Cooperation without (reliable) Communication: Interfaces for Mobile Applications

Alan Dix

alan@hcibook.com

http://www.hcibook.com/alan/topics/mobile/

appears in: *Distributed Systems Engineering*, **2**(3): pp. 171-181.

Abstract

The design of effective collaborative applications on mobile platforms has many problems. Wireless networks are fast enough that one aspires to true interactive applications, but too slow for much real-time feedback. This is exacerbated by the variability due to interference and broken connections. Mobile computing using wireless communications thus sits in a grey area between different styles of interaction. This paper examines these problems using concepts developed to understand general user interaction and computer-supported cooperative work. In particular, it will focus on the conflict between feedback for each user, awareness and feedthrough of the effect of each users actions to others, and the consistency of shared information during periods of disconnection.

1. Introduction

Both mobile communications and portable computers are now commonplace. The ubiquitous mobile phone has made it technically feasible, albeit not necessarily desirable, to be permanently available to your colleagues whatever your location. However, in contrast, the typical mobile computer stands in glorious isolation. Except for periodic direct or modem connections, the mobile worker is electronically alone. But this too is rapidly changing and the use of analogue or digital wireless telephones for data connectivity is becoming more reliable and less difficult. Sending and receiving faxes, file transfer and email are relatively painless.

Away from the mobile arena, the movement in many desktop applications is towards greater connectivity for both data access and intra-personal communications. The personal computer moved through a similar stage of isolation where the emphasis was on individual access to individual data, and transfer between computers was explicit and often painful. This is now considered totally unacceptable and access to corporate and workgroup data is the norm. This is perhaps most obvious in the commercial enthusiasm over client–server computing (in the popular sense). In addition, the importance of intra-personal communication and cooperation has been emphasised within the CSCW (computer-supported cooperative work) research community and the efficient use of groupware is seen by many as crucial for commercial success.

Users will understandably demand the same level of service from their mobile computers. However, one cannot simply assume that applications designed to run over local or even wide area networks will transfer to the wireless domain. Compared with fixed networks, wireless channels typically have lower capacity, are prone to noise, temporary loss of connection and explicit disconnection to save phone bills! The goal of many distributed systems platforms is to hide the application and hence the end-user from the detailed behaviour of the underlying channels. But the nature of wireless channels means that the user will inevitably be aware of delays and other problems. A style of interface which is designed assuming LAN connectivity may still 'work' in the sense that it does not crash or corrupt data, but may be utterly unusable.

This paper will discuss some concepts from the study of single-user and cooperative interfaces which can be used to make sense of the esoteric behaviour of mobile interfaces. One of the key problems we will find is that mobile communications are in a 'no man's land' between totally different styles of interaction. Designing effective interactive systems demands both careful user interface design and appropriate technical solutions.

We will begin by looking at issues for the individual user. In particular we will look at feedback and the availability of information. We will see how using the concept of pace we can analyse the tasks the user is

expected to perform and compare these with the feedback which is possible given the placement of data and functionality. This will also lead on to a discussion of the user's role in dynamically determining which data is replicated locally and which accessed centrally.

The second part of the paper shifts the focus to the cooperative issues, awareness of one another and feedthrough of the effects of one another's actions on shared data. We will see that mobile applications fall into an odd middle ground between synchronous groupware (such as video-conferencing) and asynchronous groupware (such as email). Furthermore, the shift between these modes will often be because of the state of communications links rather than at the choice of the user. The awareness of the state of communications is thus also a significant issue.

Finally, we will find that the requirements of the above two areas are to some extent in conflict. High availability in order to achieve sufficient feedback requires replication of data. However, periods of synchronous cooperative activity will increase the likelihood that several users will be accessing and updating the same information. Inconsistency during unplanned periods of disconnection becomes almost inevitable. The paper therefore finishes with a discussion of algorithms and user interface issues for the resynchronisation of data after disconnection.

The first two parts can be thought of as 'liveness' issues – the availability of data when it is needed, feedback of ones own actions and the awareness of other users' presence and actions. The end of the paper addresses 'correctness' the consistency of data after concurrent activity. That is the user interface problems often mirror those which face the designer of other network services such as low-level protocols and distributed databases. For example, we will be discussing caching and replication of data. However, the solutions adopted within a computer system cannot be transferred without modification when humans are involved – it is reasonable to ask a computer agent to rollback to a previous state, but not a user.

2. The individual: Pace and Feedback

First we consider issues for the individual using an application on a mobile platform. A key problem is where to place information and processing: locally on the portable computer itself, or centrally accessed over the wireless network. On the one hand we could simply replicate all data onto local machines, then use normal desktop applications on that data. periodically some form of file-transfer could be used to download up-to-date information. On the other hand, we could use the mobile computer solely as a (very) remote interface, running a terminal emulator or possibly an X server with the application running on a central site. Although we shall see that there is room for individual application specific choices, there are also good guides from the tasks being performed which force some placement decisions. central to understanding these choices is the notion of pace.

2.1. Pace and bandwidth

If one is looking for single attribute to describe a communication channel or network, it is likely that the first candidate would be bandwidth. Indeed, the word is often not used solely in its information theoretic sense, but has entered everyday technical vocabulary. For example, in comparing video with audio communications, the advantages of video are often expressed in terms of greater bandwidth. It is certainly true that video requires greater bandwidth to transmit and furthermore our human visual system has a greater information processing capacity than our aural system. However, the difference between media types is clearly qualitative not quantitative. Although useful as a measure, bandwidth is clearly insufficient.

In previous work I have argued that for the user of a system it is rarely bandwidth which is critical to interaction, but instead pace: the rate at which the user can interact with others or with an application (Dix 1992). When users type, they don't care how many megabytes pass across the network, all they notice is whether the response time is acceptable. Pace is important for both individual users interacting with data and also for users communicating with one another. Many of our innate abilities to act and communicate depend on the feedback we get from the environment and from others and it has recently been recognised that these interactions are intimately tied into our very thought processes (Suchman 1987).

