Proceedings

Twelfth International

DATA ENGINE

February 26–March 1, 1996
New Orleans, Louisiana

Sponsored by
IEEE Computer Society Technical

IEEE COMPUTER SOCIETY
50 YEARS OF SERVICE

Consistency and Performance of Concurrent Interactive Database Applications*

Konstantinos Stathatos† Stephen Kelley‡ Nick Roussopoulos†‡ John S. Baras§
Institute for Systems Research, University of Maryland, College Park, MD, 20742
{kostas,skelley,nick}@cs.umd.edu, baras@isr.umd.edu

Abstract

In many modern database applications, there is an emerging need for interactive environments where users directly manipulate the contents of the database. Graphical user interfaces (GUIs) display images of the database which must reflect a consistent up-to-date state of the data with minimum perceivable delay to the user. Moreover, the possibility of several applications concurrently displaying different views of the same database increases the overall system complexity. In this paper, we show how design, performance and concurrency issues can be addressed by adapting existing database techniques. We propose the use of suitable display schemas whose instances compose active views of the database, an extended client caching scheme which is expected to yield significant performance benefits and a locking mechanism that maintains consistency between the GUIs and the database.

1 Introduction

Fast networks and CPU abundance coupled with large main and secondary memories have been the primary reason for the nearly universal adoption of the client-server computing model. Virtually every DBMS employs this approach to offload processing from the server to client machines in order to increase the overall performance and scalability of the system.

This distribution of tasks has made possible a new generation of database applications. Very often these applications offer to the user a highly interactive environment through sophisticated *Graphical User Interfaces (GUIs)*. GUIs compose complex displays where database objects are graphically rendered in one or more ways, based on the desired representation of the objects (e.g. a graph versus a tabular format) and the choice of interface drawing components used to realize it within each application. Users access or update database objects by interacting with (e.g. pointing and clicking on) their graphical representations. This increasing use of graphical user interfaces has posed several challenges to database system as well as applications design.

An example of such an application, which was the motivation for this work, is an advanced network management system (NMS) that relies on an object-oriented DBMS for storing and managing all the necessary data [1]. This application builds graphical displays that represent the current state of part of the managed network. Through this display, the network operator can perform a number of management functions (e.g. monitor network activity, change configuration parameters) in order to ensure the expected network operation. To a great extent, the decisions of the operator depend on the display which must agree as much as possible with the real network state.

Designers and developers of interactive applications put a lot of effort in building user interfaces appealing to the users both in terms of appearance and functionality. The specific goals they set include a system that [2]:

- accurately reflects the database contents on the screen, and
- responds to user actions without lengthy and unpredictable delays.

The importance of the first goal lies on the fact that users take decisions and act based on the perception of the database offered by the interface. Therefore, it is imperative that the graphical elements composing the GUI are consistent with the objects stored in the database at all times. This requirement is not trivial, especially in a multi-user database environment where more than one user may concurrently access and update the database through varied and complex views. Then, the response time of the interface affects significantly the level of users' acceptance and satisfaction of a system. Generally, people have a positive reaction to fast predictable responses, but they easily get annoyed or frustrated from unexpected delays.

These factors present several challenges to the design of both the database and the applications, as well as the architecture of the DBMS itself. In the following section, we elaborate on these challenges by addressing problems pertinent to database design, performance and consistency for interactive database applications. Then, in section 3, we propose extensions to existing database techniques, such as caching and locking, that efficiently address these problems. In section 4 we briefly describe our experience from

*This material is based upon work supported by the National Science Foundation under Grants No. EEC 9402384 and No. ASC 9318183, by NASA under Grants No. NAGW-277S and NAG 5-2926, by the Maryland Industrial Partnership, University of Maryland under Grant No. MIPS 1122.11, and by ARPA under Grant No. F30602-93-C-0177

†Also with the Department of Computer Science

‡Also with the Institute for Advanced Computer Studies

§Also with the Department of Electrical Engineering

the development of network management application over a commercial OODBMS. Finally, in section 5, we present related work by other researchers before we conclude in section 6.

2 User interface challenges

Our previous experience in developing interactive database applications revealed some important problems. First, database design for such applications can be extremely complex. Then, database systems are tuned for efficient query processing but ignore user interface performance requirements. Last, in existing systems there is no provision for maintaining consistency between the database and the GUI.

