

Observations on practically perfect CSCW

Harold Thimbleby
Stirling University

David Pullinger
Institute of Physics

September 29, 1999

Abstract

Technical systems are so powerful that there is a temptation to try to provide ‘perfect’ support for any cooperative work practice. Some CSCW problems result from designs motivated by optimising technical criteria. A very high bandwidth network, for instance, can support a wide variety of work patterns; the consequent focus of solving the salient technical problems gives the impression that CSCW is a merely social concern. We suggest a class of CSCW system property, *observational properties*, which are required by users. This motivates a description of an appropriate technology to support such properties. Observational properties tend to be easier to support, so that systems may be more robust or able to handle degradation more gracefully. When an observer cannot distinguish a CSCW system from a perfect system, we say the system is *practically perfect*. Practically perfect CSCW can be achieved either by perfect technology or, more appropriately, by judicious design of the CSCW application. Such systems are particularly appropriate for mobile and remote activities, including the activities of personal users without permanent access to reliable and timely communications infrastructures.

1 Introduction

CSCW systems are often designed and viewed from a technological point of view, where one is concerned with consistency, integrity and other properties. However, for many purposes, such global system properties are not relevant to use of the system by users who are not concerned with the implementation, the state of the art of the technology, or the moment-to-moment state of remote systems. However in CSCW we are interested in systems from each users’ point of view, and this quite proper concern should be reflected in the technical system requirements.

We will argue that one can have useful CSCW systems without maintaining global and some other technologically-oriented (‘implementation biased’) properties, with a major benefit being that one does not need high bandwidth communications to implement useful and for practical purposes, perfect CSCW systems. Turek (1992) provides a useful summary of technical protocols.

One of the ironies of modern computing technology is that while many individuals now operate personal computers whose power rivals that of large installations, they are without the infrastructure of support that traditionally accompanies such systems. The costs of installation, connection, software licensing, and communication tariffs far exceed the capital and running costs of

a personal computer. The benefits of CSCW, based on distributed systems technology, are inaccessible to the personal user. It might therefore be appropriate for these users (rather, their systems' designers) to consider other ways of supporting their requirements than by providing over-specified technology.

1.1 General requirements: transparency, consistency and grace

Transparency. CSCW addresses the problems of providing support for many individuals, possibly mobile, distributed geographically, and who wish to cooperate on shared tasks using computers. Ideally, the computer in a CSCW system should be *transparent*, as indeed it is in other technology-supported cooperative work systems, such as telephone communications. CSCW transparency applies to (a) the choice of work units to share (b) the cost of actually accomplishing the sharing (c) the technical protocols of sharing, such as the network route taken, acknowledgement of receipt, record locking, distributed version control. Ignoring criteria imposed by effective execution of the CSCW task itself, these factors can be traded against each other: for example one can lower *b* but increase *c* by using manually managed communications.

The obvious solution to CSCW transparency is to install a high bandwidth network to support a distributed system (many researchers in CSCW consider present network bandwidths insufficient): CSCW transparency is then achieved because a distributed system provides technical transparency (Coulouris & Dollimore, 1988). This solution has the merit that it is independent of the CSCW tasks supported, and imposes no (technical) restrictions on the tasks.

Consistency. A CSCW system is *consistent* when no user can require it, as a result of legitimate operations, to hold information anywhere that it does not. If there is a network failure, then a user's operations merely have the intention of transmitting messages to other users in the system, and the system is inconsistent.

Grace. For many technical and human reasons, CSCW systems are subject to fault: error, noise, partition failure, etc. Although in the short term any fault may cause inconsistency, we require that after some repair, in the long term users can recover; furthermore, we require that there is a procedure to recover with less cost to the users than of repeating the work. When we gloss the problems of intentional human-caused faults (such as the maliciously uncooperative work content) we say that a system is *graceful* if it satisfies this requirement. Ideally, a system will be transparently graceful: it not only repairs faults in good time, but that the user is not aware of the repair.

For example, consider a mobile user without telecommunications who is disconnected from a CSCW system. Any activity of the user runs the considerable risk of not cooperating with other users in the appropriate way specified by the task domain. In an ungraceful system, such a user would be well advised to do nothing, lest it cause or exacerbate future problems! However, in a graceful system the user can gain by continuing

to work. It is perhaps worth noting (with due gratitude) that the grace of many systems is realised by systems administrators who, by their considerable technical skills, are able to make the underlying ungracefulness of the system implementation transparent to the user.

