MIT/LCS/TM-316

DATA SHARING IN GROUP WORK

IRENE GREIF
SUNIL SARIN

OCTOBER 1986

DATA SHARING IN GROUP WORK

Irene Greif
MIT Laboratory for Computer Science
Cambridge, MA 02139

Sunil Sarin
Computer Corporation of America
Cambridge, MA 02142

Abstract

Data sharing is fundamental to computer-supported cooperative work: people share information through explicit communication channels and through their coordinated use of shared databases. Database support tools are therefore critical to the effective implementation of software for group work. This paper surveys data sharing requirements for group work, highlighting new database technologies that are especially likely to affect our ability to build computer systems supporting group work.

1. Introduction

A central concern in computer-supported cooperative work (CSCW) is coordinated access to shared information. Database management system (DBMS) technology provides data modeling and transaction management well-suited to business data processing but not adequate for applications such as computer-aided design or software development. Group design applications, in particular, deal with structured sets of data objects -- design drawings, software modules -- and complex relationships among data, people, and schedules. Advanced programming languages do provide excellent abstraction facilities for these classes of data but with little support for data storage and sharing. Application programmers write their own data management functions in terms of files and operating system calls.

The objective of this paper is to examine the data management requirements of several cooperative work systems. These include flexible object-oriented models of data, maintenance of historical information, support for content- and role-based access control, new methods for dealing with concurrency, and techniques for information sharing among distributed autonomous systems. We will match these requirements with current research projects in database and programming technology such as "object-oriented database systems" and "persistent programming languages."

In Section 2 of this paper, we describe three cooperative work systems that we have built and their storage management strategies. Section 3 identifies common data management requirements, and Section 4 suggests supporting storage management mechanisms.

2. Three Cooperative Work Systems

The three examples described in this section are drawn from different application areas -- calendar management and document preparation -- and span

To appear in Proceedings of Conference on Computer-Supported Cooperative Work, December 3-5, 1986, Austin, Texas.

two modes of cooperative work -- real-time and asynchronous meetings. Because of this diversity, we feel confident that the requirements of these programs are representative of the basic support requirements of a wide range of cooperative work applications. Our conclusions are supported as well by reports of similar experiences in other research groups.

2.1. Calendar Management

MPCAL is a multi-person calendar system for meeting scheduling and resource management. Each calendar has a set of *role* definitions and a specification of which roles can be assumed by users. The system provides a default set of role definitions so that the owner does not have to describe all roles from scratch. These defaults reflect some reasonable expectation about how an individual might wish to control access to his calendar. The owner can modify the defaults and define new roles, to reflect different preferences or a different use of the calendar.

There are several types of calendars representing schedules of a person, of a common resource (such as a conference room), or of open events such as seminars. The interfaces to all calendars are similar, but they differ in their role definitions. For example, a conference room reserved by a "sign-up sheet" is built by using a "public" role that allows anyone to write a confirmed appointment, but which allows no conflicting appointments. Alternatively, a manager may have secretary, team and public roles that enforce the following:

- a "secretary" can confirm and cancel appointments on his behalf,
- members of the "public" can only make proposals for appointments; these appointments must be confirmed (or rejected) by either the manager or secretary,
- the public can see only blocks of time marked "BUSY,"
- members of the manager's "team" can see details of individual appointments.

Meetings can be scheduled through MPCAL by creating a "proposal" in a calendar. The proposal is written as an unconfirmed meeting into calendars of participants. (For each participating calendar, the caller must be able to assume a role that permits him to add proposals.) A participant receiving a proposal will be notified when he next looks at his calendar. The participant is expected to accept or reject the proposal (which sets the meeting as confirmed in his calendar, or deletes it), and his response is returned to the meeting caller's calendar. The responses -- accept, reject, hold, or no response -- are summarized for the caller who may check the status at any time. If the caller cancels or reschedules the meeting, the participants are again notified.

2.2. Real-Time Conferencing

RTCAL is a *real-time* conferencing system in which users share information from their personal calendars in order to schedule a future meeting. Participants speak to each other over a telephone connection and use the computer display as a shared blackboard. A "chairperson" invites participants and controls conference activity. The shared workspace of the conference includes a description of the meeting to be scheduled, and a filtered view of participants' calendars that shows only blocks of free and busy time. Individual participants can see more detailed information from their private calendars in a private window. Only one participant at a time, designated by the chairperson, can enter commands on the shared workspace for browsing in the calendar and proposing specific meeting times. Alternative proposals may be made: participants vote on the alternatives and can review the results. If one of the proposals is confirmed, it is permanently installed in participants' private calendars.

