a high degree of scalability. Reliability and resilience to transient network failures in the face of this scalability is achieved through a loose consistency model, and the exploitation of natural redundancy; *nte* uses text lines as its basic data unit, but most characters are entered on the same line as the previously-entered character, so successive data transmissions involve inherent redundancy, which reduces the need for retransmissions.

8.4 COORDINATION

Along with communication, simply getting the data from one point to another or a set of others, a critical concern for CSCW technologies is the *coordination* of distributed action. While communication and coordination are two sides of the same coin, in this section we look at approaches which focus more on the management of concerted action, rather than on data transfer.

8.4.1 Group Communication: ISIS and Horus

Isis is a group communication system developed at Cornell University (and subsequently at Isis Distributed Systems) [Bir94a]. Its design was originally aimed at the production of reliable, fault-tolerant systems. Isis provides a *process group* abstraction in which inter-process communication can be directed towards groups rather than individual processes, as in the internet multicast model described above.

The basis for group communication in Isis is a model called *virtual synchrony* [Bir87]. Message deliveries to the members of a group are virtually synchronous. In this approach, message delivery is controlled so that there are no observable differences in the message arrivals at process group members. The motivation behind this model is support for replication-based fault-tolerance in distributed applications. Critical services are replicated as members

of process groups rather than individual components. The system can continue to function even though the individual members of a process group may fail; *every* member of the group must fail before the group as a whole fails. Virtual synchrony ensures that all members of a process group see the same pattern of network activity; in turn, this ensures that their state is accurately replicated, so that they are each maintained in equivalence.

Although replication for fault-tolerance was the original motivation behind the development of group communication in Isis, it has been used by a number of researchers as the basis for the development of CSCW systems, including the DistEdit toolkit [Pra99] (see Chapter 5 in this book), the collaborative virtual reality system DIVE [Car93] and the COLA application platform [Tre95].

Horus [Ren96] is a more recent group communication system designed by the researchers who previously developed Isis. The primary research focus behind the development of Horus is flexibility through *micro-protocol configuration*. Rather than providing group communication mechanisms as a monolithic protocol, Horus allows programmers to compose a series of microprotocols which provide different functional elements, such as total ordering, reliable delivery, encryption and fragmentation and reassembly. In this way, the programmer can configure the protocol stack to the specific needs of any particular application, eliminating potentially costly features not needed in particular circumstances. These issues of configuration and customization will be addressed in more detail later in this chapter (Section 8.6).

8.4.2 Coordination Languages

One particularly interesting set of coordination technologies which can be exploited in developing CSCW applications is coordination languages. The earliest explicit coordination language is Linda [Gel85], originally developed at Yale in the mid-1980s. Linda comprises a set of programming language extensions which provide coordination facilities for distributed programming. Gelernter explicitly draws a distinction between the coordination language — provided by the Linda facilities — and the computation language — a standard programming language within which the Linda primitives are embedded. Early versions of Linda were embedded in a variety of languages, including C and Lisp.

A number of other languages have emerged for explicitly distributed programming, in which coordination mechanisms become programming language features, rather than library extensions for process communication, and so on. Obliq is a simple but powerful language of this sort, developed by Luca Cardelli at DEC's System Research Center. Obliq is of particular interest here, since it has been used as the basis of a graphical builder for collaborative applications, Visual Obliq [Bha94].

8.4.2.1 Linda

Linda was originally developed for parallel programming applications, although the loose coupling of components which it provides also makes it suitable for styles of programming more readily classed as "distributed" than as "parallel". Linda comprises a set of programming language extensions embedded in a traditional "computational" programming language in order to provide the coordination facilities needed for distributed programming. Linda's coordination model is explicitly designed independently of underlying connection models and topology, making it suitable for a wide range of parallel programming environments, from distributed processing on a LAN to tightly-coupled shared memory parallel computers.

The Linda model augments the base language with access to an associatively-matched shared tuple space. Any process can place data objects into the tuple space, and retrieve them by associative pattern-matching. Tuples are added to the space using the `out` primitive, which creates a tuple of its arguments and enters it into the space. Tuples can be retrieved using the `in` primitive. Arguments to `in` can be marked as *formals* — that is, variables which should be bound by the primitive, rather than used to specify patterns.

For example, consider the situation in which some process or processes have executed the following statements:

```
out(5, i, ``foo'');
out(6, i, ``bar'');
out(7, ``baz'');
```

These place three tuples into the tuple-space. The first two are 3-element tuples in which the second element has been initialized to the value of the variable `i` in the running process. Some other process can now execute the primitive `in(5, ?j, ``foo'')`. The question mark before the variable `j` marks it as a formal. The Linda system will then search the tuple-space for any 3-tuple with first element 5 and third element "foo". If there are multiple matches, then one will be selected at random and the variable `j` will be bound to its second element. If there are no matches, then the primitive will block until one becomes available.[3]

The blocking behavior of `in` can be used to coordinate the activity of different processes. A third Linda primitive, `in?`, is a non-blocking equivalent which returns true if there is currently some tuple in the tuple space which matches, and false if there is none (rather than blocking until it becomes available).

The fourth Linda primitive is `eval`. The argument to `eval` is a computational which, when complete, returns a tuple which will be added to the tuple space. The computation is spawned in parallel, and the original process continues immediately. For instance, in a "task farm" approach, a single process might spawn a whole set of computations using `eval` and then use `in` to wait for and collect the results.

Unlike the multicast mechanisms described earlier, Linda's basic (in/out) communication model transmits data to a single recipient (unless `in?` is used to read data without removing it from the tuple space). However, the senders need not name recipients; instead, data are simply placed in the tuple space and then retrieved by pattern matching. This feature makes Linda an interesting basis for CSCW implementation, since it abstracts away from details such as group membership, group naming and connectedness, as well as away from the topology and communication mechanism which supports the Linda model itself. Like the multicast model, Linda's abstract communication model supports a receiver-independent "software bus" architecture, distributed across multiple machines; but unlike multicast (or at least, current multicast applications such as *wb* and *nte*), it provides a framework for CSCW application programming which is independent of the underlying network service model.

