Contents

1 Introduction 3

2 Groupware Domain of Hoopertext 3
   2.1 Scenario of Group Information Access 4
   2.2 Domain Attributes 4

3 Hoopertext Overview 5
   3.1 Functionality Questions 5
   3.2 Hoopertext Architecture 6
      3.2.1 Hypertext Web 6
      3.2.2 Graphical User Interface 7
      3.2.3 Database Support 8
      3.2.4 Example – Replying to a message in Banyan 9
   3.3 Related Work 9

4 Architectural Implications 10
   4.1 Concurrency Control 11
      4.1.1 Lock Granularity and Access Rights 11
      4.1.2 Lock Visibility 12
      4.1.3 Contention Handling 12
      4.1.4 Alternatives To Locking 13
      4.1.5 Concurrency Control Summary 13
   4.2 Web Consistency Maintenance 14
      4.2.1 Decentralized Object-Oriented Strategy 15
      4.2.2 Global Planning of Changes 15
      4.2.3 Scope of updates 16
   4.3 Responding to Change Notifications 16
      4.3.1 Mechanics of Asynchronous Database Updates 16
5 Conclusions

6 Implications for Future Research

7 References
1 Introduction

People frequently work in groups, share information, and develop joint projects. Much of the information that we generate and use in our daily work is interrelated, both within an individual's private domain and across the overlapping domains of coworkers. Multi-user hypertext is appealing as a way of helping groups build and share an information network that explicitly captures these relationships. We believe that shared hypertext will better support the tasks for which we use information, such as co-authoring a document or managing a project's technical issues and decisions.

A general multi-user hypertext platform has been difficult to build, with many technical challenges. Several technologies - database, hypertext model, user interface, update model - must be combined to provide a useful, coherent interaction model. A platform intended for multiple applications presents special challenges, since many design and implementation decisions require a thorough understanding of the semantics of an intended application.

In 1989 the Human-Computer Interaction Department of HP Labs developed a prototype of a multi-user hypertext platform, called Hoopertext. Our goal was to provide individuals and groups with effective access to large amounts of complex, dynamic information. Our focus was on information organization and user interface strategies, with hypertext as an enabling technology. In this article, we discuss the multi-user aspects of the Hoopertext system. A more general discussion of Hoopertext can be found in [16].

We defined Hoopertext using a scenario-based analysis of the requirements of some sample applications that represented widely separated points in the application domain we intended to support [2]. We used the resulting Hoopertext platform to build Banyan, a shared electronic conferencing application [12]. Banyan tested a vertical slice of the the multi-user interactions, validating many decisions and exposing some problems.

Section 2 characterizes the domain of multi-user tasks we focus on. Section 3 introduces Hoopertext's architecture and functionality and lists user-level questions which have guided the platform design. Section 4 explores implications of the questions and tradeoffs among the design constraints. We analyze the design of the platform's concurrency control, consistency maintenance, and change notification. In each area, we discuss the interactions between the structure of the hypertext web, asynchronous operations, coordination issues, graphical user interfaces, intended application semantics, and the flexibility required of a platform.

2 Groupware Domain of Hoopertext

Hoopertext is designed for extended information management. It is a platform for multi-user applications by small groups of users connected by a high-bandwidth local area network. The target applications are those in which a group of users shares an interest in the same data, though they may be using the data for different tasks. The platform must help coordinate the flow of information so that each user has reasonable access to it.

Hoopertext's domain fits between the ends of the spectrum of coupling defined by Ellis and Gibbs [5]; it is tighter than the loosely-coupled multi-user applications in which users rarely

---

1The name Hoopertext came from Highly Object-Oriented Platform for Hypertext. One of the goals of the project was to explore the use of object-oriented technology to build a hypertext system.
interact, but it lacks the shared focus of real-time synchronous groupware applications such as electronic meeting rooms. In Johansen's classification of approaches to groupware [13], our platform is designed for applications involving primarily asynchronous communication within small groups of people, when they are between meetings. This includes applications in the areas of project management, calendar management, group authoring, and group memory.

2.1 Scenario of Group Information Access

As a concrete example, consider what is required to support a work group's design deliberations – meetings, electronic discussions, and a design document. A meeting is preceded by shared information (an agenda) and generates more information (meeting notes, a report). People ask questions, bring up new issues, and take on individual and group action items. These result in more data, issues, resolutions, and agenda for new decisions. The information and the relationships (links) continually evolve, forming a group memory. Without computer aid, the information flow and the decisions of an ongoing project are difficult to track and manage.