The pace of user interaction is driven by a range of factors. The network latency is obviously one such factor, but not the only one. If each user interaction involves a large amount of data to be transferred, then the bandwidth may dominate in determining the pace of the interaction, or if each interaction requires a large

number of small network exchanges, then an inherently low latency network might still appear slow. At a higher level, delays may be caused by buffering and processing at various points, both at a remote site and locally, or if you are interacting with a remote user, then it may be the speed of the remote users actions which dominate. Finally, the nature of the interface may determine the effective pace. If users do not notice when they are required to respond or when they have received a response, then their interaction with one another will be slowed down.

For wireless networks, the picture is further complicated by delays due to interference and packet loss and also the possibility of a completely lost connection with subsequent set-up time as the connection is re-established. These serve to reduce the pace of interaction, but in an unpredictable manner. One of the few well established facts about delays in user interfaces is that whereas slow response times are bad, variable ones are worse (Shneiderman 1982).

Figure 1 positions mobile communication in bandwidth/pace space. The problems are obvious! Things could be worse. The planned Mars robot landing will experience 20 minute communications delays and of course posting floppy disks has a turnaround of days. However, in these situations there is at least no attempt at interactive applications! In the case of mobile communications, the variability of the medium means that one can achieve reasonable interactive response when all is working well, but interference and broken connections can push one into a situation where such interaction is impossible. Similar problems do arise with conventional applications, and many do not cope well with poor network performance: applications freeze, misbehave and crash. For mobile applications these issues cannot be ignored. So, perhaps the lessons learnt from designing for mobile platforms will be of value across the spectrum.

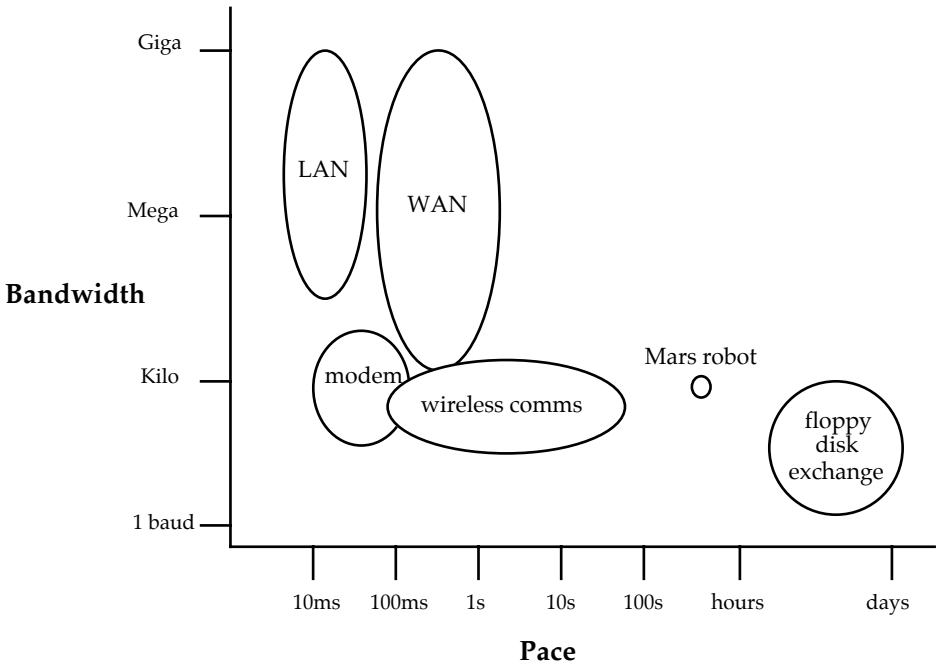

Figure 1. Bandwidth vs. Pace

2.2. Pace of tasks

In a speed dominated world, it is easy to think that faster is always better. However, what matters is not so much the absolute speed of interaction, but whether it is well matched to the tasks that users perform. If two users are writing a book together and have decided to 'own' different chapters, then a pace of interaction measured in days, mediated by email and file transfers will be sufficient (and more frequent interaction may be counter-productive). However, if drawing a line in a CAD package involved a (perhaps wireless) network access to a central database, then even a fraction of a second is too long a delay. Knowing this one can do one of two things:

(i) Match pace to task – analyse the tasks that users are performing either together or individually. If the pace of a task is greater than that possible over the communications infrastructure, then we

know that task must be performed by one individual using locally available data. This typically involves keeping local copies of shared or fixed information.

(ii) Match task to pace – restructure the task so that high pace interaction is unnecessary. For example, choosing to edit a paper in a round robin fashion means that no two authors require simultaneous high pace interaction with the paper or with one another.

We can think of (i) as being the technical fix and (ii) as the social fix.

2.3. An example

Consider a route finding application to be used by truck drivers (whilst parked!). The desktop version uses current information about traffic conditions in combination with fixed geographic information to determine optimal routes. The users see an electronic map of the country and can navigate using "zoom in" and "zoom out" buttons. The maps can be annotated by the users by freehand drawing with the mouse. If a town or city is clicked, it is added to an itinerary and when the itinerary is complete the user clicks the "find route" button and the system works out alternative routes which call at the selected towns ordered by travel time. We would want roughly similar features in the in-cab version (but using a trackball or touch screen rather than a mouse) and in addition require the system to pass on details of route choices from the driver to the central office and details about changes in jobs from the office to the driver.

Different parts of the application require different timescales:

(a) Freehand drawing. This is a hand-eye coordination task and must have feedback within tens of milliseconds.

(b) Adding a town to the itinerary and zooming. To feel that clicking a button (or a keypress) has been effectual, some feedback must occur within a couple of hundred milliseconds.