8.4.2.2 Obliq

Obliq is not a coordination language in the same sense as Linda — that is, it is not a language dealing simply with coordination issues and which can then be embedded in an existing language for computation. Instead, it is a fully-functional object-oriented programming language

[3] In statically typed base languages, type information may also be used as input to the tuple matching process.

in its own right. However, it is a language specifically design for *distributed* object-oriented computation, and one which has been used as the basis not only for collaborative applications, but for a graphical builder for collaborative applications. As such, it merits attention here.

Obliq takes the basic object/message model of object-oriented programming and uses this as a means to distribute communication across a network. Objects in Obliq are implemented using the "Network Objects" mechanism of Modula-3 [Bir94b], and inter-object communication across a network becomes a natural expansion of the message-passing model of object-oriented programming.

Obliq objects have state, and in the presence of network communication this raises a set of potentially complex issues to do with the replication of objects and the consequent replication of state. Obliq deals with this through a *distributed scoping* mechanism. First, it makes objects static, and local to their own sites. Objects cannot move across network connections. Instead, object references are made available to be communicated across network links. In combination with other language facilities, such as aliasing and object cloning, this allows object migration facilities (for example) to be built up out of the state-safe primitives which Obliq provides. In general, then, it is not objects which move around the network, but *computations*. Computations run across the network either through invocations or through the transmission of procedures and closures. Since Obliq is lexically scoped, all free variables in closures are bound to references at their original site (using the network reference model).

8.4.2.3 Obliq as CSCW Infrastructure: Visual Obliq

One reason that it is particularly interesting to look at Obliq from the perspective of CSCW infrastructure is that it has been used as the basis for a research project on the development of CSCW technologies. The goal of the Visual Obliq project [Bha94] was to develop a direct manipulation graphical interface builder for collaborative applications which was no more complicated to use than familiar equivalent tools for single-user interfaces (such as NeXT's "Interface Builder", or Sun's "Guide").

To the application developer, the Visual Obliq interface builder looks like a traditional direct-manipulation interface builder. It provides a canvas, onto which the user can drag interface components, which can be laid out according to the needs of the particular application. Dialog boxes provide controls over the attributes of each component, so that aspects of their appearance or behavior can be changed. Interfaces can be tested from within the builder, or the builder can be used to generate code which implements the created design.

The interface designer can associate callback code, written in Obliq, corresponding to the actions of the various components (e.g. pressing a button, or selecting a menu item). However, in addition to the pure Obliq language (which, of course, already embodies a model of distributed programming), facilities are also provided which support collaborative activity. The basic Obliq mechanisms — in particular, distributed lexical scope and network object references — provide a rich but simple model of distributed processing which can be used to support data migration, remote object access and distributed state.

8.5 STORAGE

Given the phenomenal growth of the World Wide Web (WWW) over the past few years, the use of WWW as a basic infrastructure for CSCW development is clearly something to inves-

tigate. The combination of platform independence and Internet accessibility makes WWW technology a clear infrastructure candidate.

8.5.1 CSCW and WWW

A variety of systems have exploited WWW in different ways. At GMD, the BSCW (Basic Support for Cooperative Work) system [Ben95] uses WWW as a means to provide Internet-accessible shared workspaces supporting group work. Projects such as Freeflow [Dou96c] use WWW to provide platform-independent interfaces to network-based collaborative services such as workflow systems. Mushroom [Kin95] uses WWW to provide a virtual shared space for group interaction, while systems such as America On-Line's "Virtual Places" augment WWW with collaborative access over existing WWW-based document repositories.

The emergence and increasing interest in CSCW systems based on WWW technology raises a number of questions for the future development of WWW, which is undergoing considerable change. There are three components of WWW technology which are exploited in the development of CSCW systems.

1. *Shared document access*. The basic hypertext access model provided by HTTP (the HyperText Transmission Protocol for communication between WWW clients and servers) provides for access to distributed document repositories across the Internet. Unified access to a shared document repository can in turn support collaborative activities.
2. *User interface management*. HTML extends the basic document markup model with support for user interfaces constructed from basic widget components. It provides a platform-independent basis for user interface management.
3. *Unified access to services*. Through the CGI mechanism, which makes external programs accessible as WWW documents, WWW technology provides distributed access to network services to participants across the Internet, independent of platform and location.

These mechanisms, independently and collectively, provide significant support for the creation of collaborative applications and, perhaps even more significantly, for their deployment.

8.5.1.1 BSCW

BSCW (Basic Support for Cooperative Work) is a Web-based collaborative system [Ben95]. BSCW maintains workspaces accessible to multiple participants over the Internet. Documents can be stored in the workspace, making them available to other participants, and retrieved by others. The workspace is a coordination point for the multiple users, as well as providing a simple visualization of the document store.

BSCW provides an access control mechanism to maintain control over who can read and write documents in the workspace. It also uses a general event mechanism to maintain users' awareness of activities in the shared space. These mechanisms are all part of the BSCW server. The Web is used to provide a network-accessible user interface and visualization environment, as well as access to the document repository (workspace) itself.

8.5.2 SEPIA and CoVer

The World Wide Web is, of course, a distributed hypertext system. However, as suggested in the previous section, most uses of WWW as CSCW infrastructure have not focused on it

as a distributed hypertext system, but rather have exploited its facilities for shared access to documents and platform-independent interface functionality. A number of other projects have used hypertext more generally as a means to support collaborative working.