We envision a future office in which the diverse activities of the design team are supported by an electronic, shared, multi-purpose information web. The web includes categories for meetings, design notes, articles, trip reports and private information. Links can be created between related data items, such as agenda items and meeting notes or open issues and design notes. Using a conferencing application like Banyan, participants can continue a discussion by linking new messages into the web. The same information web helps in co-authoring a summary report. Authors can share a to-do list, annotate and modify a common draft, and be notified of changes. The system can even help maintain two alternate structures of a report that share some of their contents.

2.2 Domain Attributes

The above scenario and similar scenarios of extended group information management share a set of common user goals and task attributes. These are:

- The web contains multiple users' information, created through extended interactions and serving different goals over time.
- Different users have separate tasks, but they often share an interest in the same data.
- There are multiple, simultaneous, independent readers and writers.
- Since users have separate tasks, asynchronous communication with a small delay is acceptable. A user's edit operation should not be visible to others character-by-character. Note that synchronous communication may not even be possible in work groups distributed over separate time zones.
- Users need flexible strategies for searching and browsing.
- Operations frequently affect multiple items – either explicitly, or because one operation changes several nodes and links. Some modifications are short, but many (such as document edits) may last minutes or hours.
As will be shown in the next sections, these constraints, along with the original detailed scenarios from our sample applications, guided the design of Hoopertext. They helped resolve many basic questions of functionality and quickly identified areas where novel technical solutions were needed to support users' desired interactions with the system.

3 Hoopertext Overview

This section introduces the design of Hoopertext. First we list some functionality questions that arose from an analysis of our groupware domain and guided our design. We then present the architecture of Hoopertext and illustrate some of its functionality with an example of the system's actions for one operation in the Banyan conferencing application.

3.1 Functionality Questions

To provide appropriate support for user tasks in our target applications, we had to identify the “right” functionality for concurrency control, data updates, presentation updates, and task coordination. We list some questions here that impact the end users' view of the hypertext system and that show how the application domain of Hoopertext differs from traditional on-line transaction applications.

Concurrency Control

- What happens when two users want to change the same node? Is there a way to queue the requests or be notified when the node becomes free?
- Can one user view the current contents of a node while it is locked for editing by another?
- Can a user add annotations while a node’s contents are being changed?
- What if a user specifies a group operation, and one of the objects (a link, node or collection) in the group is locked?

Data Updates

- Are updates of locally cached objects handled immediately? Are notifications of remote updates handled immediately? Or are updates queued until a user needs to know about the change (lazy update strategy)?
- What happens when a user tries to access an out-of-date object?

Presentation Updates

- When will one user's presentation be changed to reflect database changes made by another user? Some changes may be computationally expensive, others distracting to the user.
- Should the user's attention be drawn to changed objects? How would the system track what is new and what has already been seen?
Do we need a separate policy for keeping a user's own views consistent? For example:

- If a user deletes some data, should it disappear from all the user's views immediately or disappear only when the change is committed to the database?
- If a user has two windows on a text node and decides to edit the node, should both windows be editable?
- Should two windows on an editable text node be kept up to date on a character-by-character basis?

Task Coordination

Assuming different users have separate focuses of attention, they will still have overlapping interest in nodes. Should a multi-user hypertext platform provide any mechanisms or conventions for:

- Registering intent to change or delete,
- Desire to be informed of changes,
- Request for coordination.

The answer to one of these questions is often "it depends on the application". We will examine the rationale and some of the hypertext platform's possible alternatives in Section 4. There, we will explore the interactions among decisions and the platform hooks that would be needed to support alternative choices.

3.2 Hoopertext Architecture

The major components of Hoopertext are the hypertext data model, the user interface and database support.

3.2.1 Hypertext Web

Our hypertext data model has three basic objects: nodes, links and collections. Nodes can vary in size: a node might be a document section, an electronic mail message, or an annotation. Since links express relationships among information, they are first-class objects, not just pointers. Their attributes include owner, type (e.g., reply, cross reference, citation), and two endpoints (each of which is a node and an optional offset into the character stream). Nodes and links are grouped into container objects, called collections. Collections are similar to fileboxes in NoteCards [11], except that links can also be members of collections.