(c) Calculating the itinerary. This is a major task and the user will expect a substantial wait.

(d) Passing messages between office and driver. This sort of activity may only require a pace in the order of fractions of an hour.

Notice that for (a) the very nature of the action requires a certain pace of interaction, whereas judgements like (c) and (d) rely on knowledge of the specific task and the users expectations. To this knowledge about the users' physiology and expectations we also need to add knowledge about the domain: maps are stable and only need periodic update whereas accidents, floods and traffic jams change from hour to hour.

Typically the higher and lower pace tasks are easiest to deal with. The message passing (d) by its nature requires network transmission but is of sufficiently slow pace that only total wireless failure would hamper it. At the opposite extreme, the drawing task (a) must happen locally, even the best wireless connection would be unable to deal with this pace of activity. Note that this is not a negotiable position. Freehand drawing is impossible without a sufficient pace of feedback. If this proves impossible (say because some central information is necessary) then we must adopt a more radical redesign, changing the way the task is performed.

2.4. The middle ground

The most problematic areas are those which fit within the broad range of pace which may occur with mobile communications, from seconds to minutes. Tasks (b) and (c) fit into this category. If we take a pessimistic outlook, we could plan for disconnection to be the norm. In this case we would be forced to conclude that all the data for tasks (b) and (c) would have to be held locally or, if this is impossible, that the tasks have to be radically redesigned to be less interactive, perhaps forcing the driver to do daily route planning. However, this would squander the very advantages that mobile communications should give us. On the other hand, if we take an optimistic view and consider the pace of communications to be sufficient to allow central processing of both (b) and (c), what will happen when there is a temporary disconnection or interference?

Consider task (b), each time the user clicked a map location the coordinates would be passed back to the server which would identify the town name, add it to its copy of the itinerary and tell the in-cab machine the name of the town it added. In normal conditions the feedback from the server would be within the fraction of a second required, but if there were any interference the delay would stretch to seconds or longer. Given a lack of immediate feedback the reflex action of most users is to click again, leading potentially to a double entry in the itinerary.

This sort of problem will be familiar to anyone who has constructed distributed systems even over fixed networks. However, note that recognising such trouble spots is not based on a professional eye, or extensive user testing, but on a systematic comparison of the pace of tasks with that which can be expected by the network infrastructure. The systematic nature is crucial if one is to have any confidence that all significant problems have been considered. This is important in any interface and essential if the domain is at all safety critical.

Having identified such problem areas one can adopt a range of generic solution strategies:

(i) Change the location of functionality. The technical solution, play with where we put information at a finer level. For example, we could have information about town locations in the in-cab computer and use this to build up the itinerary. This would allow (b) to always be performed locally. However, road conditions change and instead of continually downloading this information, we could still allow the route calculation to be performed centrally. As this operation normally involves a substantial delay, the extra time to upload the itinerary would be acceptable and the user could be warned of any additional delays without breaking the flow of the task.

(ii) Change the semantics. Whenever there is a possibility that a user will think an input has been missed, it is likely that operation will be repeated. Where possible the designer should seek to make such operations idempotent, that is the second one is ignored. Although this is not always possible, it is a good rule of thumb.

(iii) Change the feedback. Note that the original pace requirement at (b) was for some feedback. The rise of direct manipulation interfaces has encouraged interfaces to be built which rely on the feedback of the effect of actions – in this case the name appearing in the list. However, low pace responses, whatever their reason, should be regarded as a form of mediated interaction. The user requesting the action and the result of that action are separate events. There should therefore be feedback at three levels:

- feedback that the input has been noticed, e.g., highlight the button, or audible click.
- feedback that the action is happening, e.g., a line of dots is added to the itinerary.
- feedback when the action has happened, e.g., the dots change to the town name.

The first of these may often be a default effect of the interface toolkit for on-screen buttons, but may be confused with cancelling the button (see (Dix and Brewster 1994)). The second is often supplied in the form of an hour-glass, or other progress indicator when the programmer thinks that something major (like reading a file) is happening. However, in an application over wireless networks virtually any action can be major. The last is the normal effect of the operation.

(iv) Change the metaphor. We may decide that it is too memory intensive to store every map locally, instead only a subset would be cached. If the names of the operations "zoom-in/zoom-out" were replaced by "fetch area map/see national map", then the user would be encouraged to expect a longer wait. One could even have a virtual glove compartment where a selection of maps were stored into which additional maps could be loaded. Note this is both deliberately stepping away from a direct manipulation interface and also deliberately destroying the illusion of network transparency espoused by most distributed platforms. This is because over any form of slow or unreliable network distribution is never transparent. Only by admitting this can we choose interface metaphors which give the users a faithful virtual world in which to work.

The choice of solution depends on a range of technical and interface design trade-offs, and the chosen solution may involve a mixture of the solutions.

2.5. Local storage and caching

Most solutions to increase the availability of data involve some form of replication or caching. This may be done by hand or automatically. The former is probably most common in everyday portable computing. Files are downloaded onto the portable when it is connected to its host machine either directly or via a modem. The user has the sole responsibility to choose what is downloaded and to update the central site when local changes have been made. One of the reasons for the popularity of Lotus Notes is that, within its own database, it handles issues of replication almost transparently. In contrast, Coda attacks the replication of the normal file

system , using a mixture of user preferences and whole file caching (Satyanarayanan et al. 1990; Kistler and Satyanarayanan 1992).

If all the required data were stable and there were unlimited local storage, then one could simply keep everything that could ever be wanted on the portable computer. However, there is always more central data than you can hold locally and even the data you hold may be changing. There are two principal causes of data volatility: changes in the environment (as in the case of traffic conditions) and changes due to the actions of other people. We will return to the latter of these in the next section.