2.1 Database design

User interfaces contain several graphical elements that are aimed to be an intuitive representation of real world entities. These elements offer some view of the real world tailored to the desired functional requirements. In database applications, each such element is associated to one or more database objects. Its appearance depends first on attribute values of the associated database object(s), and second on the user interface context. Consider, for example, a network management application that displays a graph representing the nodes and links of a real communication network. Assume that in the database schema, a class Link is defined whose Utilization attribute holds the current utilization of a link. On the user's monitor, a link would appear as a line connecting two nodes (boxes, circles, icons etc). Depending on the interface context, there are several options for displaying its utilization. Two possible examples are color coding (e.g. red, pink and white lines could represent links with high, moderate and low utilization respectively) and width coding (the line width is proportional to the link utilization).

It can be argued that the database schema should be designed in a way that incorporates user interface functionality. In other words, classes should include attributes and methods that allow object display and manipulation through a graphical user interface. However, we argue that there are several points that make this approach impractical:

Complexity Modelling real world entities through a collection of data structures is often a very difficult task by itself. Extreme caution must be taken so that the final schema design is an adequately accurate and efficient model. Obviously, any additional design requirements imposed by the user interface, makes this process far more complicated. It would prescribe the introduction of additional attributes (e.g. screen coordinates), methods (e.g. for drawing objects, clicking on objects etc) and possibly hierarchies (e.g. all drawable objects should be derived from a common base class).

Multiple perspectives Depending on the function that a user performs, it may be desirable to have several visual representations (perspectives) of the same object (e.g. color–coded and width–coded link utilization) or different "views" of the same database (e.g. tabular or graph representation). Each one would require a different set of attributes (e.g. screen coordinates, color, width, etc). This may incur a significant storage space overhead and potentially degrade performance. Moreover, user interface methods can become extremely complex to implement.

Multiple users A database schema that incorporates user interface specific attributes (e.g. screen coordinates) cannot accommodate different user or application preferences about the way data are displayed. Consider, for example, a scenario where multiple users are working on a common subset of the database objects. Even if all these users select the same visual representation, it is conceivable that they may prefer different screen layout (i.e. different object arrangement on the screen) for the specific function they perform. But this would not be possible since the attributes of persistent objects must have the same value system–wide, no matter how many physical copies of the objects exist.

Orthogonal design Most existing database systems do not efficiently support schema evolution. In many cases, a modification in the structure of the data is very expensive since it may involve off–line reloading of the entire database. But it is possible that new application and/or user interfaces are designed and developed or existing ones are modified long after the database. Obviously, it is practically impossible for the database schema to have provision for every possible user interface. Therefore, the database design should be orthogonal to user interface design, and focus on capturing the important aspects of the real world, ignoring any design requirements from user interfaces. Ideally, no modifications to the database structure should be required in order to create new user interfaces.

2.2 Performance considerations for GUI based applications

One of the main concerns of application developers is that users are very sensitive to the response time of the system. Lengthy response times are usually detrimental to productivity, increasing user error rates and decreasing satisfaction [2]. Also, users tend to establish expectations of the time required to complete a given task based on past experiences. Unexpected delays usually trouble or frustrate the users. Therefore, high variability in the response time of the user interface should be prevented. Hence, building a GUI that displays large amounts of information stored and managed by a DBMS can be very challenging. Many performance pitfalls may exist, since the response to a user action may require extensive data processing, a number network message exchanges, and several data retrievals from secondary storage.

The majority of existing object–oriented database systems are based on the client–server model, mainly in order to take advantage of the ample resources of modern workstations. The role of servers is limited to serving clients' requests for data (e.g. individual ob-

jects, object clusters or disk pages) while maintaining the integrity of data across the system. The main part of the processing load is distributed among powerful clients. This approach yields better overall system throughput and, consequently, wider scalability margins.

The degree up to which client processing can be exploited depends on the coupling between the clients and the server. The more a client depends on a server to perform a task the less it can use its local resources. A widely used method for minimizing this coupling is *client data caching* [3]. This method allows data to be located close to where they are needed, reducing the client-server communication overhead. As a result, there is a big performance benefit from reduced transaction latency and server workload.