Users may access information in the hypertext network through standard navigation techniques: following explicit links from one node to another, browsing, or selecting a node from overview graphs or indexes. We also provide database queries, since we believe they are an important capability for managing large hypertext networks.
We believe that the system should do routine consistency maintenance of the hypertext web (such as cleaning up dangling link references), so a user can concentrate on the task at hand. Therefore, the semantics of Hoopertext operations include consistency maintenance of the link, node and collection objects on the hypertext web. We also provide notification of changes to other users and to all the presentations of affected objects.

Hoopertext is an object-oriented platform implemented in a language with rich inheritance semantics. Nodes, links, collections and presentations may all be specialized by applications to provide customized semantics. For example, Banyan provides reply and cross-reference links with different behaviors and presentations.

3.2.2 Graphical User Interface

In the Hoopertext model, users view and manipulate the information web through a graphical user interface. Overviews can provide spatial or temporal context. Multiple windows can be used to present detailed views of more than one node or to provide high-level and detailed views simultaneously.

We believe that users should be made aware that all presenters are snapshots which may be out of date. Updates may be computationally expensive and distracting to the user if done automatically. They can be postponed until a time that is convenient to the user, but the user must be notified whenever the current graphical view is no longer accurate.

Figure 1 shows a sample screen in the Banyan conferencing application built on the Hoop-
ertext platform. The Banyan Main Viewer window shows a graphical view of a collection of the message nodes and links involved in conversations on two topics. The Topic: restaurants view is a linear browser of the collection of messages on one of those topics. Messages in either browser may be selected and viewed in their own windows. The small pop-up windows show previews about a message node in a graph or the other endpoint of a link.

3.2.3 Database Support

We provide sharing through a database that stores the hypertext information web. Each user runs a separate process, which communicates with the database server through interprocess communication over a local-area network (LAN). Each user process has an associated mailbox for queued messages about changes from other users. Each user process has a single control thread. Figure 2 shows a schematic of the processes and inter-process communication.

We ensure consistency through traditional concurrency control: giving one user exclusive editing rights (called a write lock) to a single object or a group of objects. We also use transactions when needed to guarantee atomicity: if a change requires a set of steps (such as changing two data structures), the steps will occur as an indivisible operation. We do not

---

Footnote: Multiple threads might simplify context switches between tasks from the user's point of view (for example, if each window were controlled by a different thread), but multiple threads would not resolve the synchronization issues we examine.
use read locks, since exclusive read access is incompatible with the needs of asynchronous groupware applications.

To give efficient access to many small-grain objects, we use private caches as snapshots for shared data, using the persistent object extension, PCLOS[17]. To maintain consistency, we recache objects when acquiring a write lock, or when a process is informed of a completed change to a locally cached object. Processes with cached objects are also informed when other processes acquire or release locks on those objects.

3.2.4 Example – Replying to a message in Banyan

Hoopertext provides standard hypertext operations, such as create node, link two nodes, and follow link. It also allows group operations, such as delete text and links in selected region and delete selected nodes and their links. Here is an example from Banyan to introduce the database and communication concepts that come into play.

Imagine that one of the users of the Banyan conferencing system wants to reply to the Message 39 shown in Figure 1. What is the functionality involved in this reply operation?

First, a new message node 60 and a reply link from Message 60 to Message 39 are created. Both the new link and node are made persistent in the database. While the link between the two messages is being created, both existing messages must be locked. In general, when an operation requires multiple locks, we avoid deadlocks by aborting rather than blocking if any of the operands is unavailable. The link creation occurs inside a transaction, so rollback can occur if the operation is aborted and changes to all the affected objects can be done as an atomic operation.

Once there is a new link and message, the empty Message 60 node must be edited to give it some contents. If an old object were being edited, the system would lock and recache the object first so the most recent copy would be in place.

After the edit, the user decides whether to abort or submit the message. If the message is submitted, then the transaction is completed, the database is updated, and other users’ processes are notified so that their views of the data can also be updated. Each of these processes has a message posted in its mailbox, indicating which objects changed and what type of change occurred.

Each process responds to this update notification by recaching any changed objects, creating cached instances of new objects, and notifying the local presenters that view the changed objects. In Figure 1 the detailed view of Message 39 on the screen would be updated to show the new link to Message 60, and the collection presenter (the Banyan Main Viewer) would be recalculated to show the new link and reply message.

3.3 Related Work

Other multi-user hypertext systems have dealt with many of the problems of concurrency control and notification that we encountered in Hoopertext. Most are specialized for a single application, or chose an application domain that allows a fixed interface or simpler concurrency controls. They provide examples of how technology limits functionality, and how a focus on a single domain can simplify technical requirements. Their experiences
and limitations influenced the requirements of our general platform for group information management.