When dealing with disk or CPU caching the algorithms used must be automatic, simple and fast. It is no good having an optimal disk caching algorithm which takes more time than it does to access the disk anyway! However, the costs of failure are not excessive. A certain percentage of cache misses is to be expected and merely has the result of reducing the overall performance of the system. It is to be expected that some of the lessons from low-level caching will be of benefit at a higher level. Indeed, Coda uses a simple on demand strategy for down loading files and least recently used discard policy. However, the balance of costs for caching on a portable computer is different from those at a lower level. Imagine a field engineer is accessing layout diagrams for a faulty electricity sub-station, half way through the repair the communications go down, and the relevant part of the plans are not on the local machine. A 'cache miss' may cause several minutes delay, perhaps longer if the lost connection is due to bad weather. In the mean time, which was the 10,000 volt cable?

Because the costs of failure are higher and the time scales are longer, caching algorithms for portable computers can afford to be more complex than their low-level counterparts. In particular, they may involve the user. For example, the CODA file system allows the user to set up a preferences file listing directories and files which should be permanently cached. Similarly, Lotus Notes gives the user some control over replication policy. This user involvement is crucial. Even at a low level it is normal for different CPU caching policies to be used for data, instructions and stack frames. You have to understand the application domain in order to design effective caching strategies. Some of this understanding may be supplied by the application designer, but often it is only the user who knows what will be required. For example, our truck driver would typically keep large scale paper maps of frequently visited towns. If the electronic system could not store maps for the whole country, it could download relevant ones each day based on the expected itinerary. If changes are made during the day, at least only the extra maps need to be downloaded over wireless, both reducing typical delays and costs.

Unfortunately, in modern applications it is getting harder to work out what will be needed for any activity. It is no longer the case that one can identify a single file as being relevant. It will often reference other files, databases or spreadsheets using live links, hypertext anchors, or other forms of cross-reference. It may be possible to find such links by examining the files, for instance by searching for '#include's in C programs or anchors in HTML, but unfortunately many links are hidden within proprietary formats. The moves by operating system vendors to make these linkages explicit, as in the Apple Edition Manager (Apple Computer Inc. 1993), can only be good.

The lesson is that neither the system nor the user have sufficient knowledge to choose what should be locally cached and what should be loaded on demand. Instead this must be a cooperative activity involving both parties. However, this imposes a difficult interface design task. There are no easy answers, but often there is information in the system which can help in the process. This will often be domain specific, for example, the lorry driver's rough itinerary will be known in advance. It is also often the case that similar access patterns are repeated so the system can predict additional useful information to download . Finally, an appropriate choice of interface metaphor can make the process far more natural for the user. For example, the in-cab system could explicitly show downloaded maps as small icons in the interface – a sort of virtual glove compartment! Rather than a hidden activity behind the scene, the need for caching can thus be projected into the user interface in a way which makes use of the user's day-to-day experience of the world.

3. Cooperation: Awareness and Feedthrough

The focus so far has been principally on issues of single user interaction. For a mobile worker, it must always be possible to work individually in case communication links are lost – hence the need for local caching and replication. However, the existence of relatively good wireless communication makes more closely synchronised work possible. In this section we'll look at some of the issues arising from this.

The time–space matrix has been very influential in categorising groupware (Rodden 1991). Systems are classified using two axes. The space axis looks at whether the users are operating at the same place (local) or a

different place(remote). For mobile workers we are principally concerned with remote cooperation (although of course we should not ignore the cooperative needs of several mobile workers at the same site). The second axis, time, is more interesting distinguishing those systems where the users work at the same time as one another (synchronous) from those where the users work at different times (asynchronous). Remote synchronous systems are typically based on high speed networks and include both communications tools, such as video conferencing, and various forms of shared editors and drawing tools. Remote asynchronous systems are commonly based on message passing over lower speed networks, the archetypal example being email. In fact, the characterisation in terms of working at the 'same' time or at 'different' times is rather misleading. Two people may be using email at the same time, and indeed send each other messages. A better way of looking at this distinction is in terms of the pace of interaction between the participants. That is the timescale over which they synchronise their activities. In typical 'synchronous' systems this in the order of seconds or less. In asynchronous groupware the pace is often in the order of hours or days.

Again mobile wireless communication sits in a strange middle ground with respect to this distinction. When a connection is established we should be able to support synchronous groupware applications, but, we must also consider the possibility of deliberate or unintentional disconnection, in which case the users are forced into working asynchronously. To make things more confusing, even the 'asynchronous' periods are likely to involve contact at a greater pace than is normal in other forms of groupware. Where groupware systems support both synchronous and asynchronous interaction it is typically at the explicit request of the users whereas with mobile communications the change may well happen spontaneously when connections are lost.

To unpack some of these issues as they relate to mobile wireless communication we will use parts of a simple framework for CSCW which has been used to taxonomise and analyse existing groupware (Dix 1994; Dix et al. 1993) and also as a design framework for novel groupware applications (Miles et al. 1993).

3.1. The CSCW framework

Consider any cooperative working situation. The word cooperation presupposes that there are one or more participants involved. These are denoted by the circles labelled 'P' in figure 2. It is also normally the case that participants will communicate with one another, which is represented by the arrow between the circles. The execution of the 'work' will involve one or more of the participants interacting with objects and things in the electronic or real world. These things, which may be in different places and controlled by different people we call collectively the 'artefacts of work' and are denoted by the single circle 'A'.