SEPIA [Str92, Haa92] is a collaborative authoring system developed at GMD which uses hypertext and hypermedia to support collaboration in various ways. The basic hypertext model provides a means to structure interactions. One component of SEPIA — the *argumentation space* — is a collaborative argumentation system, similar to models such as IBIS. Argumentation structures allow users to post issues (as hypertext nodes) and then annotate them with argumentation (backing, agreements, comments, disagreements, and supportive argumentation). The various relationships between pieces of argumentation (such as "supports" or "refutes") can be modeled as different forms of hypertext link. As the collaboration progresses, the argumentation structure emerges as a hypertext document. In this way, then, the basic hypertext model directly supports this form of collaboration.

Another component, the *rhetorical space*, exploits hypertext to represent and manipulate the structure of the document being produced. Document sections are unpacked as hypertext nodes, with the rhetorical organization of the document made visible as hypertext relationships. Again, the basic hypertext model provides a decomposition of the task, and so supports visualization of the collaborative process.

SEPIA uses a collaborative versioning system called CoVer [Haa93] which is also specialized to the need of collaboration. Activity over hypertext nodes causes new versions to be created, and CoVer maintains the relationships between new and old versions. The version mechanism provides a historical record of the actions of individuals and the evolution of the document. It also allows concurrent versions to be created in the presence of simultaneous work by multiple participants, as well as providing for their subsequent integration into a single, unified document.

8.6 INFRASTRUCTURE AND SPECIALIZATION

In the first part of this chapter, we have seen a number of elements of CSCW infrastructure, and technologies which can be used to provide infrastructure services to collaborative applications. In this second part, I want to take a different tack. Here, we will step back to consider the issue of infrastructure provision more generally.

The focus in this section will be on what it means to provide infrastructure services, and what is demanded of them by applications and application programmers. I will outline a set of systematic problems introduced by conventional approaches to system structure, and introduce a solution which has been developed and demonstrated by a prototype CSCW toolkit called Prospero.

8.6.1 Layered Models

A critical assumption underlying the discussion of CSCW and infrastructure in the discussion above concerns the separation of system components. At some point or other, we have discussed a large number of components — networking services, distributed object services, hypertext storage services, CSCW support, user interface and applications. We have relied, implicitly, upon a standard model of the relationship between these components in which the operating services assume the "lowest level", the applications the "highest", and other compo-

nents are ranged in between, organized in a "stack" separated into different "levels" each using facilities offered by components lower in the stack, and offering services to the components above.

This approach to structuring large software systerms is familiar, even commonplace — so much so, in fact, that it can remain implicit in discussions such as those above without causing confusion. Perhaps one of the best-known layered models of this sort is the seven-layer ISO Reference Model (ISORM) created as part of the Open Systems Interconnection standardization effort [Zim80]. The ISORM defines seven different levels of network processing (Physical, Data, Network, Transport, Session, Presentation, and Application) layered on top of each other and each depending on the services provided by the layers below. It is perhaps because this influential model was developed in the context of data networking that, while the layered approach is very common in all sorts of systems, it is particularly common in describing networked and other distributed systems, including CSCW systems.

8.6.2 Abstraction and Mapping Dilemmas

The development of models such as the ISORM described above arises directly from the notion of abstraction in software design. Abstraction is a basic tool which we use to manage system problems — to break them down into components, to compose them into larger systems, and to separate issues of concern for independent analysis and solution. Abstraction allows us to separate the details of an implementation from the means by which it will provide its functionality or set of services to other system components. It allows for a separation of (and hence an independence between) the *implementation* of a system and the *interface* it provides. Abstraction allows us to tackle large problems, to organize the work of large software teams, and to reuse software. Our concern here is on the place of abstraction in CSCW infrastructure.

A module, or system component, offers an abstraction at its interface, which sets the terms in which other system components can make use of its services. The responsibility of a component is to allow other components to talk in terms of that abstraction, while the implementation itself talks in other terms (perhaps those terms offered to it as a client of other system components). For example, a window system provides abstractions such as windows and scroll bars, while internally it deals with screen areas and pixels; a statistical package offers abstractions such as distributions and means, while internally it deals with data arrays and functions; and a programming language compiler offers abstractions such as function calls and arrays, while internally it deals with stack frames and memory blocks. The job of the implementation (or the job of the implementor) is to map these higher-level structures of the abstraction into the lower-level structures available at the implementation. Since there are frequently a range of ways in which some higher-level feature can be implemented, the implementor makes a set of *mapping decisions* from higher to lower level. For instance, in implementing a simple records system, an implementor might choose whether to store records as an array or a linked list. Decisions like these — normally quite simple — occur throughout an implementation. They are the work of programming.

However, these decisions — such as between arrays and linked lists — carry with them consequences for the use of the abstraction by clients. Linked lists favor particular sorts of access patterns at reduced storage cost, while arrays represent a different approach to the same tradeoffs. The programmer is, then, making a set of decisions which are informed by expectations of likely access patterns; that is, expectations of the need of clients of the abstraction.

The problems begin to emerge when multiple clients (different programs or system modules) wish to make use of the same abstraction and implementation. This is a common — indeed, desirable — state of affairs. We would hardly exert much effort developing a window system unless we expected it to be able to support more than one windowing application. However, consider the case where the two applications wish to make quite different use of the abstraction. One wishes fast access to any record, in unitary time; the other favors sequential access to large, sparse sets of records. This is a *mapping dilemma* — the implementor must make one decision or the other, but in doing so, favors one style of client over the other.

So the mapping decisions which the implementor makes can affect the performance and behavior of clients. What's more, these decisions are invisible to the clients. Locked away behind opaque abstraction barriers, mapping decisions cannot be seen by the client. This combination of opacity and mapping dilemmas leads to *mapping conflicts* — occasions on which the client code encounters problems because it presumed that a mapping decision has been made one way, while in fact it has been made another.