Client data caching appears as the best approach to deal with the performance problems of the user interface too. Database objects cached in client's main memory can be directly used for user interface manipulations. This can reduce secondary storage accesses and client-server communication overhead. However, data caching as has been implemented in current systems does not completely address the user interface requirements, mainly for two reasons:

- It is possible that only few of the database object attributes are required in order to build a display. In the link example we mentioned earlier, a Link object may contain a large number of attributes that characterize the actual link. But for display purposes, only the end points of the link and its utilization are necessary. With traditional caching techniques, database objects are cached as a whole. Therefore, it is possible that a large part of the client memory is wasted for storing data completely useless to the user interface.
- Usually the applications have no explicit control on the clients' cache. Although they can cause data to come into the local memory, they cannot "pin" data there either due to space limitations or concurrency control considerations. The DBMS architecture and parameters (e.g. buffer replacement policy, buffer size, object clustering) as well as the system-wide workload affect the contents of the cache. For example, the buffer manager may drop an object out of the buffer in order to free memory space or simply because its copy has become invalid. As a result, a simple user action such as zooming or panning that involves that object maybe unexpectedly delayed until it is brought back into the buffer.

Hereafter, we refer to this form of caching as client *database caching* in order to emphasize its dependency to the database system.

2.3 User interface consistency

A non-trivial task for the graphical user interfaces of database applications is presenting a consistent and up-to-date view of the database. This task is even more more difficult in a multi-user environment where different users may view and possibly update the same database objects. Obviously, some sort of display synchronization mechanism is required which preserves the consistency of the user interfaces, under the performance requirements mentioned above. Generally, the straightforward approach of periodically refreshing the user interfaces is not considered acceptable, since it may cause excessive overhead.

From the database perspective, the *display consistency* problem is not much different from the client cache coherency problem. GUIs retain graphical representations of database objects much like caches keep copies of these objects. Therefore, the consistency requirements imposed upon the database system by user interfaces are similar to the those of client caches.

DBMSs preserve client cache coherency as well as transaction semantics by enforcing some kind of concurrency control protocol. Transactions that read and/or update data must satisfy the *ACID properties* [4]. Among those, isolation is usually guaranteed by a data locking mechanism. Under such mechanism, a transaction must obtain *exclusive (write) locks* for data it wants to update, and *shared (read) locks* for reading data. An exclusive lock can be granted to a transaction only if no other lock of any kind has been granted to any other transaction.

Displaying some database objects can be considered a kind of long transaction, a *display transaction*, which spans the lifetime of the display. However, traditional transaction semantics cannot be used to preserve GUI consistency as too restrictive. User interfaces cannot hold shared locks on objects being displayed, since that would prevent any updates to them or, at best, it would require all but one client to remove (erase) their renderings of the database objects to be updated, at least until the updates are committed to the database. In other words, these display transactions cannot be isolated. In section 3.3 we describe how data locking can be extended to handle this looser requirement.

3 Proposed database extensions

The user interface challenges presented so far have not been systematically addressed by database researchers. In the following subsections, we propose such a systematic approach which extends and combines techniques previously employed in different contexts.

3.1 Display schema

Through GUIs, applications try to offer an environment where users can carry out their tasks with minimal effort. Thus, depending on the specific tasks performed as well as personal preferences, interfaces may present several views of the same database. This way, users may perceive a number of different data organizations (schemas) which may be quite different from the actual database schema.

In section 2.1 we argued that the database design should not be affected by the user interface. As an alternative, we propose that, for each interactive application, a proper external *display schema* should be defined over the existing database schema. Display schemas are composed of *display classes (DCs)*

that encapsulate the desired user interface functionality and form inheritance and/or containment hierarchies that better meet GUI requirements (e.g. for screen layout computation, for screen navigation etc) both in terms of implementation effort and runtime efficiency.

The definition of a DC depends on the database class(es) it represents as well as the user interface context. It should include only attributes and methods that are necessary for the display and manipulation of the corresponding user interface elements. These attributes may be a subset of the database class(es) attributes as well as additional GUI specific attributes (e.g. screen coordinates).