Systems such as KMS (Knowledge Management System) [1] use fixed-size nodes, a fixed window layout and optimistic concurrency control to achieve excellent responsiveness. The one-page nodes are not locked for editing. If two users try to write changes to the same node, the second user is informed that someone else has already saved changes to that node and is allowed to map his changes into the new version. The design choices of KMS simplify concurrency and update problems, but they also reduce the user-visible context.

The Intermedia system [8, 15] is intended to be used in the instructional domain. It has a relational database back-end and a sophisticated user interface. Intermedia’s domain allows simple concurrency control and notification – the coursework web created by instructors does not change frequently, and students create independent webs of annotations. Intermedia’s recent extension, InterNote, supports group annotations and revisions to a document, but does not provide support to help groups coordinate their efforts [3].

The gIBIS (graphical Issue Based InformationSystems) [4] system is closest to Hoopertext in that it provides a shared database, concurrency control, graphical overviews, and notification. Its hierarchical node structure and fixed screen layout constrain the update problems. The system knows what each window is used for, so it appears that only three windows must be notified of other users’ changes. Unlike gIBIS, Hoopertext allows an arbitrary structure of nodes and links, with a flexible window organization.

Each of the above systems is a success in its domain, but each succeeded with some simplifying assumptions. By designing a multi-user platform for a range of applications, we exposed the difficult interactions among component technologies and the effects of users’ tasks on basic architectural decisions. Hoopertext’s successes and limitations in accomplishing its goal of providing a broad platform offer insights about the component technologies and illustrate the tradeoffs currently available.

Note that there are many interesting engineering tradeoffs and knowledge representation issues that are outside the scope of this paper. We only touch on access control, since it presents problems very similar to those of concurrency control. We also do not discuss database crash recovery or archiving. Hoopertext chose not to address non-text media in hypertext systems, synchronous groupware, and the scale and social problems arising from global hypertext environments such as those envisioned in NLS/Augment[7]. Either synchronous groupware or very large web sizes require different strategies and implementations than the ones we chose.

4 Architectural Implications

A hypertext system that allows groups of users to collectively manage an information network requires multi-user interaction. It was challenging to build a system which would do the “right” thing from the user’s perspective. The information web is a dynamic, shared resource. Each user may change the web, and the changes must be revealed to other users in a clear, unobtrusive way. There are times when access to an object must be denied because the object is busy. An event such as a presentation update or the denial of a lock request must conform to the user’s expectations of what is reasonable.

There were three factors that made it difficult to assemble a hypertext platform from stan-
standard technologies. First, we repeatedly found that user needs in near-real-time groupware ran counter to traditional database goals. We required long transactions, notification and coordination among users, and complex, manipulable views in the user interface. Traditional databases only support short transactions, applications which give an illusion of independence rather than collaboration, and single-threaded, query-based interfaces. Second, distinct application semantics dictated different solutions for different applications, complicating the platform definition. Third, in combining platform technologies we found that what seemed an appropriate design in one layer often conflicted with assumptions of another layer.

In this section we discuss the multi-user lessons learned in building Hoopertext. We focus on issues which arise when defining concurrency control, consistency maintenance and change notification.

4.1 Concurrency Control

In our scenario of a design team at work, multiple writers access the same agenda items, group calendar, meeting notes and follow-up items. Collisions are inevitable, and changes must be coordinated so the data do not become corrupted. In Banyan, for example, collisions occur whenever two users simultaneously compose messages on the same topic, since the topic node must be modified to contain a link to each of the new messages. We did not want to limit the Hoopertext platform to applications where users would manually manage potential collisions, or where nodes would be made artificially small. Although we planned for concurrency control from the beginning, it was difficult to do well.

Answers to questions like “what size data should be locked?”, “which operations require locking?”, “what should the lifetime of a lock be?”, and “what coordination support is needed?” must be guided by the user’s activities and the user’s model of the system. Concurrency control issues that affect usability are lock granularity and access rights, lock visibility, conflict resolution and deadlock-avoidance schemes, performance and programming methodology.

4.1.1 Lock Granularity and Access Rights

Lock granularity depends on the user’s goals. We believe that the operations of reading, editing and annotating a node have distinct access rights and concurrency needs. For example, one author should be able to add an annotation to a section as a reminder of changes to make, even though another author has the section locked. For this type of concurrency, the implementation would have to allow different parts of a node to be separately lockable.3 4

Why are we concerned about lock granularity? Fine-grained locking decreases contention at a cost of poorer performance and increased software complexity. Locking at the paragraph or sentence level must be done instantaneously and transparently, or users will get irritated and confused. However, this can be difficult in operations which require simultaneous locks.