These problems are endemic to the way abstraction is used in system design, and occur in all areas of system development. Dealing with them is part of the daily experience of programming, and mechanisms to cope with them are familiar to any programmer. For example, the way in which some systems — such as databases and graphics systems — have to be written carefully so as not to cause excessive paging behavior in the virtual memory system is an example of the efforts which programmers have to exert in the face of mapping dilemmas. However, rather than developing new programming strategies to cope with these situations, the approach we will explore here takes a deeper look at the source of the problems and opportunities for avoiding the mapping dilemmas altogether.

8.6.3 Open Implementation and Reflection

The problems with abstraction encountered in the previous section have been the motivation for recent work in *Open Implementations* [Kic96]. An open implementation is one which reveals aspects of its internal design in a principled way, so that these aspects can be examined and controlled by clients of the abstraction. The clients can adjust their behavior according to the details of the implementation which lies below the abstraction or, more radically, can adjust the abstraction, tailoring it to their own particular needs.

One technique which has been particularly useful in open implementation is *Computational Reflection* [Smi84]. The reflective approach was originally developed in the area of programming language design, but it has much wider potential applications. The principle behind computational reflection is that a system can embody a representation of its own behavior which is "causally connected" to the behavior it describes. This causal connection defines a *two-way relationship* between the representation and the behavior. Changes in the system's behavior will result in changes in the representation (so that the representation always provides an accurate view of the system's behavior at any time); and, at the same time, any change made to the representation will result in a change to the system's behavior.

Early work with reflection took place in the domain of programming language design and implementation. A reflective programming language might give programs access to a runtime model of the language's execution model. Programs written in that language have access to, and control over, an operational model of the language's semantics, portable across implementations of that language. This can be used to extend language semantics (adding new language features, such as procedure parameter mechanisms), or to adjust implementation de-

cisions to suit the needs of the client (specializing internal language implementation features, such as data representation procedures). From the problems identified with abstraction in the previous section, the argument is that this access can be used to see and control the mapping decisions which have been made, and so avoid mapping dilemmas, where the needs of the client and the (hidden) details of the implementation are in opposition.

Open implementations provide not only an implementation of a core set of abstractions, but also an abstract view onto the inherent structure of the implementation. The interface to the core abstraction is called the "base level interface" (or just the base interface), while access to the abstract view of the implementation is provided through the "metalevel interface" (or just meta-interface). The meta-interface provides the means to view and control the way in which mapping decisions are made, so that applications can customize how the abstractions which the system offers are provided. The separation of base and meta-interfaces results in a clean separation between base code (which uses the base interface and implements the system) and the meta-code (which uses the meta-interface to customize the implementation). This separation results in more easily maintainable systems.

8.6.3.1 Reflection in CLOS

Let's consider a more detailed example. One of the best-developed and most widespread reflective systems is the Common Lisp Object System (CLOS). CLOS is an object system for Common Lisp, which is directly incorporated into the language (and which is now included in the ANSI language specification). CLOS programmers can write object-oriented programs using familiar object-oriented mechanisms such as classes, objects and methods (as well as a few less familiar ones, such as multi-methods and method combination). These basic components of the programming language constitute CLOS's base level.

CLOS also offers a metalevel, which allows the internal details of the programming language and its implementation to be tailored to the needs of specific applications. The CLOS implementation offers a view of its own internal mechanisms — for instance, the creation of new instances, or the search for method code when a generic function is invoked.[4] This model of internal action is structured as a CLOS progam; essentially, CLOS is defined as if it, itself, were a CLOS program. Representations of the internal structures of CLOS, such as classes and methods themselves, are presented as CLOS objects. So, any particular class is available in CLOS as an instance of the predefined class `standard-class`. Newly defined classes are, by default, instances of `standard-class` (that is, `standard-class` is their *metaclass*); and operations over classes (such as finding their superclasses, allocating instances or adding methods) are represented as methods on `standard-class`.

This metalevel arrangement allows CLOS programmers to "reach into" the implementation and change aspects of it to suit their own needs. Since `standard-class` is a normal CLOS class, it can be subclassed like any other. New methods defined on the subclass will override those already defined. Since the methods defined on `standard-class` are the internal behaviors of the object system, those internal behaviors will be replaced for any class whose metaclass is the new programmer-supplied metaclass, rather than `standard-class`. The programmer has changed how aspects of the language behave.

This mechanism can be used for a wide range of purposes:

1. The reflective mechanism can be used to make *efficiency* improvements for particular cases.

[4] A generic function occupies the place in CLOS of a virtual function in C++ or a message in Smalltalk.

For instance, a programmer might wish to make changes to the way the language implements instance allocation and slot lookup, perhaps to support "sparse" objects which define many slots (instance variables) but only use a small number.

2. The reflective mechanism can be used to effect *compatibility* changes, such as how the conflict resolution mechanism works for multiple inheritance. This can be used so that legacy code from a different object system can still be supported.
3. The reflective mechanism can be used to *extend* the base language's functionality. For instance, we might wish to provide a constraint mechanism which looks to the programmer like normal slot lookup.

The reflective approach allows these sorts of modifications to be done *within* the scope of the language, rather than being performed on a particular implementation, which would be inherently non-portable.

It is important to note that what CLOS offers at the metalevel is a *representation* of its internals, in terms of a CLOS program. In other words, there is a level of interpretation between the representation at the metalevel and the details of the actual implementation which lie below. After all, the structure of the CLOS metalevel is part of the definition of CLOS, and must be portable across different implementations. The details and performance optimizations of specific implementations, such as the uses of partial evaluation in the PCL implementation [Kic90], play no part in the metalevel representation. So while *aspects* of the implementation — or views of specific mapping decisions — are offered at the metalevel, this is at least one step removed from the details of the implementation code itself. The essence of open implementation design is to give *principled* access to aspects of the implementation; access that is organized around the metalevel designer's expectations of future needs.

Open implementation techniques developed largely in the domain of programming languages, although recently they have been applied to other systems, including window systems [Rao91], distributed systems [Oka94] and databases [Bar96]. My own recent work has focused on the use of these same principles and techniques in the CSCW context, leading to the development of a reflective CSCW toolkit called Prospero.