The graphical elements that compose the image displayed by a GUI must be instances of display classes, i.e. *display objects (DOs)*. Display objects are created by copying and/or computing the necessary information from database objects. During their lifetime, they are explicitly associated and kept consistent with those database objects[1]. This association turns the collection of display objects into an active (updatable) view of the database as opposed to a passive snapshot.

For our network management example, the ColorCodedLink and the WidthCodedLink display classes can be defined, as in figure 1. Both classes have attributes for screen coordinates whose values are computed by graph layout algorithms, as well as methods for drawing the graphical elements (i.e. lines). In addition, the ColorCodedLink class has a Color attribute and the WidthCodedLink class a Width attribute. The value of these attributes is determined by the Utilization attribute of the associated Link database object.

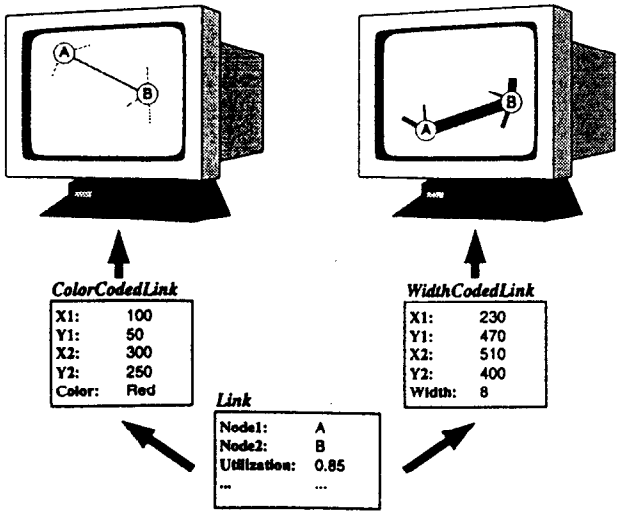

Figure 1: Display Classes Example

DCs can also be defined in order to combine multiple database objects into a single graphical element or, reversely, represent a single database object with multiple graphical elements. For example, the path between two nodes in a communication network may be represented by a line connecting the two nodes, without showing the actual links in the path. The graphical element for that line can be a display object that is associated with all the Link database objects of the path. Its utilization (i.e., color, width etc) would depend on the utilization of all these database objects (e.g. maximum or average utilization).

3.2 Display caching

The various memory spaces found within a client-server database system form a memory hierarchy according to the data access latency associated with each of them [5]. Usually, it is a three level hierarchy consisting of the server's disk, the server's main memory and the clients' main memory[2]. Client requests force data to be copied from a lower to an upper level in the hierarchy in order to reduce the latency of future accesses to the same data.

In principal, client database caching can enhance the user interface performance. However, in section 2.2 we argued that it has two important drawbacks. As an alternative, we propose a double client caching scheme, with the introduction of client *display cache* as the new topmost level in the memory hierarchy (figure 2).

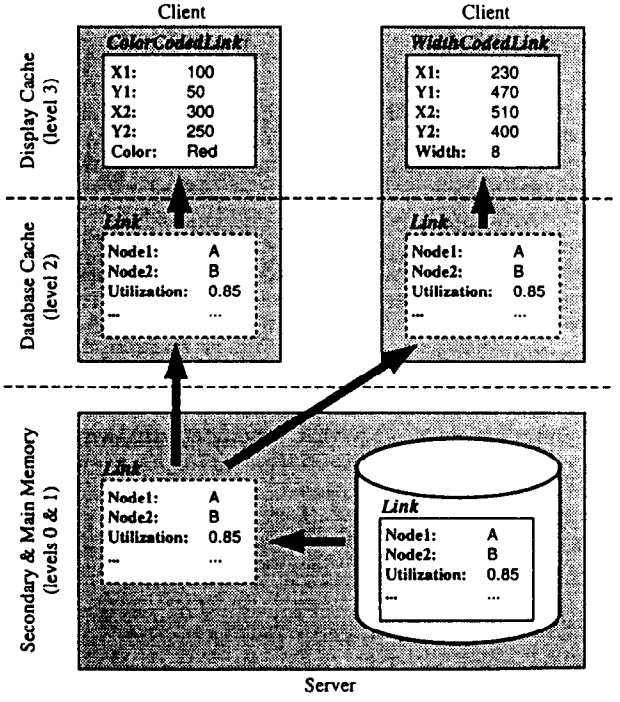

Figure 2: Extended Client–Server Memory Hierarchy

Unlike the other memory levels, the display cache holds display objects. They are created by reading database objects that are brought into the client database cache. This double caching may initially appear as an unnecessary overhead and waste of memory space. However, it has two major performance advantages:

[1] Each display object keeps an OID list for all its associated database objects

[2] A client's local disk has occasionally been considered as an extra intermediate level of the hierarchy

- As we mentioned earlier, one of the functions of display objects is filtering out database information that is irrelevant to the GUI. Using this approach, the display cache space can be managed in an optimal way from the GUI perspective. Once display objects are created, they are retained for as long as they are displayed. The associated database objects are no longer of any use to the GUI and, eventually, they will be dropped out of the database cache[3]. This way, the information overlap between the two client caches is reduced. The database cache is released from the GUI data requirements and can be more effectively used for answering database queries. In cases where large database objects are associated with relatively small display objects, the double caching scheme could actually save instead of wasting client memory.
- It is explicitly managed by the application which gives the GUI the flexibility to "pin" data in local main memory according to its own performance requirements. The contents of the display cache are affected neither by database system parameters and policies nor by system workload and concurrency control considerations. This is very crucial for avoiding long and unpredictable user interface responses.

3.3 Display locks

From what we described so far, we can think of the image presented to a user by the GUI as an accurate visual reflection of its display cache. In this way, the problem of keeping application interfaces synchronized and consistent with the database turns into a client cache coherency problem. The only difference has to do with transaction correctness criteria since display transactions cannot be isolated.

In client–server DBMSs, the server is usually responsible for maintaining data consistency through the enforcement of a concurrency control protocol. It must make sure that all user accessible copies of the same data in any level of the memory hierarchy are consistent, so that a user never accesses stale information. Generally, there are two major classes of protocols: *detection–based* and *avoidance–based* protocols [6]. Detection–based protocols allow stale data to reside in a client's main memory but require that transactions validate any cached data before they commit. On the other hand, under avoidance–based schemes, cached data are guaranteed to be valid at any time. This is achieved by employing the read–one/write–all (ROWA) replica management paradigm. Locally cached data are considered read–locked (and therefore valid) across transaction boundaries, unless instructed differently by the server. As a result, no explicit communication with the server is required for reading cached data. For updates, the server is responsible for "calling back" data that some client either intends to update or has already updated.

Detection–based protocols, which allow stale copies of data to reside in the client's cache, are not suitable for display objects. Moreover, within the display's lifetime there are no transaction boundaries, thus there are no clear points when data consistency should be validated. The user interface, therefore, needs to be somehow notified on relevant data updates so that any necessary action can be taken (i.e. redraw the updated part of the display). This makes avoidance–based protocols more appropriate since, under such a scheme, data validation is initiated by the server whenever necessary.

However, these protocols are mostly designed to enforce strict transaction correctness. For the relaxed correctness requirements of display transactions we propose a non–restrictive form of shared locks, called *display locks*. They are non–restrictive in the sense that display locked database objects can be updated, provided that at any time all lock holders get notified about the updates committed to the database.

The display locking protocol is quite simple and can be easily integrated with a strict avoidance–based protocol. A client requests display locks for all database objects that are associated with display objects. The database lock manager on the server is expected to grant those locks, since display locks are compatible with all types of locks. When a transaction wants to update some data, it does so after obtaining an exclusive lock for that data. After the update is committed to the database, the lock manager releases the exclusive locks and notifies all clients that hold display locks on the updated data. The notified clients refresh the associated display objects (and therefore the display) by reading the new data from the database. We call this protocol *post–commit notify protocol*.

A variation of this protocol is the *early notify protocol*, more suitable for long transactions. In this case, user interfaces are notified about update intentions as well. When a client requests an exclusive lock for a database object, the lock manager sends notifications to displays holding display locks on that object. The displays could then graphically mark (e.g. turn red) the object being updated, deterring users from modifying objects already being updated. As a result update conflicts and therefore transaction aborts can be significantly decreased. After the exclusive lock is released, the lock manager sends again messages informing whether the update transaction committed so that the affected clients can take appropriate actions. We must note that a server can easily support both protocol variations. Also, in every case the database consistency is ultimately guaranteed by the existing concurrency control algorithm.