2 Observational properties

Conventional views of distributed systems' consistency (e.g., one copy equivalence) are unnecessarily strong for CSCW systems. For example, if a user is away from his office computer, it does not matter *to him at that moment* that his laptop is inconsistent with it, for there is no way of telling and no consequences derive from any inconsistency. We therefore introduce the notion of observational properties; in this case, observational consistency.

We use the term *component* for a local subsystem, and *system* itself to refer to the aggregate CSCW system (potentially) composed from its components.

2.1 User models

At any time an idealised user u will have a *model* M_u of the system state. This model is altered in two main ways: by viewing (part of) a component or by updating (part of) a component. In the first case, the user obtains information from the component and revises his model. In the second, updating the component does not change what the user knows, but changes what the user models the system as knowing.

We will be primarily concerned with the second case: that is, with users who do not change their minds to suit the system! Thus a user performs operations in his head on M_u that correspond to the operations he performs on the components he interacts with. A user interacts over a period of time updating various components and this is what determines his model. At any moment, however, he is concerned with one component. A user only does one thing at a time, and that in one place.

We make a careful distinction between *knowing* and *modelling*: a user may know something that he chooses not to model. For example, after a telephone conversation with another user, a user may know something about the system that cannot be determined from his local component. In this case the user would probably wish to distinguish between what he knows and what he knows about the local component, that is what he models of the system. Crucially, however, a model is not a model of a component but of a system: the user is never concerned with which specific component he deals with (due to transparency).

We treat the user's model as an algebra though clearly the notion needs refining both for particular CSCW applications and for various psychological reasons. However the purpose of this paper is to motivate concepts rather than to develop towards particular applications.

A user with perfect cognitive resources can do no better than our idealised user; the purpose of modelling with an idealised user is that, sooner or later, some real user could detect an inconsistency known to an ideal user: the idealisation is safe. In particular, a real user, though perhaps far from ideal, may know critical questions (he has an oracle) so his performance in testing properties of a system is feasible. One could model a user more 'psychologically'

(e.g., modelling that the user can remember the history of actions, rather than the present state), which would perhaps be more appropriate for certain CSCW tasks, however to do so felicitously would raise a number of empirical questions that cannot be answered in general and do not need to be addressed here.

2.2 More on modelling

The way a user thinks about a system is a morphism of the operations physically performed on system components.

If ϕ is ‘how the user thinks about the system’ (i.e., is the morphism relation that respects the corresponding operations in the component and in the head, strictly ϕ_u since different users u can think differently), then for any operation f the following mapping diagram commutes.

$$\begin{array}{ccc} c & \xrightarrow{f} & c' \\ \phi \downarrow & & \downarrow \phi \\ M_u & \xrightarrow{\phi(f)} & M'_u \end{array}$$

A user can issue an operation (a computer command) f to change the component from c to c' . The diagram shows that if the user models c by M_u , following $\phi(f)$ —the mental operation corresponding to f —then c' is modelled by M'_u .

The commutivity of the diagram (the arrows can be followed from c to M'_u by either of two routes) indicates that the user can model c' equivalently by ‘thinking about’ c' (following the route $c \xrightarrow{f} c' \xrightarrow{\phi} M'_u$) or by ‘thinking about’ c and f (following the route $c \xrightarrow{\phi} M_u \xrightarrow{\phi(f)} M'_u$). *Since each choice is equivalent the user can do whichever is easiest.*

If c is large, then the user will presumably find applying ϕ tedious. When the user starts on a new system, $c = \emptyset$ (or something else trivial) initially and we can assume a realistic diagram of what users actually do is more like the following:

$$\begin{array}{ccccccc} \emptyset & \xrightarrow{f_1} & c' & \xrightarrow{f_2} & c'' & \xrightarrow{f_3} & \dots \\ \phi \downarrow & & & & & & \\ M_u & \xrightarrow{\phi(f_1)} & M'_u & \xrightarrow{\phi(f_2)} & M''_u & \xrightarrow{\phi(f_3)} & \dots \end{array}$$