8.6.4 Prospero: Open Implementation and CSCW

The problems described above, problems of opaque interfaces, abstractions and mapping conflicts, are endemic to the way we use abstraction in systems design. As a result, they occur in all the various domains to which system design principles are applied. In CSCW, we can see a number of manifestations.

For instance, consider the problem of data replication. Toolkits for building collaborative applications will often provide a "shared data object" abstraction, which allows different clients to process and manipulate data, with the effects being propagated across a network to other interfaces. This is clearly an extremely valuable abstraction for collaborative applications, and one which we would certainly wish to exploit and build upon. However, we have to consider what implementation decisions are being masked by the shared data abstraction.

One set of decisions focus on data replication. Is the user data object to be replicated, so that copies of it exist at each likely access site, or is there one central copy on which actions are performed? If there is a single copy, where is it located? If there are multiple copies, how are conflicts managed? The goal of the abstraction is to hide exactly these sorts of decisions — ones which are unnecessary for the maintenance of the abstraction itself. However, these decisions are critical when it comes to *using* the abstraction. Data replication and conflict

management decisions have significant implications for the ways in which the abstraction can be used to support collaboration. For instance, if there is a single copy of the data item, then the access latency for widely distributed users may increase beyond the level necessary for fast interactive response. On the other hand, if there are multiple copies, then conflict management and resolution strategies may begin to have effects which are reflected at the interface. Users may have to obtain locks on data, for instance, and there may be pauses while these are obtained; or actions may be subsequently "undone" in order to maintain overall consistency. (These issues are discussed in detail by Greenberg and Marwood [Gre94] and in Chapter 5 of this book [Pra99].)

Prospero is a prototype toolkit for collaborative applications which uses open implementation to give the application developer control over how the toolkit will provide its support [Dou95a, Dou96a]. In particular, Prospero provides mechanisms for data distribution and concurrency control which not only support particular styles of CSCW application, but also allows application programmers to reach into the toolkit and customize those mechanisms to the needs of specific applications.

8.6.4.1 Data Distribution and Divergence

Traditional approaches to data distribution in CSCW are concerned with issues such as centralization versus replication, or supporting synchronous versus asynchronous working. However, as discussed earlier, distinctions like these begin to affect the ways in which applications can be built on top of toolkits, and in which those applications can be used in collaborative working.

The standard approach is to manage access over potentially distributed data by mapping the activities of multiple users onto a single stream of activity. Techniques such as dividing access across asynchronous sessions, establishing total orderings over simultaneous distributed activities, or serializing access at a single central data store, are all ways of mapping the activities of multiple users into a single, unified stream.

The establishment of a single stream out of multiple, potentially simultaneous sources of activity is the focus for a number of mapping decisions critical of significance to collaborative activity. The distribution mechanism which Prospero offers is explicitly based on multiple streams of activity, around which it manages distributed data and distributed action in terms of *divergence* and *synchronization* [Dou95b].

Actions which arise in the course of collaboration — creating objects, editing them, changing attributes, or whatever — are each associated with some particular stream. Streams normally correspond to different individuals in each collaborative session, although this is not a requirement of the model. Streams might represent session recorders, for instance, or be proxies for remote groups, etc. When an action is added to a stream, the effect is to cause a *divergence* between that stream's view of the data store and the views of other streams, since those streams have not yet seen the action take place. Periodically, streams are *synchronized* to re-establish a shared view of the data store.

The model is defined independently of any particular period of synchronization, so that the period can be varied in different applications. With a small period of synchronization, streams will be synchronized frequently, after only small changes have been made. For instance, when the period is fractions of a second, then the effect will be similar to that of traditional "synchronous" applications, in which the activities of one user are reflected quickly in the views or interfaces of others. However, when the period of synchronization is large, perhaps of the

order of hours or days, then the effect is similar to that of traditional "asynchronous" applications, in which individuals work separately, coordinating their work and exchanging changes less frequently.

Divergence and synchronization are made explicit in this model so as to open them up for examination and change within the toolkit. Application programmers can gain control over the means for adding actions to streams, and for establishing divergence. Similarly, the programmer can gain control over the conditions under which synchronization takes place, as well as the extent of synchronization required.

8.6.4.2 Consistency Guarantees

One traditional way of managing exclusion and hence maintaining data consistency in the face of parallel user activity is the use of *locks*. Prospero extends the basic locking approach with a new abstraction called *consistency guarantees* [Dou96b]. Consistency guarantees provide a more flexible approach to managing data consistency, as well as supporting customization by application programmers to define new models of consistency management specialized to the semantics of individual applications.

The basis of the traditional locking mechanism is that the server (or lock-granting authority) gives a guarantee of data consistency (the lock) in exchange for a characterization that the client provides of upcoming activity (commonly, a description of the area over which the lock should operate). The lock can be regarded as a guarantee of future consistency for two reasons: first, because inconsistency could arise due to simultaneous activity if the locking mechanism was not used; and second, because the server will grant the lock to only one client, ensuring serial access.

The consistency guarantees mechanism which Prospero provides generalizes the locking mechanism in two ways. First, clients can provide richer descriptions of upcoming activity. These are called *promises*, specified in terms of the semantics of operations. Clients create promises from sets of semantic properties (idempotency, monotonicity, destructiveness, etc). These promises contain more useful information than the traditional read/write distinctions, which allow the server to make more informed decisions.

The second generalization is in the form of the locks. Rather than returning normal locks, Prospero servers return guarantees of achievable consistency when synchronization occurs. (Although this discussion is framed in terms of client/server for familiarity, Prospero uses a peer-to-peer model.)

A traditional lock guarantees absolute consistency. Prospero consistency guarantees, on the other hand, may offer more limited forms of consistency (such as "syntactic consistency", in which multiple possible values for data items are collected together so that all participants share a common view, although more work must be done later to resolve the situation).