4 Implementation

In order to demonstrate the concepts presented and investigate any potential implementation problems, we designed and implemented a multiple user, limited functionality version of a network configuration management application [7]. This application employs two different visualization techniques, the *Tree–Map* [8] and the *PDQ Tree–browser* [9], to display complex hardware hierarchies. ObjectStore [10], a commercial object–oriented database system, was used to store the network database.

The implementation included three major tasks:

[3] Assuming LRU buffer replacement policy

1. Extend the database server with display locking capabilities,
2. Enhance the client applications structural design to incorporate the display locking mechanism, and
3. Design the user interface in terms of defining appropriate display classes for the tree–map and PDQ tree–browser.

The overall system architecture is presented in figure 3. In the following subsections we will discuss in more detail each of the three tasks and explain the various modules of the figure.

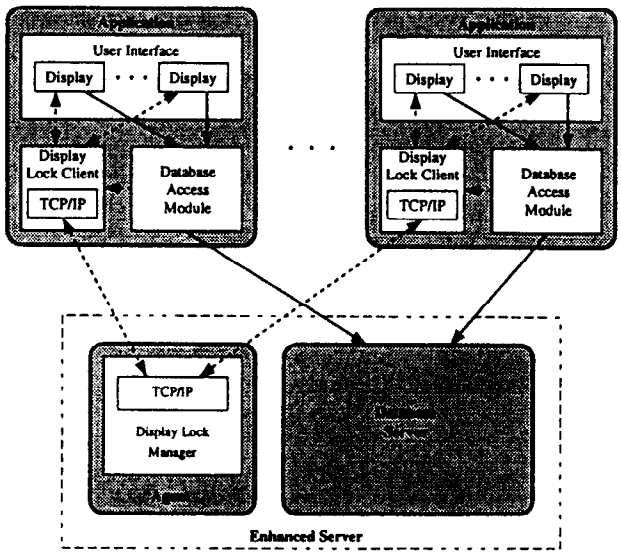

Figure 3: Implementation Architecture

4.1 Extending the server

Extending the database server with display locking capabilities is straightforward, assuming that the server already implements an avoidance–based protocol for client cache consistency. The lock manager is be the only module that should be modified. The required modifications are simple extensions, since it has already built–in most of the required structures and functionality[4].

However, using a commercial system prohibited us from directly modifying the existing lock manager. The desired functionality had to be implemented on top of the existing server, at the application level. Therefore, we built the *Display Lock Manager (DLM)* as a separate application, acting as an agent to the server. This approach is possible because display locks are compatible with all types of locks and there is no need for the DLM to interact with the existing lock manager. The obvious drawback is that most of the lock management functions are replicated at the agent, but on the other hand, there is a potential performance benefit by relieving the database server from additional overhead[5].

[4] For more details look chapter 8 in [4]

[5] The database server and display lock manager may run on different machines

The DLM has two–way communication capability with the clients. It receives messages for holding or releasing display locks as well as update notifications and propagates notifications to clients as necessary. Display lock requests are not acknowledged back to the clients since they are expected to be satisfied. Internally, it maintains information about the clients' display locks. This information is updated with any display lock request or release and determines which clients (if any) should be notified upon updates.

4.2 Building client applications

Creating client applications within the proposed framework requires the incorporation of the display locking mechanism as well as the design of appropriate display classes.

4.2.1 Display lock client

Early in the implementation process, we made the observation that a single client application often uses multiple displays (windows) concurrently. It is also possible that such displays may share some database objects, in which case the same consistency problem arises. One solution to this is to consider the different displays of an application as different clients for display locking purposes. Each display would contact the agent separately and the agent could send update notifications directly to each display. However, this solution would add some extra overhead to the agent in terms of communication, processing and memory requirements.

A better solution is to use a hierarchical approach that distributes part of the display locking responsibility to the clients. For this purpose, each client should include a *Display Lock Client (DLC)* module. The DLC functionality and internal structure is almost identical to the that of the DLM. This concept is similar to that of having a single buffer manager per client. The benefit is two–fold:

- The DLC can take over all communication with the DLM. Display lock requests and releases as well as update notifications can be sent through the DLC. Also, this module can be the "listener" for DLM notifications, relieving user interface development from network programming details.
- The DLC can be a local display lock manager that manages, filters and dispatches local display lock requests and update notifications. This can result in significant decrease in the number of messages that need to exchanged with the DLM. A database object is display–locked at the DLM only once, no matter how many local displays depend on it. Also, the DLM has to send only one update notification to the client no matter how many of the client's displays are affected.