---

3 Since the two changes to the node do not conflict, an exception to locking may be possible using a concurrency control algorithm like Ellis and Gibbs'. [6] This algorithm requires the ability to encode the semantics to resolve conflicting operations. Martin also discusses a promising model for semantic-based concurrency control [14].

4 Traditional page-level locking is clearly inappropriate - a single user editing a node with a dozen links might cause many pages to be locked for hours.
on multiple areas - such as cut-and-paste. We chose the simpler model, and locked entire nodes and links. We also locked a node's link location information whenever we locked the node, because link locations can change when the node's text changes.

4.1.2 Lock Visibility

Before initiating an operation, a user may want to know which objects must be locked by an operation, and whether they are free or locked. Let us examine these issues in turn.

In many operations several objects must change together. If a link is added between two nodes, both nodes must be locked, so their link tables can be updated. It is easy for a user to understand why both nodes must be locked, since both nodes were specified by the user as part of the create link operation. Other operations visibly change some objects, but the full set of affected objects is larger; there may be filtered objects (objects not currently of interest and therefore removed from view), related objects in the web, or hidden system objects. Colab [21] has shown that even in synchronous groupware, a visual indicator of what is locked can reduce contention by setting users' expectations. In our applications, users may have some nodes write-locked for hours. A busy signal on locked items would be a very useful device.

For our system, we needed two lengths of locks with different semantics: short locks for minor changes (such as changing permissions, adding a link or adding an element to a collection), and long locks for long content edits. Due to pragmatic issues, we could not implement both lock types, but we believe such facilities to be highly desirable. The planned semantics of short locks were to notify other users only when a change was completed. For long locks, other processes would be notified both when the lock was acquired and when it was released. Each receiving process could then give the user a busy signal for the locked objects. The system could also provide a request for lock message for the user to send to the holder of the lock.

4.1.3 Contention Handling

Even if the lock status of an object is visible, collisions will occur, particularly when a change involves multiple objects. When a user initiates an operation for which some object is busy, the system must do something reasonable. The three possible responses to contention are to (1) abort the operation, (2) block and retry after a short time, or (3) queue the part of the operation that requires the busy object, but complete changes on the available objects. The platform can decide which response to take, or it can allow the application to choose. In the latter case, sufficient information about the lock failure must be passed up to the application so it can decide whether to abort or retry. Sensibly

---

4For example: consider a copy-region operation in Banyan, where a region of text and links is to be copied from a message the user has just read (Message 1) to a new message being composed. Suppose that inside this region is a link to Message 2. The copy operation should create in the new message (the destination of the copy) a new link to Message 2. Therefore it must acquire locks on all three nodes. The user might have a filtered view of Message 1 and may not even know about the existence of this link or of Message 2.

6This improves user responsiveness, but can lead to data inconsistency. It requires good understanding of semantics to ensure that the queued subgoal will be executed correctly, or that the completed subgoals can be undone if the queued goal becomes infeasible.
informing the user of which action was taken is also essential. In Hoopertext, the default action is to abort the operation, but this behavior can be redefined by an application.

When multiple objects are involved, it can be confusing to either queue or abort the operation. From the user's point of view, it might be an unknown or seemingly unrelated object that is unavailable. Meeting the user's expectations is even harder from the platform perspective, since the platform or an application may use some shared system objects that are hidden from the end user.

Task-appropriate handling of lock failures was especially difficult in our multi-layer platform, because several modules must participate in unwinding and retrying partially completed operations and informing the user. In Hoopertext, we had default error handling routines that passed error information to presenters so they could construct user messages. We did not have control over the error handling strategies and information collected in lower layers. Our platform would have benefitted from a more systematic approach to exception handling.

4.1.4 Alternatives To Locking

The Colab project experimented with several control regimes for maintaining consistency in their environment. For our domain, a promising idea may be their roving locks model in which the lock-granting control for items is moved along with lock ownership. For users with a fixed working set of items, this can speed up lock access while maintaining safety. Other locking alternatives for asynchronous collaboration include tickle locks [9], which frequently save editing changes in a document, and release the lock to a new co-author after an idle period.8

We also considered a control strategy similar to that of the GROVE group editor [6] in which some changes (e.g., link manipulations, attribute edits) would not require a lock, but would be done immediately on the local machine. A request would be sent to an object manager which would queue the request and execute the change when the database object became available. Such object management was impractical given our available databases, but is a promising approach to reduce abort/retry cycles and improve user response.