The displays of the private and shared workspaces are next to each other on the screen so that as participants browse in the shared space, their private calendar displays change to show the same time period. A participant may leave and rejoin the conference at any time and may update his private calendar while away from the conference. When he rejoins the conference, the participant's display is brought up-to-date with the other participants, and the shared workspace is updated to reflect changes made to the participant's private calendar.

2.3. Collaborative Document Editing

CES [21] is a Collaborative Editing System for a group of co-authors working asynchronously on a shared document. Storage of CES documents is determined by the co-authoring arrangements: primary authors of individual sections store their sections on their own workstations but a document outline is maintained at all sites. All co-authors can always get access to the shared outline of the document. Individual sections can be read by people who have appropriate access rights.

Authors may work independently on separate sections of the document. If two or more authors try to write in the same section at the same time, one of them will be granted a "lock" and the others will be informed of who holds the lock. The lock is obtained implicitly when an author starts editing and is held as long as some editing activity continues: the lock will be released to a new co-author after an idle period. These "tickle" locks are most useful in applications in which a user may forget to release a lock when he stops working. If a co-author tries to edit while the current lock-holder is idle, the lock will be released. In anticipation of this event, small editing changes are recorded on permanent storage on a regular basis while the document is locked; if the lock is released without the explicit consent of the author, his committed changes will still be incorporated in the document.

CES allows users to read text that is being modified. If someone else is writing a section, the reader's view will be refreshed periodically as the section is updated. While CES does not explicitly support real-time conferencing, that functionality can be approximated by multiple users viewing the same section and taking turns updating.

2.4. Implementation Notes

MPCAL and RTCAL were both implemented in CLU [13], a language in which users can define new data abstractions. CES was implemented in Argus [12], which supports distributed programming using atomic transactions and retains the data abstraction facilities of CLU. The ability to define application-specific abstract object types contributed to the rapid and reliable development of all three systems.

Interfacing with the file system for permanent storage had several problems. First, since calendar objects were stored in files, whole calendars had to be written to files in order to record even small changes to individual appointments. Second, unlike most data bases which make it easy to add new fields to records, CLU routines for mapping objects to files represent data in a fixed way that rules out later changes to the type. In anticipation of later data structure changes we translated from abstract objects into our own extensible record structures for disk storage. Finally, protection of files did not match with the access rights needed in our applications. In order to realize the desired behavior, it was necessary to leave files (e.g., MPCAL calendars) completely unprotected at the operating system level. This solution would not be acceptable for a production system involving storage of sensitive data in files since it leaves open the possibility that users could access or modify files by means other than the application program.

Concurrency control was dealt with in application code in each of our programs. In MPCAL we took an approach that maximizes concurrency at the risk of having to back out of actions. If two users accidentally enter appointments in the same time slot they will be notified of the conflict and may then choose to take compensating action such as rescheduling one or the other appointment.

Even though CES was built on a system that includes locking and atomic transactions (as do many database systems), it was necessary to implement additional facilities at the application level; the built in locks did not allow enough concurrency to users accessing multiple versions. (See [9] for a full discussion of the Argus implementation.)

3. Data Abstraction Requirements

The example applications have many common requirements for modeling and management of data, which we describe in this section. Some of these requirements are met in DBMSs, others in "object-oriented" programming languages and systems. Researchers in both areas are beginning to incorporate features from the other [14, 17]; these "object oriented database systems" (OODBS) are likely to be the choice for building CSCW applications of the future.

3.1. Types, Relationships, and Inheritance

Conventional DBMSs typically support large collections of objects of very regular structure, and provide efficient associative access to objects based on their properties and relationships. They are less useful for storage and retrieval of unstructured information such as chunks of text. Many group applications require a synthesis of such unstructured information with the more structured information supported by DBMS. Object class hierarchies offer an appropriate mechanism for integrating the two: less structured objects are members of classes close to the root of the hierarchy, while objects with more regular structure appear in object classes deeper in the hierarchy. The Information Lens system [15] is an example of a system that uses this principle to classify electronic messages.