4.2.2 Designing display classes

The display schema need to be carefully designed to meet the GUI functional requirements efficiently. Under our proposed framework, display objects are also required to request and release display locks for

database objects that affect the appearance of the interface, and react properly upon receiving update notifications.

Object–orientation offers features that allow systematic design of display classes that meet those requirements. More specifically, constructors and destructors of display objects can explicitly request or release the required display locks, since they can have accurate knowledge about their associated database object(s). Moreover, display object methods can encapsulate both GUI functionality and the desired reaction to database updates. In other words, when a display receives an update notification, it can invoke properly designed update methods of the affected display objects, in order to refresh its appearance. In addition, combining display classes into proper inheritance hierarchies can significantly reduce implementation effort.

4.3 System evaluation

In order to test our system, we had up to 4 concurrent users performing simple monitoring and updating functions. In addition, there was a separate process that was continuously modifying attribute values of database objects, simulating real–time network monitoring. Although we only tested our system with a small number of concurrent users, we can briefly present some performance remarks from those tests.

From the user point of view, the application performance was very satisfying, in terms of user interface responsiveness. Considering our tests scale, it is not clear that this is a result of the double caching scheme and the GUI's decoupling from the database workload. But, because of the relatively high update rate caused by the updating process, we can more safely conclude that, at the client side, the display consistency maintenance overhead is very small to deteriorate performance. On the other side, our tests indicated no effect of the server overhead for handling display locks. Extending the traditional locking mechanisms to include display locks will only contribute a very small fraction of overhead and, therefore will not hurt the overall server's performance.

An important performance metric is the time required for updates to be propagated to the users screen. Generally, the actual time between an update commit to the database and its appearance on all relevant displays was in the order of 1 to 2 seconds. We must note that, since copies of updated objects in the client database cache are dirty, this propagation time includes the exchange of at least three network messages: the DLM notification to the client, the client request to the database server for the updated objects, and the database server reply with the objects. The notification protocol could be extended so that updated objects are shipped to the affected clients along with the update notification. In many cases, this more eager approach could eliminate two of the three messages. However, it is not always clear how an update affects the contents of client displays, since the DLM has no semantic information about them, and therefore, it may better to let the clients explicitly request the new information required for display refreshing.

Last, we noticed that the required size for the client display cache was from 3 to 5 times smaller than the corresponding client database cache. Although for our small scale system this was important, it is expected to be a significant factor for real systems whose sophisticated displays are often required to accommodate a very larger number of (possibly bigger) objects.

5 Related work

To the best of our knowledge, not much emphasis has been given by the database research community on user interfaces even though it has been recognized as an important area [11]. User interfaces are usually considered external to a database system and communication is limited to pure data exchange.

Views in the context of object–oriented databases [12] are similar to the display schemas that we propose. As in relational databases, views provide external schemas for user convenience and data protection. They allow dynamic definition of classes, sets of objects, multiple interpretations of objects and can facilitate schema evolution [13]. Work on this area has mainly concentrated on view definition mechanisms and query languages, as in [14] where a query language is proposed to define *virtual classes* which are populated either with database or with *imaginary* objects. Performance and consistency issues have been largely ignored. However, it would be interesting to investigate the applicability of these view definition techniques for dynamic user interface specification.

A two–level client caching architecture has been also proposed in [15]. Their approach uses the top cache level for realizing application specific schemas. However, the focus of this work was on exploiting this architecture to efficiently implement object–views over a relational database as well as reuse local data for answering subsequent queries.

Last, display locks are similar to the *notify locks* presented in [16]. Their difference is that the notify lock algorithms were designed to provide strict transaction concurrency control. However, it seems that the rich set of locks and communication modes offered by ObServer [17] for cooperative transactions, can be used to implement display locks. Non–restrictive read (NR–READ) locks allow a transaction to read an object without prohibiting write privileges to other transactions. These locks can be combined either with the update–notify (U–NOTIFY) communication mode which notifies lock holders upon updates (post–commit notify protocol), or with the write–notify (W–NOTIFY) communication mode which notifies lock holders when another transaction request object for writing (early notify protocol).