Although the consistency guarantees approach loosens various restrictions of traditional locking, there is still a significant problem with the promise/guarantee model. Because promises must be given before action, there is a need to predict what user action will take place, and then to restrict action to precisely what was promised. Especially in asynchronous (or, rather, infrequently synchronized) working, this restriction can prove a serious limitation to the styles of work which users can perform. To avoid this, Prospero allows clients to break their promises. If a user "breaks a promise" — that is, engages in activity other than that which was promised — then the guarantee no longer holds, although the system may still attempt to incorporate the changes made. Particular client applications may or may not offer this facility

to their users; they may insist upon keeping to plan, or they may choose to warn users when a promise may be broken. The framework as a whole, however, is designed to deal with these sorts of situations.

8.6.4.3 Configuring Infrastructure in Prospero

Like other open implementations, there are two aspects to Prospero. The first is the default or base level behavior — the basic mechanisms which programmers can use to develop applications. Programmers can use Prospero to develop CSCW applications in which user actions are associated with streams of activity which are periodically synchronized with each other. The default stream type, `bounded-stream`, allows a certain number of actions to be accumulated before it automatically forces synchronization with peer streams in the system. Concurrency control is optimistic by default.

The second aspect is the metalevel control which the open implementation provides. In Prospero, the relationships between actions, streams, divergence and synchronization mechanisms is made available through the provision of the system's meta-objects and the generic functions which relate them within a programming structure. So programmers can reach in and modify the ways in which divergence is observed, or the triggers to synchronization, or the nature of synchronization which will be performed. Similarly, the consistency guarantees mechanism provides a programmatic way for application developers to express semantic features of their programs, so that these can be incorporated into the consistency management mechanism, effectively specializing internal toolkit behaviors to the characteristics and requirements of particular applications.

Just as in the CLOS example provided earlier, this metalevel programming takes place largely through the subclassing and specialization of the metaobject classes which the toolkit reveals. This allows programmers to precisely direct their adjustments, in two ways. First, it reduces the amount of metalevel programming they need to perform; most behaviors can simply be inherited, rather than rewritten, and only the new behaviors must be described. Second, it narrows their focus to the particular areas of the system requiring modification; the generic dispatch mechanism of object-oriented programming allows multiple behaviors to exist side-by-side.

These mechanisms have supported the development of widely different applications in Prospero; synchronous and asynchronous, graphical and data-based, with centralized and replicated data, and loose and strict consistency policies. These applications demonstrate the way in which Prospero's open implementation design allows application programmers to avoid the mapping conflicts which emerge in traditional designs, and take control of the infrastructure which supports them.

8.6.4.4 Example: The Bibliographic Database

Let's consider a brief example to illustrate how Prospero is used to create collaborative applications and, at the same time, illustrate the new role of infrastructure under a reflective approach. Longer and more detailed examples are provided in [Dou96a].

Consider creating a shared application for managing bibliographical entries. You might read the store of references to browse them, look up specific entries, or to generate a set of formatted references from the citations in a document. You might update the store to correct an error in an existing entry or to add new publications as they become available. The application

is shared amongst a number of users, perhaps the members of a research group who share a set of common interests (and, therefore, are likely to refer to the same set of publications).

The first step is to organize the actions around streams. Updates, changes, lookups and retrievals are separate operations which are captured in streams of activity associated with each user. The critical issue is the set of circumstances under which streams will be synchronized. Prospero offers a number of pre-defined streams with different synchronization characteristics; `bounded-stream` is a stream which will synchronize with its peers whenever a certain number of operations have been performed on it, or an `explicit-synch-stream` will accumulate actions until one particular synchronization action occurs. Alternatively, at the metalevel, a new stream class can be constructed with specialized behaviors for any given setting. However, in this case, let's take `explicit-synch-stream`.

To encode the application's behavior in Prospero, the programmer creates new application action classes which corresspond to the different sorts of activities in which clients can engage (`lookup`, `new-record`, `change-record`). Objects corressponding to each application action are generated as the actions are performed, and are added to the stream. Prospero handles the synchronization between streams.

Prospero's behavior can be further specialized by using semantic properties of the application actions to increase parallelism. As described earlier, the idea here is that we can use the detailed semantics of the application domain as the basis for consistency management, rather than simply using the generic "read" and "write" of the database infrastructure. In this example, the major opportunity is in the two conditions in which data might be written — correcting a record or adding a new record. Three observations are critical:

1. Updates are far more common than corrections.
2. Updates do not conflict with lookups.
3. Two parallel updates are unlikely to conflict. Even if they are for the same publication, then they should contain the same information, and so either one can be executed and the other discarded.

These observations allow us to encode application semantics in the consistency management mechanism. First, we adjust the definitions of the application actions defined above, so that they are now defined in terms of a set of *application semantic properties* — in this case, whether or not they introduce potentially conflicting changes into the data store. This is only true of corrections, so only the `correct-record` action class will inherit from the property class `changes-data`.

Now that actions are specified in terms of semantic properties, the consistency management mechanism is updated in terms of these properties. The programmer can choose how to make use of the properties and what sorts of consistency guarantees to use. This is specified by providing methods for the compatibility testing methods which compare specific operations and return an indication of compatibility. In some cases, this might involve consulting recent execution history, or combining a set of compatibility predicates over a number of operations. In this case, however, we can solve the problem with only two operations — one method which says that any two generic application actions are compatible, and a second overriding method which says that no action is compatible with one which inherits the property `changes-data`.

As with the streams mechanism, once the action of the application has been specified in these terms, the Prospero mechanisms will handle synchronization and consistency management independently. However, in much the same way as these mechanisms have provided the

means for the programmer to specialize the toolkit's mechanisms for particular settings, so these automatic mechanisms may themselves be further appropriated and specialized. This example, however, does not require further specialization.