The ability to define relationships among objects is important both within an application and to permit integration across applications. For example, MPCAL provided very little support (a text "comments" field) for unstructured discussion among the participants of a meeting. It would have been considerably more useful to be able to link calendar objects and mail messages, so as to take advantage of both the calendar tool and the mail systems when trying to schedule a meeting. Systems that support structured documents (such as Augment [6] and Intermedia [26]) typically provide one kind of "link" between objects and allow the users to supply their own interpretations. In DBMSs, on the other hand, relationships among objects are classified into disjoint types and users are forced to name the relationship type they wish to access. If relationships can be treated as objects and classified into a hierarchy of types, this would provide a means for integrating the two ideas. An example in the group work setting is the classification of link types in TEXTNET [25]. The general class of "Argument" links is used to relate a piece of text to a piece of supporting text. Depending on the nature of the supportive argument, an Argument link can be further classified as "A-deduction," "A-induction," "A-analogy," or "A-intuition." A user browsing a text network can then ask to see general support by following all Argument links, or one specific kind of reasoning by following, e.g. A-deduction links only.

3.2. Persistent and Transient Objects

Support for long-lived or "persistent" data (in a file system or DBMS) is typically very different from the support provided by a programming language for short-lived or "transient" data within a program's address space. This leads to problems in mapping data between the two, as described in Section 2.4. In addition, it is becoming clear that similar features are often needed for managing both kinds of data. For example, triggering reevaluation of derived data when any of the underlying data has changed is needed both in DBMSs and in systems displaying transient graphical views of data [18]. Similarly, copies of replicated data need to be coordinated in both long-lived distributed systems (such as the outlines in CES) and in real-time meeting support systems (such as RTCAL and Colab [23]).

The integration of persistent and transient objects is addressed in "persistent programming" projects such as [1]. The uniform treatment of both kinds of data will relieve the application programmer from most storage management coding.

3.3. Views

Most applications require access to information that is derived or computed from other information, e.g., the set of appointments falling on a given day in MPCAL. In DBMSs, these are referred to as *views*. By allowing users to treat views as full-fledged objects, the system can insulate them from the details of how the view is computed and whether or not it is stored to avoid repeated reevaluation. Current DBMS view definition capabilities do not include important functions such as recursive traversal of hierarchical structures and networks of links between objects; object-oriented database systems are beginning to incorporate such facilities [19].

3.4. Access Control

Our application programming experience has shown that constraints on group interactions are not easily expressed in terms of traditional file system access rights such as "read" and "write" for "owner", "group" and "public". Instead, more sophisticated controls are needed, which take into account factors such as the following:

- The operation being performed, which may be an abstract operation other than read or write.

- The "role" of the user performing the operation.

- The relationship between the user's identity (or role) and object properties whose values are user (or role) identifiers. For example, an MPCAL calendar may choose to allow only the user who created a meeting object (and the calendar owner) to modify or cancel the meeting.

- The contents of the database. In MPCAL, it is possible to prevent users within a given role to enter meetings that conflict with existing meetings.

MPCAL's role-based access control facilities included the above capabilities, but using an ad hoc mechanism. The ability to permit or deny access based on arbitrary database predicates is provided in some DBMSs; what is needed to meet the above requirements is the inclusion of roles and of abstract operations.

3.5. Meta-Data

In addition to representing and manipulating application data (e.g., objects and relationships), a system will also need to represent and manipulate various kinds of "meta-data" about the system itself. An example is a view description. If view descriptions were made accessible to users, then they could easily change the way they examine data, or change the ways that data will appear to others. In MPCAL we built an ad hoc roles definition facility with a special user interface. To the extent that meta-data can be represented using the same object model as application data, users will only need to learn a single interface for accessing all data.

4. Data Management Support

CSCW data management requirements can be met at many levels of system architecture. Some features requiring efficient storage techniques or high-speed performance must be built into the storage management system. Others that provide generic abstractions for tailoring to application needs, may be provided by data management tools that reside on top of a storage management system. Most of the requirements we discussed in the previous section can be met through high-level abstractions written in object-oriented programming languages and underlying OODBS. In this section we consider additional features that should be incorporated in the storage management system for an OODBS. We discuss them first for single logical "databases" without regard to physical distribution of data; Section 4.4 covers distributed systems consisting of multiple autonomous databases.