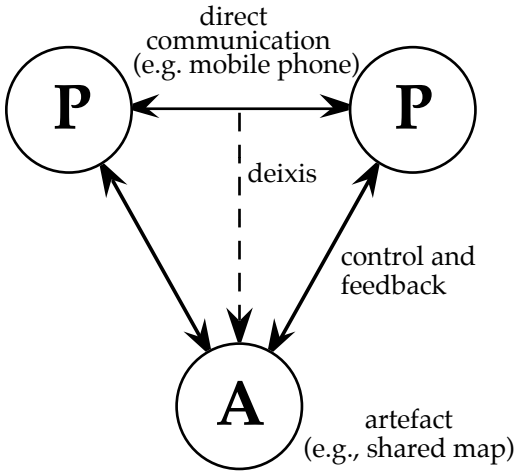

Figure 2. CSCW framework

If the participants are cooperating, then we may expect that their direct communication is about and makes reference to the artefacts on which they are working. In face-to-face working, these references between conversation and action are very rich, involving physical actions such as pointing, explicit spoken references (e.g., 'grab the left handle') and implicit dependence on the environment and context (which way you are looking when you say 'left'). Furthermore, these forms may be mixed (e.g., 'turn left when you get here <*points at map*>'). These various forms of pointing and description are called deictic reference or deixis and are indicated by the broken arrow in figure 2.

3.2. Linking communication and action

It is often the relationship between communication and action which is lost in remote communication, even over fast networks. Often explicit means are introduced to help, for example, distributed shared editors often include a 'group pointer', an arrow or hand icon, which can be picked up by one participant and is then shown on all screens. This enables the participants to point at things in the shared objects. If there is any appreciable lag between the movement of the group pointer and its appearance on other participants' screens then there is the danger of a breakdown in communication (e.g., when the pointing participant says 'here', the others see the pointer at the wrong position). These problems are particularly bad in mobile working if the voice channel (supporting direct communication) is separate from the data channel. If a group pointer were supplied then the data connection could be lost, but this not be apparent as the pointing participant's voice continues.

In asynchronous applications the need for reference between data and communication is often ignored, but is supported in some systems. The most common form of support is in the form of annotations. These are found in research co-authoring systems, such as Quilt (Leland et al. 1988) and Prep (Neuwirth et al. 1992), and also in commercial word processors such as Microsoft Word. Also, many email systems support file attachments, which are a way of referring to work, albeit at a course level. To contrast these, annotations can be seen as supporting the link by embedding the communication within the data, whereas attachments embed the data within the communication (remember that an attachment is usually a copy rather than a reference to a file). The relationship can be somewhat more symmetrical where the system is based, as in the case of Quilt, on a hypertext structure. As noted previously, these systems are all built assuming a pace of interaction of hours or days. Thus these mechanisms are suitable for adoption during long periods of disconnection. However, to expect participants to switch from simply chatting about things to writing explicit annotations during periods of temporary disconnection is unacceptable. In such cases the best we can do may be to simply make the participants aware that there is a communication problem.

3.3. Awareness

The temptation as a computer scientist is to address problems by thinking of clever algorithms to solve them. In distributed systems this often takes the form of different types of transparency – using clever algorithms to hide the fact that things are distributed from the application programmer and the user. However, a lesson that has been repeatedly learnt in the field of CSCW is that people are very adaptable and, given suitable information about what is going on, they are often able to solve problems for themselves! In particular, transparency is often precisely the wrong approach, hiding exactly the information that users need for effective cooperation.

The word awareness has some important connotations. Firstly, it suggests a low effort, often sub-conscious reception of appropriate information, as opposed to the information being available if one is prepared to look for it. Having information on-screen, for example in a status line, may not be sufficient, even if the user knows it is there. Secondly, awareness should not interfere with the users' primary tasks. So, for example, putting up a dialogue box after each keystroke is not a good way to give awareness of memory usage! Mechanisms for increasing awareness must therefore be subtle and implicit – a difficult design goal.

Various forms of awareness are important in cooperative working. In figure 3, imagine that you are the participant on the right and your colleague the one on the left. You are manipulating a shared artefact. The shaded arrows represent the different kinds of awareness:

(a) that you're there

(b) what you did

(c) how you did it

In face-to-face collaboration the first of these is obvious. In most phone conversations it is also no great problem as we implicitly give little 'uhms' and 'ah-has' (called back-channels) when we are listening, and of course, can be heard when we are talking. This gets a little more difficult with half-duplex channels, but people have developed ways of coping. However, in a mobile application, the participants may be focused on using the application itself, and may be largely silent. It is then not at all obvious whether a period of silence is due to the other person being quiet or due to a break in communications.

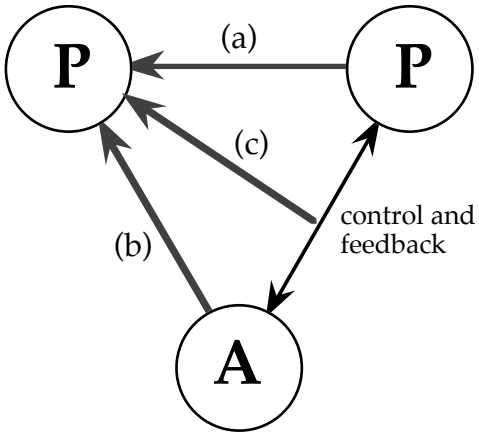

Figure 3. Forms of awareness

The second form of awareness, (b), concerns the fact that the effect of your actions on a shared object may be visible to other people. This does two main things. Firstly, it tells the other users that you are busy (if they are aware of who is doing the manipulation) – that is, it helps awareness of type (a). Secondly, the other users become aware of the changed information. Note that both of these can be seen as a form of communication between the participants – *communication through the artefact*. In many cases this communication is more important for effective cooperation than direct communication. However, to be effective, the pace of such communication must match the cooperative task. In the beginning of this paper we were mainly concerned with the pace of feedback for the participant, but it is also important that we maintain effective *feedthrough* to the other participants. Traditional distributed databases and file systems prevent simultaneous access to the same object, or at best allow a static snapshot of the changing data. These remove the need to maintain feedthrough (and hence the associated cost), but in doing so prevent any communication through the artefact and hence much close cooperation on a task.