From this it can be seen that users need only remember what they do to understand a component; they do not have to repeatedly keep understanding it with ϕ . We make two observations on this. First, a user may *forget* how to deploy ϕ through lack of rehearsal, and the fact that its only necessary use was in the trivial case $\phi(\emptyset)$ —which, in any case, was arbitrarily long ago. Secondly, for realistic systems, the user need rarely remember *every* operation (f or $\phi(f)$), since there will be various reductions that can be applied, greatly simplifying what the user actually needs to know. For example, if an operation is `delete x` then, likely, it and every prior operation on x (say, of the form `update x`) can be omitted from memory. Indeed, many systems are designed so that users are at liberty to delete information in order to avoid remembering anything about it!

If components contain information that is not relevant to a user, it is apparent that the morphism ϕ_u is composed of two functions, R_u that projects

the ‘relevant’ parts of the component for the particular user u , and Φ that is a ‘standard’ way of thinking about $R_u(c)$. Φ might be obtained from manuals, online help or other training material *independent of a particular user*.

As a final aside, we note that most manuals define Φ extensionally, whereas they might be considerably more effective defining Φ^{-1} (which for our more formal purposes is equivalent) intensionally. The function Φ^{-1} enables a user to convert how they are thinking about a system, its tasks and so forth, into operations implemented by that system; whereas the function Φ tells the user how to think about the computer system’s functions. These alternative approaches are conventionally termed *task orientation* and *function orientation* respectively.

2.3 Observational properties

An observational property is an observer’s empirical judgement of that property relative to what he is able to observe, necessarily located from moment-to-moment in a particular place and time. A system, for security or other reasons, may not reveal certain information, in which case a user cannot observe that information.

An observational property p is relative to a particular user u and the observed component c ; we notate this $u_p(c)$. Thus an observational property is relativistic (or solipsistic).

We will be interested in observational properties that hold for several users, with the notation $U_p(c)$ for an observational property p that holds for each user member of the set U for the component c : $U_p(c) \equiv \forall u \in U: u_p(c)$. A global property $G(p)$ is the conjunction of p over all components. We will use letters $u, v, w \in U$ for users; letters $a, b, c \in S$ for components of the CSCW system S .

A component c may represent information not modelled by the user, but everything the user knows about it is represented in the component. Equivalently, the component knows everything the user thinks he knows about it *before* looking at it, and possibly modifying his model in the light of that observation. This is *observational consistency* for that component:

$$u_{consistent}(c) \equiv c \supseteq M_u$$

We now introduce the important notion of the *join*, notated \oplus . Two systems may be joined, typically by autonomous communications technology, but possibly by explicit user action, such as calling up from one component to another via a modem. In this paper, we take join to be symmetric and associative. As with other operations on components, we assume the user models, or is able to model, operations with join. This means not that the user has to be ‘clever,’ but that join should be appropriate to the users’ tasks, and a ‘natural’ operation.

Join is the way in which components of a system are be composed in order to realise the overall cooperative goals of the system; a very simple (and, for some CSCW applications, inadequate) composition is set union. For CSCW systems using fixed networks, join itself may be transparent; for mobile users, join is typically associated with a definite user-level act, such as dialling into another component of the system.

Following *full* observation of a component c , a user u therefore knows $c \oplus M_u$. After observing a component a user may or may not choose to replace M_u by $c \oplus M_u$; it may be preferable to modify c , alternatively an inconsistent system

becomes consistent if the user changes his mind about it, that is by performing the mental assignment $M_u := c \oplus M_u$. Any task performed by exactly one user on a component necessarily preserves observational consistency for that user, since the user's model is required to correspond to any updates to the component made by that user. Thus if a user's model is initially empty, consistency is trivially maintained for any particular component. A basic question is whether observational consistency is preserved following a join of any component with any other component.

In a CSCW system, by definition of join, for any two users u and v with components a and b , $(a \oplus u) \oplus (b \oplus v) = (u \oplus (v \oplus (a \oplus b))) = (u \oplus (v \oplus (a \oplus b)))$. This means that the results are the same whether u and v work separately, then cooperate, or whether they work in any order—possibly together—on their joined component $a \oplus b$. Many similar identities also follow: join's symmetry and associativity permit flexibility in the users' physical cooperation—which, of course, is essential for randomly mobile users.