4.1.5 Concurrency Control Summary

We learned two important lessons from implementing concurrency control in Hoopertext. First, we identified a set of critical database features for near-real-time applications, and areas where standard database goals conflict with groupware needs. Groupware applications require:

- Variable or small lock granularity,
- Dirty reads by multiple readers,
- Locks of different lengths,

7Transactions do not solve the entire undo problem, since they do not undo changes to transient objects (such as presenters).

8One might think that versioning could solve the concurrency problem — simultaneous edits could just create branching versions. Versioning is useful in CASE applications, in which users try out hypotheses or make revisions in parallel. However, when there's a single shared artifact, versioning simply moves the problem from concurrency control to "merge management". In addition, it complicates the semantic model of links — e.g. adding the question of "which version does this link point to — the latest version, or the version it was created with?".


- Deadlock detection,
- Selectable notification of changes to lock status,
- Queueing and merges of non-conflicting object modifications, where possible.

Second, and most importantly, it became clear that even within the domain of near-real-time shared hypertext - applications such as shared authoring, coordination, or CASE - different applications require different semantics in the areas of lock length and granularity and contention handling.

### 4.2 Web Consistency Maintenance

We feel that users should focus on their tasks, not on routine bookkeeping. In the design team scenario, if one person removes an item from the shared to-do list, any links from that item (to other to-do items, future meetings, etc.) should be deleted. Other users’ views of the to-do list should be updated as well. If different windows show different snapshots of the data, this can confuse users and introduce “mode” errors. This is especially true in groupware, where not all the changes are made by one user. Our philosophy was that Hoopertext should preserve data consistency in the network whenever possible, to reduce the amount of detail users have to manage.

Automatic consistency maintenance in a hypertext web is a noble goal, but also a messy problem. This is probably why many hypertext systems leave it to the user. Here we will show what is required to do consistency maintenance, and then discuss two alternative algorithms for consistency-maintenance. The two alternatives are: (1) a decentralized object-oriented strategy, and (2) global planning of changes.

Here are the updates that must be done for a simple destroy-node operation.

If Node A is destroyed, the semantic layer operations to clean up loose ends are:

- Destroy any links with at least one endpoint in Node A,
- Remove references to these links from the nodes which are their other endpoints,
- Remove references to Node A from any collections to which it belongs.

The presentation layer update messages are:

- Destroy presentations of Node A itself,
- Change presentations of nodes which are the other endpoints of links to or from Node A, to reflect the disappearance of these links,
- Change presentations of collections of which Node A was a member, to reflect the disappearance of the node.

If Node A has many connections in the web, this will prompt many updates. If Node A is cached and displayed in several users’ processes, each of these processes must be notified of the changes.

14
4.2.1 Decentralized Object-Oriented Strategy

An object-oriented strategy is a conceptually clean strategy for consistency maintenance. In an object-oriented strategy, each class (text node, link, aggregate) has methods which make the necessary changes for different kinds of change operations (such as add-link, move-link-anchor, destroy-self). The local changes may include updating a link table, preparing for deletion, or performing database operations such as locking or caching. Each method also fires off change messages to any related objects. The related objects manage their own changes and perhaps send new messages to their neighbors. In addition, each object sends a message to each of its presenters, telling it of the changes at the semantic layer so that it can redraw the presentation if necessary.

Such an object-oriented control flow is ideal for a platform. It has the advantage of modularity and easy extensibility: by subclassing a node, link or collection, an application can easily change the connectivity of the network or the effect of common operations. Object-oriented decomposition is natural, since it matches the problem structure.

The main problem of this decentralized strategy is the lack of coordination – multiple update requests may be sent to the presenters affected by an operation. For example, suppose that a node, its links and the links’ destination nodes all belong to the same collection. Then in a delete node operation, a collection presenter might get many update messages: one from the deleted node, one from each of its soon-to-be-destroyed links, and one from the other endpoint of each of these links. A presenter cannot filter out the earlier update messages while waiting for the last one to appear, because there is no way to know how many messages there will be. Responding to each message would not be reasonable – it would be very annoying to users to see the same presenter update itself several times during a single operation.