8.6.5 Reconsidering Infrastructure

The reflective approach opens up a new view of infrastructure. Instead of having to map the functionality required of an application into the generic facilities which the infrastructure provides, this approach instead allows programmers to specialize the infrastructure components so that they match the needs of particular settings.

This radically changes the nature of infrastructure, which takes a much more active role in the applications we might develop. Further, the relationship between application and infrastructure is changed, since the infrastructure no longer stands alone, unchanging, against the backdrop of different uses. Instead, it provides a framework within which each application can gain access to resources, but deploy them differently, reflecting the different needs, requirements and expectations for different applications or domains.

Prospero is a demonstration and exploration of these ideas as applied to CSCW. As was explored in the first half of this chapter, collaborative applications and settings can require significant flexibility in the underlying infrastructure. Prospero shows how the reflective/open implementation approach can recast this relationship and so provide a means to creating much more flexible levels of infrastructure.

8.7 SUMMARY

Since the design and implementation of CSCW applications draws on a number of areas of system design, such as data communication, distributed systems and user interfaces, there are a range of technologies and techniques which can be employed as infrastructure for CSCW systems design. This chapter has provided an overview of some of these areas, as well as discussing particular components which have been, or can be, used as infrastructure supporting CSCW systems development.

However, there are some important considerations to be borne in mind when evaluating infrastructures for CSCW systems. Experience has demonstrated that the needs and goals of CSCW design are often at odds with the design goals of these infrastructural components, and in particular, the way in which infrastructure services are implemented and combined can systematically introduce problems for the design and use of CSCW systems. For example, the management of distributed or replicated data, and subsequently the mechanisms which are used to maintain consistency in the face of potentially simultaneous action by multiple individuals, can interfere with patterns of collaborative activity. To support the rich forms of interaction which we observe in studies of cooperative work, applications need to be able to configure the way in which infrastructure services are offered to them.

In the final section of this chapter, I outlined an approach to this problem. The solution uses an architectural technique called Open Implementation, which provides clients of an abstraction with a principled form of access to a model of internal operations. The clients can use this mechanism to examine the way in which internal mapping decisions have been made, and to adjust those to suit particular application requirements. This approach has been exploited

in Prospero, a prototype toolkit for CSCW applications, based on the open implementation approach.

CSCW is a young and rapidly expanding field; and at the same time, many of the infrastructures on which we base our technologies are changing even faster. As we learn more about how these infrastructures can be deployed, and more about how CSCW applications are designed and used, then we can expect to see not only new opportunities for infrastructure support, but also new models of the integration and mutual adaptation of infrastructure and CSCW applications programming.

ACKNOWLEDGEMENTS

Jim Holmes provided useful feedback on an earlier draft of this chapter, and Jon Crowcroft and Mark Handley useful pointers for the section on internet multicast. Prospero was developed while I was working at the Rank Xerox Research Centre, Cambridge Laboratory (formerly EuroPARC) and at University College, London.

REFERENCES

[Bal93] Ballardie, A., Francis, P. and Crowcroft, J., Core Based Trees (CBT): A scalable inter-domain multicast routing architecture. *Proc. ACM Symposium on Computer Communications SIGCOMM'93*, San Francisco, California. ACM, New York, 1993.

[Bar96] Barga, R. and Pu, C., Reflection on a legacy transaction processing monitor. *Proc. Reflection'96*, San Francisco, California, 1996.

[Bar91] Barghouti, N. and Kaiser, G., Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, 1991.

[Ben95] Bentley, R., Horstman, T., Sikkel, K. and Trevor, J., Supporting collaborative information sharing with the World-Wide Web: The BSCW Shared Workspace System. *Proc. Fourth International World Wide Web Conference*, Boston, Mass. O'Reilly and Associates, Cambridge, Mass., 1995.

[Bha94] Bharat, K. and Brown, M., Building distributed, multi-user applications by direct manipulation. *Proc. ACM Symposium on User Interface Software and Technology UIST'94*. ACM, New York, 1994.

[Bir87] Birman, K. and Joseph, T., Exploiting Virtual Synchrony in Distributed Systems. *ACM Operating Systems Review*, 22(1):123–138, 1987.

[Bir94a] Birman, K. and van Raneese, R., *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, California, 1994.

[Bir94b] Birrell, A., Nelson, G., Owicki, S. and Wobber, E., *Network Objects.* Systems Research Center Research Report 115, Digital Equipment Corporation, Palo Alto, California, 1994.

[Bly93] Bly, S., Harrison, S. and Irwin, S., Media spaces: Bringing people together in a video, audio and computing environment. *Communications of the ACM*, 36(1), 1993.

[Car95] Cardelli, L., A language with distributed scope. *Proc. ACM Symposium on Principles of Programming Languages*. ACM, New York, 1995.

[Car93] Carlsson, C. and Hagsand, O., DIVE: A platform for multi-user virtual environments. *Computer Graphics*, 17(6):663–669, 1993.

[Cla90] Clark, D. and Tennenhouse, D., Architectural considerations for a new generation of protocols. *ACM Communications Review*, 20(4):200–208, 1990.

[Dav73] Davies, C., Recovery semantics for a DB/DC system. *Proc. ACM National Conference*. ACM, New York, 1973.

[Dee88] Deering, S., Multicast routing in internetworks and extended LANs. *Proc. ACM Symposium on Computer Networks SIGCOMM'88*. ACM, New York, 1988.

[Dou92] Dourish, P. and Bellotti, V., Awareness and coordination in shared workspaces. *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'92*, Toronto, Canada. ACM, New York, 1992.

[Dou95a] Dourish, P., Developing a reflective model of collaborative systems. *ACM Transactions on Computer–Human Interaction*, 2(1):40–65, 1995.

[Dou95b] Dourish, P., The parting of the ways: Divergence, data management and collaborative work. *Proc. European Conference on Computer-Supported Cooperative Work ECSCW'95*, Stockholm, Sweden. Kluwer, Dordrecht, 1995.