Join need not be monotone. Whether, of two components respectively S - and T -consistent, their join is $(SUT)_{consistent}$ depends on what users (in S and T) choose to model; monotonicity occurs when users' models are *independent*. A model M_u is *independent* of M_v when any legitimate operation u performs, also modelled by v , on any component makes no change to M_v . For example, independence occurs when users only model their 'own' information that they are able to update (due to the security assumptions imposed); colloquially, as when users mind their own business. In this case it follows that when all users join the resulting component is globally consistent, furthermore that they may join in any permutation, and the joined systems form a Boolean lattice—joins can only get better.

We may define observational properties in terms of joins:

$$u_{sound}(c) \equiv c \oplus M_u \supseteq c$$

—*or*: After looking at a component c , there is nothing in the component that is not what the user then knows, $c \oplus M_u$. What the component represents is consistent with what the user models, but may not be complete. Alternatively, the component does not know something a user does not contain after looking at it.

$$u_{complete}(c) \equiv c \oplus M_u \subseteq c$$

—*or*: The component c may contain information not known to the user, but everything the user knows about it after looking at it, $c \oplus M_u$, is contained in the component. Alternatively, the component knows everything the user thinks he knows (about it) after looking at it.

A component is *compatible* if joining it to any component does not damage the observational consistency of the component for any other user:

$$compatible(c) \equiv \forall a, u: u_{consistent}(a) \Rightarrow u_{consistent}(a \oplus c)$$

Here we see a notion of grace. If a component is compatible but not consistent, it may be rendered consistent by updating it with its join with any consistent component. One would therefore wish to ensure that, even in the presence of communications failure, that a component remains compatible. Note that compatibility is not an observational property.

2.4 Task/observational property ¶

We say that a system has an observational property just when no operation can be performed by any user that breaks the corresponding observational property for any user in any component of the system. Thus a CSCW system is observationally consistent when no incompatible components may be constructed.

Observational consistency may not be achieved when the systems themselves infer consistency. For example: if the updating itself is an explicitly logged activity; if the system interacts with the world via a common representation (e.g., a program, or a financial account); if updates are time-stamped and order of processing is important (absolute time is a global measure for activities, so dependency on time is likely to compromise observational consistency). Although unsuitable tasks are easy to imagine, it is also the case that lack of observational consistency implies severe human management problems however systems are composed from their components.

A system need not be observationally consistent if it has an internal structure that can be compromised by another user (and there is more than one user). Example: a multi-author Pascal program. Tasks supported by systems of independent data records are generally observationally consistent.

Thus we see that observational properties are properties of particular classes of CSCW domain. The importance of this observation is that by judicious choice of domain and properties, a transparent, graceful and sound system can result. We may say that such a system is *practically perfect*: for practical purposes it cannot be distinguished by any user with respect to the relevant properties from a perfect system, with the corresponding global properties, under any operational circumstances. The observational properties are chosen to fit the task—and the task is chosen to fit the properties.

We have defined several observational properties. Are there relevant, non-trivial tasks that fit interesting properties?

3 A problem for practical CSCW

Users, particularly with access to more than one computer, find it hard to manage personal information, whether in filestores or backups. Groups of users experience the same problems, but with greater urgency: information management is more complex. User mobility, home working, group working, and the increasing popularity of laptop computers increases the number of places that users can work on copies, lose track of where current information is, and pick up virus infections that are then spread to all the other computers. With national and even world-wide databases the problems are inevitably greatly magnified. It is generally impossible to solve such wide-ranging problems without normative procedures (restricting access and other protocols). Users of personal information, without this administrative control, generally adopt ad hoc, unreliable version control and backup strategies—with inevitable consequences of lost and corrupted data. For personal, mobile users of conventional computer systems CSCW is a mess. Press (1992), using the term *collective dynabase*, describes some of the relevant technology.

We are concerned with defining appropriate forms of CSCW to address such problems.

4 Liveware

Our solution to the problems described above is termed *liveware*. Liveware is a method for reliable personal information sharing. It is a cheap and reliable technology, requiring exchangeable discs or better. It has applications in: distributed groupware; portable personal information, including smart cards and optical cards; mobile users, particularly with laptops; and for massive, even world-wide, databases of personal information. Liveware supports cooperative work in free social systems involving personal rather than institutional computers. Liveware has been described elsewhere (Thimbleby, 1990, 1991; Witten & Thimbleby, 1990; Witten, Thimbleby, Coulouris & Greenberg, 1991). The insight is essentially that, under certain assumptions, it does not matter ‘where’ information is, nor what state it is in when it is not ‘here’ or not ‘mine,’ and therefore there is nothing to manage—provided these assumptions can be maintained.