Note that the need for up-front locking and clean undo operations complicate object-oriented algorithms in the same way that presentation updates do. In a group operation there is no way to know in advance whether all the locks can be acquired, no simple way to know what to undo on an abort, and no way to coordinate change notifications from the different primary objects of the operation. We began with an object-oriented algorithm and then redesigned the system to follow a more procedural model of group change, which we call a global planning model.

4.2.2 Global Planning of Changes

To simplify the consistency maintenance strategy, we encode in each semantic operation the set of objects that need to be locked and updated. As an operation progresses, we assemble the messages destined for each neighbor object and each of their presenters, then send them all at once, which resolves the multiple update problems. The disadvantage to this approach is that it makes the design much more rigid, and it breaks platform/application encapsulation. Data modifications cannot be localized in the change methods of affected objects, but must be reflected in each high-level operation. In addition, application-level operations must use this same strategy, which makes it difficult for the platform to manage private objects.
4.2.3 Scope of updates

The consistency maintenance problem includes deciding which presenters and other processes to notify after an operation. Hoopertext allows applications to choose for each operation one of three broadcast distances to indicate who will be informed of the change. For example, suppose two people are co-authoring a report. While one of them is editing a chapter, the character-by-character changes are typically sent only to the current text presenter window. If one of the authors creates a new link while in the process of editing a chapter, all the relevant presenters belonging to that author are notified, but the other author will not find out about it until the edit has been completed. (Links might appear and disappear frequently during an edit, and notification on each change might distract the other author.) Whenever one author begins or ends an edit operation on an existing chapter, the other author will be notified. These three levels of broadcast — originating presenter, local process, remote processes — worked well for Banyan.

4.3 Responding to Change Notifications

The previous section described how consistency is maintained at the data level, and how messages are sent to the presenters and other processes that are affected by changes in the web. It is up to each presenter and remote process to decide how to respond to these change notifications. The goals of notification are to inform users at the right granularity about changes that affect their work, to update without unduly degrading response time, and to minimize surprises due to out-of-date information. This notification is best triggered by an active database, which can monitor changes to raw and derived data [18] [20]. The results of these triggers or “notify” locks are update messages to the cached objects in users’ processes.

The timing and granularity of visual updates depends on the coordination needs. Too-frequent updates of peripheral graphs or nodes are distracting, and they degrade response time in the user’s current task. Infrequent updates allow the user’s views to become out of date, frustrating a user who plans operations based on the old information. Consistency is not always desirable, although it simplifies database interaction. Users may want advance notice of changes by seeing data from incomplete long transactions. It would be very useful to experiment with alternative approaches to updates and concurrency such as discussed by Greif [9].

The update problem is exacerbated by the complexity of the user’s views in typical applications. The many windows and the graphical and web overviews that help users avoid the “lost in hyperspace” problem [19] present a great deal of contextual information. Any or all of this context may become out of date.

This section examines the mechanics of change notification and performing updates from the database onto local caches. It also explores the questions of when and how to show changes to the user.

4.3.1 Mechanics of Asynchronous Database Updates

When a remote process receives a change notification message, it must:

- Find out that there are change messages in its mailbox,
• Update the right objects in the local cache,
• Notify those presenters that show old data, so they can decide how to notify the user.

The user’s process can find out about changes in two ways. It can be told (with interrupts) or it can periodically ask (by polling).

The advantage of an interrupt is that it has no overhead when there are no messages. However, since some operations shouldn’t be interrupted, it may require additional complexity in the receiving process. Polling has a fixed overhead, has a delayed response to changes, and requires either a “busy-wait” loop or a timed wait even when there is no user input. Note that it is completely unreasonable to poll for messages only when a user event such as a keystroke or mouse movement is received. First, an idle system would not be able to inform the user that there have been changes. Second, the moment when the user begins an action is the worst time for the system to stop responding in order to process update messages. We prefer the efficiency of interrupts, but due to implementation restrictions we used polling, with a polling interval modifiable by each application. To avoid race conditions, we also check timestamps when recaching an object (for example, before placing a write lock).

The next question is how to identify which items must be updated in the local cache when multiple items are out of date. The answer depends on the user’s goals, the network’s connectivity, and where the system will check for inconsistencies. Updating only the primary modified object can lead to an inconsistent cache. Updating all changed objects may waste time on objects irrelevant to the user. We chose a middle path: we updated the primary modified object and the closure of all the objects that are related to it.

4.3.2 Showing Changes to the User

As Grudin points out in [10], user expectations may not be consistent or predictable. Users want to see relevant changes, but not while they’re in the midst of an action (like a search). Our goals are to prepare the user for unexpected changes, and to fail quickly and gently if the requested operation doesn’t make sense in the current state.