4.1. Versions

A DBMS is usually designed to represent current data; but past data is used regularly in group work for accountability, recovery and exploration of alternatives. Many design applications require explicit access to different "versions" or "configurations" of the same logical abstraction such as a manufactured part or a software product. Automatic storage of linear sequences of versions is offered in many file systems. More advanced version support systems (such as [11]) have extended this idea to include objects with complex substructure (some of which may be shared among versions), and multiple branching "alternatives" or "layers". User interface facilities for viewing "differences" between versions have also been developed [5, 3].

In group work it is often useful to record how an object was derived from versions of other objects. This derivation can be represented by links among the objects involved; the ability to link objects is a sufficient base for constructing version support tools. In addition, efficiency dictates that information common to multiple versions (of the "same" or "different" objects) not be duplicated in storage. An example system that provides this storage efficiency, and allows applications complete flexibility in constructing version relationships, is the storage manager for the EXODUS object-oriented database system [2].

In a multi-user system, it is important to record more than the content of object versions: contextual information such as who created an object version and when will allow object versions to be selected based on properties of group interaction. In many applications, it may be important to record not only the identity of the user but also the role that the user was playing when performing a given action (e.g., creating a given object version). Computer conferencing systems need to record what objects (or messages) a user has read. Similarly, MPCAL keeps track of whether the calendar owner has seen changes made by other users (e.g., a secretary) to a given appointment. Most file systems and DBMSs record a fixed set of such attributes. In groups where people's working relationships may change, it will be necessary to occasionally change the properties on which to track versions. The ability to define triggers on operations can provide this flexibility.

4.2. Triggers

A *trigger* specifies an action to be taken automatically when an event or condition is detected. (These are conceptually similar to the rules used in knowledge-based systems.) The event may be one of the following, or a combination:

- The passage of time, past a specified deadline.
- A predicate on the contents of the database becoming true.
- The invocation of a specified operation (which may be the reading of an object); the trigger may take into account the identity of the user performing the operation and his "role". (This form of trigger was supported by MPCAL.)

Triggers support user notifications, as well as automatic initiation of validation or compensation procedures. Ad hoc triggering mechanisms appear in many systems, usually built into the application code. For example, RTCAL will scroll participants' private windows when the shared workspace is scrolled, to keep the two in unison, and will update the shared workspace whenever a participant's private calendar is updated. MPCAL supported user-defined defined triggers of a limited form, for notifications and recording historical information. General-purpose trigger mechanisms have begun to appear in DBMSs such as POSTGRES [24] and Sybase Inc.'s DataServer^{TM}.

4.3. Updates and Concurrency

The conventional approach to ensuring database consistency in the face of concurrent access is to ensure that each transaction on its own preserves consistency, and that the effect of running multiple transactions must be the same as if they had been executed one at a time. The latter property is referred to as transaction *serializability*. Transaction serializability can be ensured using concurrency control methods such as locking and timestamps. An underlying premise is that transactions will be short and conflicts resolved quickly. If long user interactions are managed in the same way, they can impose severe limits on concurrency. For example, a user who holds locks during an editing session will prevent other users from accessing the same data. (This problem can be particularly severe if there is a crash or network partition in a distributed system such that it is impossible to determine whether a transaction committed or aborted and thereby release its locks.)

Several approaches have been developed to allow for more concurrency, usually at the cost of a larger number of transaction aborts. Most such approaches either weaken or completely eliminate the "read locks" set on the possibly large amount of data that a user is viewing. This can be done using "breakable" read locks [7] that inform the transaction whenever the data read has been changed. Or, read locks can be eliminated if the user agent remembers the version identifiers of data that was read and at the time of committing the user's changes, verifies that none of the data has changed. The Whiteboards system [4] uses this method for multiple transactions, extending across program invocations and system crashes, in effect providing "long-lived" read locks.