It is usually the case that feedthrough can operate at a slightly slower pace than feedback. This is because you know that you have pressed a key and so expect a response. Your colleagues need no such immediate feedback. We can take advantage of this over low-pace or low-bandwidth channels by, for example, sending groups of keystrokes every few seconds. However, this requires some care as it is precisely when feedback and feedthrough become too out of step that problems of deictic reference occur.

The last kind of awareness, (c), concerns the way in which a change occurred. For example, if you draw a line and then the line simply appears on everyone else's screen, they won't know which end was the beginning of the stroke. Arguably, if this were important enough, you would have made it explicit using an arrow rather than a line. However, it is the nature of the informal collaborative use of such media that we implicitly make such distinctions and it is the ephemeral, rather than the persistent meaning of many marks which is most important. This is evident from the appearance of most white boards at the end of a meeting!

Unfortunately, reducing the pace of feedthrough by chunking will reduce this form of awareness. For example, if we deliberately chunk several keystrokes together before transmitting them, then the other users will see the typing appear in bursts, rather than a continuous stream. Delays due to packet loss or temporary interruptions to service will only make this worse. To some extent this is an inevitable loss due to limited communications channels, however, the effects can be alleviated by use of animation techniques. For example, when a packet containing several characters arrives, they can be displayed one by one rather than in a single burst. Note that such animation is only mimicking the original users actions, but offers some feeling that it is someone else working with you.

All of these forms of awareness fail or are misleading when communications fail or suffer unusual delays due to disconnection or protracted interference. Thus a fourth form of awareness is needed – awareness of the state of the communication channels. Remembering the general principles for awareness information, it is not sufficient to have a status line or logging window, nor is acceptable to put up a dialogue box every time a communication problem occurs. Although it is easy to reject bad alternatives, it is considerably harder to find good ones. The best solutions will add information to points of shared focus. For example, in a shared text editor where other users' cursors are shown then any temporarily disconnected users can have their cursors greyed out.

4. Consistency and Resynchronisation

The need for feedback and high availability required local copies of shared data. However, in a cooperative application, the need to have a shared focus for work and communication will mean that several users will need copies. That is, shared data is replicated on several users' workstations leading to the possibility of inconsistent data.

4.1. Why mobile applications are difficult

The traditional approach to this has been to use pessimistic algorithms which prevent simultaneous conflicting updates even in the face of complex network partition failures (Davidson 1989). However, the cost of such preventative measures is a reduced availability of data for updates. To counter this a range of optimistic algorithms have been proposed and implemented. The general approach of such algorithms is to allow possibly inconsistent updates and then to detect and possibly correct and inconsistencies at a later stage.

It is clear that mobile computers cannot hold long term locks or similar mechanisms necessary for pessimistic algorithms without seriously jeopardising availability. Even where wireless networks can be employed full-time there will be periods of unplanned disconnection. However, there are also problems with optimistic algorithms.

The normal assumption underlying optimistic algorithms is that conflicts and hence repair will be infrequent. For example, the designers of CODA ran extensive tests and estimated that when a UNIX file was updated there was only a 0.5% chance that it would be updated by a different user within a day. That is only 1 in 200 file updates would be likely to require conflict resolution after a days disconnected operation. At first sight this looks hopeful where wireless networks are being used, typical periods of disconnection are likely to be in the order of minutes rather than days. Unfortunately the relatively high availability and pace of the medium is likely to cause trouble for cooperative applications. Because an acceptable bandwidth and pace are available we are in a position to support synchronous applications. For example, in the in-cab route system we may allow several remote users to draw on the same map, simultaneously seeing each others annotations. However, now when a disconnection occurs the users are likely to be simultaneously updating the same map. That is the availability of synchronous activity means that conflicts are more likely during the periods of (enforced) asynchronous activity. Furthermore, if the system is attempting to handle temporary interruptions to service, the users may not even be aware of the potential problem.

As with other issues in this area, solutions depend on a mixture of technical fixes, user involvement and domain-specific methods. Any solution requires at least two stages: conflict detection and conflict repair. The repair may either involve some completely automatic merging or an element of user involvement. Furthermore, the repair process may well proceed over a limited bandwidth network and so must attempt to minimise unnecessary transmission. A determining factor for all this is the type of data and the manner in which it is update, so we begin with a discussion of this, before moving on to the other issues.

4.2. Types of data and update

If we want to access existing centralised data or modify an existing interface, then we may have little control over the types of data and updates that are allowed. However, it is often the case that we can influence these choices for some or all of a system if we do so early enough in the design process. These choices make a fundamental difference to the difficulty of handling multiple updates and the methods used when conflicts occur.

We can consider three classes of data:

- Independent facets or objects — The data comprises a collection of objects each of which can be updated without affecting the others. The smaller such objects are, the less likely it is that any individual item will be updated by more than one person. Thus conflict is often avoided. However, this only works if there are no integrity constraints between the objects.
- Linear, growable objects — The primary example being text, where insertions, moves etc. may change the absolute and relative positions of text ranges.
- Other structured objects — For example, trees, graphs, file directories, hypertext etc.

To these data objects we can perform several types of update:

- Monotonic — where items are only added and the order of addition is (more or less) immaterial. In these cases merging updates is little more than a process of concatenation. Examples include annotations in co-authoring environments such as Quilt, bulletin boards and some forms of hypertext. Even some kinds of databases, such as bibliographies, are largely addition based with few amendments or deletions.
- Single owner or single stream of update — we may ensure that each object can only be updated in one site by enforcing ownership or otherwise. This is a form of pessimistic concurrency.
- Reflection of the world — the information reflects the current state of some aspect of the real world. In this case, we may often simply assume that the most up-to-date information is correct.
- Application specific updates — often the semantics of specific objects offers opportunities for easy and safe simultaneous update. For example, incrementing or decrementing a counter (a form of monotonic update) (Schwarz and Spector 1984). Of course, these are often guarded by integrity constraints which reduce the range of use.
- True distributed update — when none of the above hold! A special case of this is:
- Major structural change — In most areas of computing structural changes are more complex than simple updates to data. Many databases find changes in their schema difficult even when operating on a single machine – what hope for mobile workers! Even when data objects are independent they may be named or organised using a complex structure (such as a hierarchical file system). Also, part of the complexity of 'simple' linear text is that every update may change the location (the name) of subsequent text.