Confusion due to outdated information is especially likely with direct manipulation interfaces. Since the graphs, list browsers and link icons are manipulable, they have the illusion of being real and current. In a multi-window interface, the updates should not be limited to a single view, since the user can change focus simply by looking at a new window or panning a graph. The user may not even be conscious of a change of context. Our solution is to prepare the user for information becoming out of date by clearly presenting a snapshot model, showing an update flag or a timestamp for the window. Whenever possible, we update all windows which show relevant objects, so windows will be consistent with one another.

Failing quickly means checking the object early to make sure that the latest version is available – a user should not compose an annotation to a node, try to commit, and only then find out that the node has been deleted or that the user does not have permission to annotate. This was facilitated when we switched to the previously-discussed global planning strategy for database operations, where all locks are acquired at the beginning of an operation.
Failing gently means to make the right response to an request initiated because of insufficient information. Queue/abort/retry options have already been discussed in Section 4.1. There are also higher-level questions of what actions to take next and how much to update in order to explain to the user what the problem is. Questions include "Does the user still want to edit a node if it has changed?" and "If the user requests deleting three nodes and one is locked, should the other two be deleted?" As a platform provider, we chose not to enter this confusing area, and we did not try to support the complete range of failure options.

5 Conclusions

In this article we have identified some functional needs of shared hypertext-based applications for asynchronous group information management, and have shown the effect of user level issues on fundamental architectural decisions. We explored the relationships and conflicts among component technologies, especially concurrency control, database locking and access models, graphical user interfaces, and interaction models. Our experiences in building the Hoopertext shared hypertext platform and the Banyan computer conferencing application provided many insights:

- **Concurrency control.** Critical features for near-real-time applications include variable-grain locks; dirty reads by multiple readers; long and short lock lengths with different notification semantics; deadlock avoidance; notification of lock status changes; and queueing and merges of non-conflicting object modifications, based on semantic analysis. Locking alternatives such as tickle locks and active object managers are worth exploring.

- **Consistency maintenance and change notification.** With separate tasks and graphical overviews, users require coordination aids and notification of changes to shared data. We dealt with consistency and update problems, and succeeded in providing a flexible, direct manipulation user interface to a dynamic hypertext network. We designed Hoopertext to minimize inconsistent data, clearly present a snapshot model, and aid coordination by flagging locked and outdated data.

- **Object-oriented paradigm.** We found the object-oriented paradigm useful. It provided toolkits of specializable classes and a common protocol. However, the decentralized flow of control caused coordination problems in locking, undo operations and presentation updates of multiple objects. We suggest augmenting the paradigm with exception handling and a global coordination capability.

- **Task analysis.** A recurring theme has been that meeting users' needs requires understanding the task. Transaction lengths, lock granularity, contention handling, change notification, and consistency maintenance of the hypertext web are all strongly influenced by the semantics of the target application. Even within our application domain, these issues cannot be solved generically.

6 Implications for Future Research

Building our Hoopertext platform has shown two things important for future research. First, even within our niche of non-tightly coupled, shared, hypertext applications, the
multi-user issues require an understanding of the user tasks and a platform that is flexible in many dimensions. Second, decisions in the user interface, hypertext data model, database and update model all constrain each other and the user-level functionality. Thus, collaboration among computer sub-disciplines is essential to understand the interdependencies and develop novel solutions.

The field of multi-user hypertext-based applications is still in the exploratory phase. We need to build up a design space of group-information tasks and identify resulting concurrency, synchrony, performance, web size and communication needs of groupware applications. We should also identify the social and organizational issues that will affect the system's design and utility.

To make progress, we must better understand how to analyze application areas, and we must experiment with narrow-focus platforms. It is critical to analyze our artifacts in order to identify the user needs that mold the platform design and the application semantics that must be declaratively specified to platform layers. Hoopertext was one such experiment, and this paper attempts to provide a beginning framework for discussion.

Acknowledgements

Dennis Freeze, Pankaj Garg, Nancy Kendzierski, Allan Kuchinsky, Bruce Martin and Andreas Paepcke gave us many useful comments on early drafts of the paper. We thank our fellow Hoopertext, Banyan and platform developers: Mike Creech, Cathy Fletcher, Dennis Freeze, Warren Harris, Shari Jackson, Bob Leichner, Andreas Paepcke and Jarrett Rosenberg. Finally, we appreciate the insights and support for writing provided by Robin Jeffries.

7 References