A liveware system (and any of its components) is a partial function $User \times ID \rightarrow (Data + \emptyset) \times Timestamp$. We say that a user *owns* the data in the image of that user; the ID can have cryptographic properties and we are entitled to call the pair $User \times ID$ a *signature*. The signature, data and timestamp together is an *object*. Here we assume no structure for the data; it may simply be textual data, but it could be ornamented in anyway, in particular the data may have serial numbers, graphics, relations, and further cryptographic properties depending on the actual liveware application. Updates are operations $User \times Data \rightarrow Data$, and any updates on a record update the timestamp; a user’s model may be the set of data in the image of that user. To permit deletion of objects, the data component is set to \emptyset . All timestamps for any user are different (and obviously increase with time): a user, by assumption, cannot do two things at once. This requirement need not assume synchronous distributed clocks.

The join \oplus of two components a and b results in a component c with domain $dom(a) \cup dom(b)$:

$$\forall i \in dom(c): \begin{cases} a(i).Timestamp \geq b(i).Timestamp \Rightarrow c(i) = a(i) \\ a(i).Timestamp \leq b(i).Timestamp \Rightarrow c(i) = b(i) \end{cases}$$

From all times for each user being distinct, it follows that \oplus is symmetric and associative as required. Typically c then replaces both a and b .

Liveware works because it is simple. It does not try to be a general solution, which would in any case allow people to work in complex ways that could in themselves cause further problems. Liveware imposes specific limitations on what it supports, but for personal users of information these restrictions are natural and unobtrusive. Liveware is sometimes criticised for not supporting the full generality of distributed systems operations; yet its restrictions stop such activities as users deleting or corrupting each other’s information. If the people using liveware are collaborating on a common goal these technical restrictions go unnoticed.

Liveware applies to tasks and environments provided:

- Either only one user is involved or users own information and can only be in one place at a time

- Information is subject to personal ownership and capabilities;
- Users' information travels at least as fast as they do;
- If a user has updated a component c , then c is joined with any other component before updating it;
- System consistency is observational.

Liveware is ideal for any personal information owned by a single individual: there are no practical restrictions. Single user liveware need not be treated as a special case; single users are a special case of multiuser CSCW (Anderson, Thimbleby & Witten, 1990; Cockburn & Thimbleby, 1991). A non-timestamped liveware system suitable for single users is described in (Coulouris & Thimbleby, 1993). For more than one individual, however, liveware is ideal for such applications as shared calendars, diaries, discussion groups: generally, for sharing all forms of personal information. Where the constraints of liveware, particularly the primacy of personal information, are appropriate to very large databases, the impossibility of continuous world-wide communications, perfect database replication and so forth are relatively inconsequential.

Liveware is unsuitable for tasks involving shared information that has an impersonal or centralised view of consistency, for example for developing computer programs, for airline reservations, for bank accounts, for inventory control. Liveware is also unsuitable for applications requiring real-time communications, such as emergency services; on the other hand, it is ideal for news networks. These points should be contrasted with networks, which assume communications infrastructure, such as network cabling. They require reliable connectivity, and often fail badly when networks are damaged or are disconnected.

Networked systems are ideal for supporting interperson communications (telephone or email), but they are not so obviously necessary for supporting the shared work of collaboration. If a multi-author document is being written, there have to be protocols to stop several people simultaneously editing the same section and causing chaos. Protocols can be socially intrusive or may result in obscure failures (e.g., when two or more people repeatedly edit a section and keep reinstating their favourite texts, or become irate when their contributions are tampered with). Liveware starts with the uncontentious notion of users owning what they contribute, this permits restriction of update permission to the owner alone, a protocol that is socially familiar and less intrusive than any technical protocol and, above all, cannot fail. A successful collaborative writing system has used liveware (Jones, 1992).