Opting for simpler forms of data and update one can make the job of designing distributed applications easier and often achieve better results for the user. This idea was developed in multiple source control, a proposed mechanism for handling data-resynchronisation problems (Dix and Miles 1992). A branching version tree is stored at each site (similar to those in conventional version control systems like RCS (Tichy 1985)). Changes to version trees are by addition only, hence updates to linear text (a difficult case) have been converted to monotonic version-tree updates.

4.3. Detecting conflicts

Detecting updates conflicts requires three things: identifying the 'same' objects at the different sites, detecting any differences, and if they are different working out which has changed. The identification problem is itself not trivial. File-based systems often rely on file-name matching, but need to cope with differently structured name spaces and file name conventions. Custom applications can use some sort of unique identifier, often a combination of a site id and a per-site sequence number. Difficulties may arise even here if the 'same' entry is simultaneously created at two different sites (consider the merging of road accident reports), leading to alias records and further complexity!

Having located the two copies of the same object at the different sites, one needs to detect whether they are different. This may be based on an actual comparison of contents (accelerated using checksums), but is more often accomplished with timestamps. Even assuming that the clocks are synchronised, two problems arise:

- If one copy is newer than another has only the newer copy been updated, or both copies?
- If only one copy exists does it mean it is a new object, or is it an object which has been deleted at one site?

In systems which enforce strictly sequential updates, the first is no problem – the newer copy is always the correct one. However, in general systems must keep track of the state of the system when it was last synchronised. For the first problem a timestamp of the last synchronisation is sufficient. If both copies are dated after the last synchronisation then there is a conflicting update. Deletes take more care and may involve permanent records being kept. To illustrate this let's look at Liveware.

Liveware is an unusual database designed for maintaining information shared by loosely structured groups of people (Witten et al. 1991). It has been used to maintain contact information among the Scottish HCI community. Each person has a copy of the database and when meeting other people you take a copy of the database with you on floppy disk. When inserted the copy on floppy merges with the hard-disk copy on your colleagues machine. When you return to (one of) your own machine(s) you perform a similar process and so information spreads from person to person. In early work this was likened to a form of benign virus. Each

person is responsible for their own contact details and so each record has a single owner and hence a single stream of update. This means that records can be merged on a simple most-up-to-date principle. If a record exists on one database and not on another the record is simply copied. However, this runs the risk that a deleted record could spring back into life if an older copy of the database (with the record still in existence) were merged. To avoid this special 'deleted' records have to be maintained. because there is no limit on the number of copies of the database, the deleted records have to be kept around indefinitely causing a 'space leak'.

4.4. Automatic merging

If disconnection is reasonably frequent then it is obviously good if systems can automate the merging of updates wherever possible.

Some file transfer programs, such as Laplink have options for automatically merging file systems. The mechanisms have been designed for someone wanting to merge a portable computers file system with that of a desktop machine. However, the implicit assumption is that both are private files and will not have been updated by anyone else, the single stream of update again, and so, like Liveware, they use the most-up-to-date policy. They therefore deal badly with deleted files and are certainly not optimal for multi-user applications.

Systems designed with multi-user update in mind, such as Coda and Lotus Notes, correctly identify conflicting updates by keeping explicit track of who knows what. Coda automatically manages the case when only one update has occurred, keeping the most up to date, and flags true conflicts for the user to deal with manually. Lotus Notes is somewhat more sophisticated. It handles all the simple cases, but options can be set to allow either manual or automatic action on conflicts, although the automatic options merely involve adding the alternative versions of a note as 'responses' to the note.

It is not obvious whether finer granularity is desirable in an automatic merge. For example, would it make sense to put together fields from different versions of a database record? However, in specific applications totally automatic merging may be possible. For example, in the Grove distributed editor any single character insert or delete operations can be performed concurrently (Ellis and Gibbs 1989). The updates from each site are broadcast to all others and are transformed into actions which have an equivalent effect at the recipient site. The Grove algorithms were proposed to cope with the short lived concurrency for replicated text over any network, but not aimed at long periods of disconnection. However, dynamic pointers, a similar technique have been applied to the analysis of long term concurrent activity (Dix 1995).

Again note that it is possible to deliberately design data structures to facilitate merging at any pace. For example, annotations to a map could all be performed in XOR mode. This would mean that the order of updates, whether creating, changing or deleting shapes, would not matter.

4.5. User involvement

There will always be cases where automatic merging is either not desirable or impossible (for example if two people set the same attribute to different values). In these cases the users have to manually restore consistency. Systems vary in the extent to which they support this. Many simply give up when true conflicts are found. For example, Coda hands control back to the user when a write-write conflict is discovered, Similarly, in the software engineering domain, the version management system described in allows optimistic concurrency in certain circumstances, but restricts resynchronisation to 'safe merges', that is when there is no true conflict. The support in Lotus Notes at least associates the variants with the note itself, but offers no help in merging the versions.

There are, of course, various file difference tools which can be used to facilitate manual merging. These range from the UNIX 'diff' program to flexible diff, a visual tool which makes it easy to differentiate major and minor changes (Neuwirth et al. 1992). Unfortunately, simple differencing between two versions of the same file gives the users exactly the same problems as the system has in determining which files are in true conflict. If a paragraph is different does that mean that it has been updated in both documents or only one. If the latter, then which is the newer version of the paragraph? A two file difference cannot determine this for the user. The updated versions must also be compared with the most recent common ancestor (hat is, the copy of the file just before the two sites diverged). This third file could either be displayed to the user or simply used by the system to determine which parts have been updated. Figure 4 illustrates the problem: without the original it would be impossible to tell whether "trapping her" or "pinioning her arms" is the updated text.