If transaction serializability is to be preserved, a transaction must be aborted if it is determined (by a notification or when checking version identifiers) that anything that the transaction read has changed. This can be a severe problem if, as in many design applications, it is desired to have "long transactions" that last for days or weeks. To support long transactions, it is necessary to reexamine the serializability requirement, which was developed for database systems in the interest of preserving integrity constraints. In many applications it is possible for users to tolerate seeing inconsistent data [11]; they may even want to share intermediate and possibly inconsistent results with their colleagues. Thus, it is necessary to support "composite" transactions that are long-lived and may involve multiple users. Such a transaction will be composed of short atomic transactions, which are still needed for preserving the structural invariants of stored data, but the composite transaction may not be atomic as a whole. If concurrent execution of such transactions leads to inconsistencies, users (or automatically triggered programs) can "compensate" for the inconsistency at their convenience. The methods described above (such as weak read locks and version-checking) can be important in knowing when compensation is needed, but the effect of a change in a transaction's read-set should not force the transaction to abort. In the object server described by [22], a transaction can set "notify" locks on an object that it reads. If the transaction receives a message indicating that the object has been changed, it can react in any way it chooses, e.g., it may warn the user of the change or reevaluate its actions. A similar mechanism is also used to implement triggers. In POSTGRES [24], a "T-lock" is set on all data referred to in a trigger condition. The lock notifies the DBMS to reevaluate the trigger condition if there is any change to the data.

In [20], we presented a generalization of these locks called *reservations*. A reservation associates an action with an object. The action is triggered whenever the object is accessed concurrently. For example, the action can be a warning message stating that someone else is working on the same data and suggesting that the user not update this data. Alternatively (or in addition), a reservation may lock out concurrent updates for a limited period of time only. The expiration time for a reservation might be fixed, or some mechanism for extending or renewing it (as in CES tickle locks) might be provided. Reservations open up a wider range of strategies by making it easy to trigger user-defined routines to avoid conflict.

4.4. Autonomy and Multiple Databases

DBMSs support sharing by providing the logical view of a single repository of information (which may be physically distributed). Mail and message-based systems assume autonomous storage in mailboxes and support transfer of data among these data bases. Group data sharing involves a combination of access to shared artifacts (designs, documents) and communication among autonomous data bases (private calendars, mailboxes). Group data management systems must accommodate both concepts and let the users control and modify the tradeoff between the two. Users in RTCAL and CES had autonomous control over their private databases, but the sharing permitted across databases was fixed by the respective programs. MPCAL provided more flexible control over autonomy and sharing, through its role definitions.

The federated architecture of Heimbigner and McLeod [10] supports cooperation among autonomous systems by allowing an individual database to dynamically determine the "schema" of the data held by other databases. Systems *negotiate* for permission to perform a given class of query or update operations on another system. Replication can be supported by permitting one system to maintain a copy of an object from another system. Naming conventions (usually some form of unique identifier) must be established to allow update messages to refer to replicated objects.

Coordinated update to multiple databases can be complicated by the fact that autonomous data bases may not all be accessible at the same time. MPCAL meetings could only be placed in a calendar if it was available for update at the time the caller created the meeting proposal. The system should provide background processing that can monitor availability and complete the full set of updates whenever possible. Gifford and Donahue's "persistent actions" [8] or the "suspense file" of Tandem's distributed database system [16] provide this functionality.

5. Conclusions

In summary, our experiences with several prototype systems for cooperative work reveal unsupported requirements for flexible data modeling consistent with those addressed in OODBs and object-oriented programming languages. Data management support can be provided at many different levels of a software architecture. Some capabilities such as memory sharing for efficient version management should be provided as part of the underlying storage management system. Others, such as role-specific views of object versions, may be provided in a "toolbox" for cooperative work applications. The implementation of cooperative work tools at this level will be simplified once the proper low-level, generic facilities are made available with acceptable performance. Object-oriented data base systems offer the potential for meeting these needs.

6. Acknowledgments

We would like to thank John Cimral and Robert Seliger, who designed and implemented MPCAL and CES, respectively, and Marvin Sirbu who helped to clarify the abstraction of roles.

7. References

1. Atkinson, M. P., and R. Morrison. "Procedures as Persistent Data Objects". *ACM Trans. on Programming Lang. and Systems 4*, 7 (Oct. 1985), 539-559.

2. Carey, M. J., D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and File Management in the EXODUS Extensible Database System. Proc. Int Conf. Very Large Data Bases, Aug., 1986, pp. 91-100.

3. Davison, J. W., and S. B. Zdonik. A Visual Interface for a Database with Version Management. Conference on Office Information Systems, ACM, Providence, RI, October, 1986.

4. Donahue, J., and J. Widom. "Whiteboards: A Graphical Database Tool". *ACM Trans. on Office Information Systems 4*, 1 (January 1986), 24-41.