[Dou96a] Dourish, P., *Open Implementation and Flexibility in CSCW Toolkits*. Ph.D. dissertation, Department of Computer Science, University College, London, UK, 1996.

[Dou96b] Dourish, P., Consistency guarantees: Exploiting application semantics for consistency management in CSCW toolkits. *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'96*, Cambridge, Mass. ACM, New York, 1996.

[Dou96c] Dourish, P., Holmes, J., Maclean, A., Marqvardsen, P. and Zbyslaw, A., Freeflow: Mediating between representation and action in workflow systems. *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'96*, Cambridge, Mass. ACM, New York, 1996.

[Far89] Farran, A. and Ozsu, M.T., Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, 1989.

[Flo95] Floyd, S., Jacobson, V., McCanne, S., Lui, C-H. and Zhang, L., A reliable multicast framework for light-weight sessions and application level framing. *Proc. ACM Symposium on Computer Communications SIGCOMM'95*, Boston, Mass. ACM, New York, 1995.

[Gar89] Garfinkel, D., Gust, P., Lemon, M. and Lowder, S., *The SharedX Multi-User Interface User's Guide, Version 2.0.* Software Technology Lab Report STL-TM-89-07, Hewlett-Packard Laboratories, Palo Alto, California, 1989.

[Gel85] Gelernter, D., Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1), 1985.

[Gra75] Gray, J., Lorie, R. and Putzolu, G., *Granularity of Locks and Degrees of Consistency in a Shared Database.* Research Report RJ1665, IBM, San Jose, California, 1975.

[Gre94] Greenberg, S. and Marwood, D., Real-time groupware as a distributed system: Concurrency control and its effect on the interface. *Proc. ACM Confeerence on Computer-Supported Cooperative Work CSCW'94*, Chapel Hill, North Carolina. ACM, New York, 1994.

[Gre99] Greenberg, S. and Roseman, M., Groupware toolkits for synchronous work. In Beaudouin-Lafon, M. (Ed.), *Computer Supported Cooperative Work*, Trends in Software Series 7:135–168. John Wiley & Sons, Chichester, 1999.

[Haa93] Haake, A. and Haake, J., Take CoVer: exploiting version support in cooperative systems. *Proc. InterCHI'93*, Amsterdam, Netherlands. ACM, New York, 1993.

[Haa92] Haake, J. and Wilson, B., Supporting collaborative writing of hyperdocuments in SEPIA. *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'92*, Toronto, Canada. ACM, New York, 1992.

[Han97] Handley, M. and Crowcroft, J., Network Text Editor (NTE): A scalable shared text editor for the MBone. *Proc. ACM Symposium on Computer Communications SIGCOMM'97*, Cannes, France. ACM, New York, 1997.

[Har95] Hardman, V., Sasse, A., Handley, M. and Watson, A., Reliable audio for use over the internet. *Proc. INET'95*, Hawaii. 1995.

[Hea92] Heath, C. and Luff, P., Collaboration and control: Crisis management and multimedia technology in london underground control rooms. *Computer Supported Cooperative Work*, 1(1), 69–94, 1992.

[Her90] Herlihy, M., Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1):96–124, 1990.

[Kic90] Kiczales, G. and Rodriguez, L., Efficient method dispatch in PCL. *ACM Symposium on Lisp and Functional Programming LFP'90*, Nice, France. ACM, New York, 1990.

[Kic96] Kiczales, G., Beyond the black box: Open implementation. *IEEE Software*, 8–11, January 1996.

[Kin95] Kindberg, T., Mushroom: a framework for collaboration and interaction across the Internet. *Proc. ERCIM Workshop on CSCW and the Web*, Sankt Augustin, Germany, 1995.

[Mac99] Mackay, W.E., Media spaces: Environments for informal multimedia interaction In Beaudouin-Lafon, M. (Ed.), *Computer Supported Cooperative Work*, Trends in Software Series 7:55–82. John Wiley & Sons, Chichester, 1999.

[McC95] McCanne, S. and Jacobson, V., Vic: A flexible framework for packet video. *Proc. ACM Multimedia'95*, San Francisco, California. ACM, New York, 1995.

[Moy94] Moy, J., *Multicast Extensions to OSPF.* RFC 1584, SRI Network Information Center, Menlo Park, California, 1994.

[Oka94] Okamura, H. and Ishikawa, Y., Object location control using meta-level programming. *Proc. European Conference on Object-Oriented Programming ECOOP'94*, Bologna, Italy. Springer-Verlag, Heidelberg, 1994.

[Pos81] Postel, J., *Internet Protocol.* RFC 791, SRI Network Information Center, Menlo Park, California, 1981.

[Pra99] Prakash, A., Group editors. In Beaudouin-Lafon, M. (Ed.), *Computer Supported Cooperative Work*, Trends in Software Series 7:103–133. John Wiley & Sons, Chichester, 1999.

[Rao91] Rao, R., Implementational reflection in Silica. *Proc. European Conference on Object-Oriented Programming ECOOP'91*, Geneva, Switzerland. Springer-Verlag, Heidelberg, 1991.

[Ren96] van Reneese, R., Birman, K. and Maffeis, S., Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[Smi84] Smith, B.C., Reflection and semantics in Lisp. *Proc. ACM Symposium on Principles of Programming Languages*, Salt Lake City, Utah. ACM, New York, 1984.

[Str92] Streitz, N., Haake, J., Hanneman, J., Lemke, A., Shutt, W. and Thuring, M., SEPIA: A cooperative hypermedia authoring environment. *Proc. ACM Conference on Hypertext*, Milano, Italy. ACM, New York, 1992.

[Tre95] Trevor, J., Rodden, T. and Blair, G., COLA: A lightweight platform for CSCW. *Computer-Supported Cooperative Work*, 3:197–224, 1995.

[Zim80] Zimmerman, H., OSI Reference Model — The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications* 28(4):425–432, 1980.