Figure 4. Two updates based on a common original

Manual merging is acceptable after long periods of disconnection, especially if it is deliberate. However, it is likely to be annoying after short periods of disconnection due to interference or temporarily broken connections. So, if the pace of interaction is such that involving the user is felt to be inappropriate, it may be better to accept a certain level of inconsistency between different users interfaces. For example, we may allow non-XOR operations for map annotation and not worry if the occasional pixel is different. Algorithms which allow a level of inconsistency may be considerably simpler, less computation and network intensive than their correct equivalents. In a sense this still migrates work to the users who have to be aware of and compensate for the differences in their interfaces.

4.6. Reducing the impact on the users and on network traffic

The Coda developers say that "typical disconnected sessions ... lasting a few hours required about a minute for reintegration". Since disconnection is planned there are no comparable figures for very short periods of disconnection. For the same reason, the reintegration phase is not concurrent with normal operations. Whereas this sort of interruption is acceptable for deliberate disconnection it would not be for temporary unplanned broken connections. In addition, measurements show that the reintegration phase may involve the transfer of several megabytes of data, acceptable for direct connection to a local area network, but not over typical wireless links. One reason for this high volume is that the granularity of Coda is whole files. This is compounded by the fact that many UNIX operations create and update many large files for the user. In contrast, in an application like Lotus Notes, the basic unit (the note) is on average smaller and there is less automatic generation and update of notes. The lessons are clear:

- User input not system output — People can only generate so much data in a given time and hence resynchronisation mechanisms which operate closer to the user's input are likely to have lower bandwidth demands than those which operate on the outputs or effects of user actions. Given some level of concurrency is inevitable in non-centralised architectures, the algorithms to deal with high pace concurrency (such as the Grove algorithms) may be able to be modified to handle batches of operations.
- Minimise granularity — Where possible application specific algorithms should identify which portions have been altered during periods of disconnection, to avoid sending large units between sites.

The sort of incremental algorithms suggested by the above usually require that the sites need extensive knowledge about each other. Not only must the sites know *when* they were last synchronised, but also know something about their mutual state at that point.

Summary

We have seen how a systematic analysis of the pace of different user tasks can indicate where data needs to be cached or replicated at a local machine and where remote operations are possible. Also we have seen how collaborative work requires awareness of other users' presence, feedthrough of the effects of their actions on shared data, and awareness of the current state of communications.

However, these conflicting demands of high interactivity and collaborative working mean that we both require replicated data and expect that data to be being accessed simultaneously by different users. When disconnection occurs the potential for inconsistency is high. Existing mechanisms assume infrequent, planned and relatively long periods of disconnection, and require extensive modification to cope gracefully with occasional unplanned broken connections.

Although some solutions and design advice has been given, these are by no means solved problems. Each situation requires a different balance between the various conflicting factors. Also there is at present no satisfactory generic implementation framework. Although general advice can be used, each application must be individually crafted for a portable platform.

References

Apple Computer Inc. (1993). *Inside Macintosh: Interapplication Communication*. Addison-Wesley.

S. B. Davidson (1989). Replicated Data and Partition Failures. *Distributed Systems*, Ed. S. Mullender. New York, ACM Press – Addison Wesley. 265–292.

A. Dix and S. A. Brewster (1994). Causing Trouble with Buttons. *Ancillary Proceedings of HCI'94*, Glasgow.

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

A. J. Dix (1992). Pace and interaction. *Proceedings of HCI'92: People and Computers VII*, Cambridge University Press.

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

A. J. Dix (1995). Dynamic pointers and threads. *Collaborative Computing*, **1**(3): 191–216.

A. J. Dix and V. C. Miles (1992). *Version control for asynchronous group work*. Technical Report YCS 181, Department of Computer Science, University of York.

C. A. Ellis and S. J. Gibbs (1989). Concurrency control in groupware systems. *Proceedings of 1989 ACM SIGMOD International Conference on Management of Data, SIGMOD Record*, **18**(2): 399-407.

J. J. Kistler and M. Satyanarayanan (1992). Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, **10**(1): 3–25.

M. D. P. Leland, R. S. Fish and K. Robert E (1988). Collaborative document production using Quilt. *Proceedings of CSCW'88*.

V. C. Miles, J. C. McCarthy, A. J. Dix, M. D. Harrison and A. F. Monk (1993). Exploring designs for a synchronous-asynchronous group editing environment. *Computer Supported Collaborative Writing*, Ed. M. Sharples. Springer-Verlag.

C. M. Neuwirth, R. Candhok, D. S. Kaufer, J. Morris and D. Miller (1992). Flexible Diff-ing in a Collaborative Writing System. *CSCW'92 – Proceedings of the Conference on Computer-Supported Cooperative Work*, ACM Press.

T. Rodden (1991). A survey of CSCW systems. *Interacting with Computers*, **3**(3):

M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel and D. C. Steere (1990). Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, **39**(4): 447-459.

P. M. Schwarz and A. Z. Spector (1984). Synchronising Shared Abstract Types. *ACM Transactions on Computer Systems*, **2**(3): 223–250.

B. Shneiderman (1982). Response time and display rate in human performance with computers. *ACM Computing Surveys*, **16**(3): 265-286.

L. A. Suchman (1987). *Plans and Situated Actions*. Cambridge, Cambridge University Press.

W. F. Tichy (1985). RCS – a system for version control. *Software Practice and Experience*, **15**(7): 637–654.

I. H. Witten, H. W. Thimbleby, G. Coulouris and S. Greenberg (1991). Liveware: a new approach to sharing data in social networks. *Int. J. Man-Machine Studies*, **34**: 337-348.