5. Elliott, W. D., W. A. Potas, and A. van Dam. Computer Assisted Tracing of Text Evolution. Proc. Fall Joint Computing Conference, 1971, pp. 533-540.

6. Engelbart, D. C. Authorship Provisions in Augment. IEEE COMPCON Digest of Papers, IEEE, February, 1984.

7. Gifford, D. K. "Violet, an Experimental Decentralized System". *Computer Networks 5*, 6 (Dec. 1981), 423-433. Also available as Xerox PARC technical report CSL-79-12..

8. Gifford, D. K., and J. E. Donahue. Coordinating Independent Atomic Actions. IEEE COMPCON Digest of Papers, IEEE, February, 1985, pp. 92-94.

9. Greif, I., R. Seliger, and W. Weihl. Atomic Data Abstractions in a Distributed Collaborative Editing System. Proceedings of the ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida, January, 1986.

10. Heimbigner, D., and D. McLeod. "A Federated Architecture for Information Management". *ACM Trans. on Office Information Systems 3*, 3 (July 1985), 253-278.

11. Katz, R. H., M. Anwarrudin, and E. Chang. A Version Server for Computer-Aided-Design Data. Proc. 23rd ACM/IEEE Design Automation Conference, 1986, pp. 27-33.

12. Liskov, B., and R. Scheifler. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs". *ACM Trans. on Programming Lang. and Systems 5*, 3 (July 1983), 381-404.

13. Liskov. B., R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. CLU Reference Manual. Lecture Notes on Computer Science Volume 114, Springer-Verlag, 1981.

14. Maier, D., and J. Stein. Indexing in an Object-Oriented DBMS. Proc. Int. Workshop on Object-Oriented Database Systems, Sept., 1986, pp. 171-182.

15. Malone, T. W., K. R. Grant, and F. A. Turbak. The Information Lens: An Intelligent System for Information

Sharing in Organizations. Proc. CHI'86 Conf. Human Factors in Computing Systems, April, 1986, pp. 1-8.

16. Norman, A., and M. Anderton. EMPACT: A Distributed Database Application. Proc. National Computer Conference, 1983, pp. 203-217.

17. O'Brien, P., B. Bullis, and C. Schaffert. Persistent and Shared Objects in Trellis/Owl. Proc. Int. Workshop on Object-Oriented Database Systems, Sept., 1986, pp. 113-123.

18. Reiss, S. P. "PECAN: Program Development Systems that Support Multiple Views". *IEEE Trans. on Software Engineering SE-11*, 3 (March 1985), 276-285.

19. Rosenthal, A., S. Heiler, U. Dayal, and F. Manola. Traversal Recursion: A Practical Approach to Supporting Recursive Applications. Proc. SIGMOD Int. Conf. Management of Data, May, 1986, pp. 166-176.

20. Sarin, S., and I. Greif. "Computer-Based Real-Time Conferences". *IEEE Computer 18*, 10 (October 1985), 33-45. Special issue on Computer-Based Multimedia Communications.

21. Seliger, R. The Design and Implementation of a Distributed Program for Collaborative Editing. Master Th., Massachusetts Institute of Technology,Sept. 1985. Also available as Laboratory for Computer Science Technical Report TR-350..

22. Skarra, A. H., S. B. Zdonik, and S. P. Reiss. An Object Server for an Object-Oriented Database System. Proc. Int. Workshop on Object-Oriented Database Systems, Sept., 1986, pp. 196-204.

23. Stefik, M., G. Foster, S. Lanning, and D. Tatar. WYSIWIS Reconsidered: Early Experiences with Multi-User Interfaces. Proc. Conf. Computer-Supported Cooperative Work, Dec., 1986.

24. Stonebraker, M., and L. A. Rowe. The Design of POSTGRES. Proc. SIGMOD Int. Conf. Management of Data, May, 1986, pp. 340-355.

25. Trigg, R. H., and M. Weiser. "TEXTNET: A Network-Based Approach to Text Handling". *ACM Trans. on Office Information Systems 4*, 1 (January 1986), 1-23.

26. Yankelovich, N., N. Meyrowitz, and A. van Dam. "Reading and Writing the Electronic Book". *IEEE Computer 18*, 10 (October 1985), 15-30. Special issue on Computer-Based Multimedia Communications.