Liveware does not solve human problems! It is still possible to infringe copyright, to impersonate and so forth. The aim of liveware is to make trusted sharing reliable, including the automatic inhibition of accidental virus sharing; if a group of collaborating users have trust failure, they are going to have problems regardless of liveware. Liveware can be augmented with various forms of audit trails that may additionally help reveal compromises and responsible individuals. Any liveware system is much faster and more reliable than a manual system. The speed of performance of liveware is not as important as its ease of use or its robustness.

4.1 Problems with liveware

For completeness we mention two problems with liveware, one systemic, one circumstantial. Liveware systems require distributed garbage collection: current systems either ignore full garbage collection since the rate of memory loss from deleted data is insignificant (given that partial garbage recycling is possible) or they assume a system administrator who has authority to force collection of information deemed to be obsolete (as when a user leaves a system). The circumstantial problem is that we do not yet have any non-experimental liveware systems extant. Coulouris and Thimbleby (1993) provide HyperTalk code for a liveware framework. The intrinsic difficulty of implementing liveware is discussed in Thimbleby, Witten and Pullinger (1992).

5 Conclusions

It is natural, and sometimes proper, that CSCW systems are specified to be as powerful as possible. Since connectivity and communications bandwidth pose obvious technological limitations on CSCW, one may incorrectly assume the CSCW system is limited by the technological bottlenecks; this view results in a technology-driven field. Observational properties make it clear that a CSCW system can be specified in terms of what tasks its users do and what requirements they want satisfied.

Liveware is an example of a CSCW technology that supports observational consistency (under the relevant task-fit assumptions): it demonstrates that effectively perfect CSCW can be achieved in a practical fashion for mobile and personal users, or indeed for organisational users who want greater reliability.

Acknowledgements

Work on liveware started with Ian H. Witten, funded by an SERC Visiting Fellowship in 1990; he is now at Waikato University, New Zealand. Liveware has continued development with the present authors together with George Coulouris, Alan Dix, Saul Greenberg, Stephen Marsh and Ian Witten, and supported by funding from the Church of Scotland. The authors are very grateful for comments from Alan Dix, University of York, England.

References

- S. O. Anderson, H. W. Thimbleby & I. H. Witten, "Reflexive CSCW: Supporting Cooperative Long-Term Personal Work," *Interacting with Computers* **2**(3), pp330–336, 1990.
- A. G. Cockburn & H. W. Thimbleby, "A Reflexive Perspective of CSCW," *SIGCHI Bulletin*, 23(3), pp63–68, 1991.
- G. F. Coulouris & J. Dollimore, *Distributed Systems*, Addison-Wesley, 1988.
- G. F. Coulouris & H. W. Thimbleby, *HyperProgramming*, Addison-Wesley, 1993.
- S. Jones, "Designing Computer Systems to Support Collaborating Authors," *OZCHI'92, Interface Technology: Advancing Human-Computer Communica-*

tion, M. J. Rees & R. Iannella, editors, pp134–141, Ergonomics Society of Australia, 1992.

I. Press, “Collective Dynabases,” *Communications of the ACM*, **35**(6), pp26–32, 1992.

H. W. Thimbleby, “Liveware: A Personal Distributed CSCW,” IEE Colloquium, CSCW: computer supported co-operative work, IEE Digest No. 1990/133, pp6/1–6/4, 1990.

H. W. Thimbleby, “Sharing HyperCard Stacks,” Proceedings 7th Annual Apple European University Consortium Conference, pp68–71, 1991.

H. W. Thimbleby, I. H. Witten & D. J. Pullinger, Laws for Cooperative Artificial Life, Stirling University Computing Science Department Technical Report, 1992.

J. Turek, “The Many Faces of Consensus in Distributed Systems,” *IEEE Computer*, pp8–17, 1992.

FIX REF!

I. H. Witten & H. W. Thimbleby, “The Worm That Turned,” *Personal Computer World*, pp202–206, July 1990

I. H. Witten, H. W. Thimbleby, G. F. Coulouris & S. Greenberg, “A New Approach to Sharing Data in Social Networks,” in *Computer Supported Cooperative Work and Groupware*, pp211–222, S. Greenberg, editor, Academic Press, 1991.

Not cited

A. J. Dix, “Cooperation Without Communication: The Problems of Highly Distributed Working,”

A. J. Dix & R. Beale, “Information Requirements of Distributed Workers,”