6 Conclusions

In this paper we have presented and discussed several issues necessary to support the creation and maintenance of consistent, acceptably performing GUIs for highly interactive, multi–user database applications. We focused on schema design, memory organization and management mechanisms as well as

locking protocols as the principal areas of investigation.

In the area of schema design we noted the difference between database object attributes and methods, and those needed by the GUI only to render the objects on its display. We proposed the adoption of GUI specific display schemas, external to the database. This way, GUIs are realized through instances of that schema, called display objects, that need not (and sometimes cannot) be maintained by the database. For performance reasons, we defined a new niche for them in the topmost level of the system's memory hierarchy where they can be cached according to the GUI needs, not affected either by DBMS policies and parameters or by other concurrent user accesses to the data.

This approach to GUI implementation, gave us the opportunity to address GUI consistency as a client cache coherency problem. Since it is necessary that cached display objects are at all times consistent with persistent, yet updatable, objects, we proposed a locking mechanism based on a display locks and two variations of a notification protocol.

Finally, we presented some implementation issues from a simple prototype of a network configuration management application to test the feasibility of our approach.

Acknowledgments

The authors wish to thank Michael Franklin, Ramesh Karne and Sandeep Gupta for their helpful comments and suggestions.

References

[1] Jayant R. Haritsa, Nick Roussopoulos, Michael O. Ball, Anindya Datta, and John S. Baras, "MANDATE: MAnaging Networks using DAtabase TEchnology", *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 9, pp. 1360–1372, Dec. 1993.

[2] Ben Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley, Reading, MA, second edition, 1992.

[3] Michael J. Franklin, "Exploiting Client Resources Through Caching", in *Proceedings of the 5th International Workshop on High Performance Transaction Processing*, Asilomar, CA, Sept. 1993.

[4] Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Francisco, CA, 1994.

[5] Michael J. Franklin, Michael J. Carey, and Miron Livny, "Local Disk Caching for Client–Server Database Systems", in *Proceedings of the 19th International Conference on Very Large Data Bases*, Dublin, Ireland, Aug. 1993, pp. 641–655.

[6] Michael J. Franklin, *Caching and Memory Management in Client–Server Database Systems*, PhD thesis, Department of Computer Science, University of Wisconsin – Madison, 1993.

[7] John S. Baras et al., "Next Generation Network Management Technology", in *AIP Conference Proceedings 325, Conference on NASA Centers for Commercial Development of Space*, Albuquerque, NM, Jan 1995, pp. 75–82.

[8] Brian Johnson and Ben Shneiderman, "Tree–Maps: A Space–Filling Approach to the Visualization of Hierachical Information Structures", in *Proceedings of IEEE Visualization Conference*, San Diego, CA, Oct. 1991, pp. 284–291.

[9] Harsha P. Kumar, Catherine Plaisant, and Ben Shneiderman, "Browsing Hierarchical Data with Multi–Level Dynamic Queries and Pruning", Technical Report 95–53, Institute for Systems Research, University of Maryland, College Park, MD, Mar. 1995.

[10] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb, "The ObjectStore Database System", *Communications of the ACM*, vol. 34, no. 10, Oct. 1991.

[11] Michael Stonebraker, Rakesh Agrawal, Umeshwar Dayal, Erich J. Neuhold, and Andreas Reuter, "DBMS Research at a Crossroads: The Vienna Update", in *Proceedings of the 19th International Conference on Very Large Data Bases*, Dublin, Ireland, Aug. 1993, pp. 688–692.

[12] Sandra Heiler and Stanley Zdonik, "Object Views: Extending the Vision", in *Proceedings of the 6th International Conference on Data Engineering*, Los Angeles, CA, Feb. 1990, pp. 86–93.

[13] Elisa Bertino, "A View Mechanism for Object–Oriented Databases", in *Proceedings of the International Conference on Extending Database Technology*, Vienna, Austria, Mar. 1992, pp. 136–151.

[14] Serge Abiteboul and Anthony Bonner, "Objects and Views", in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Denver, CO, May 1991, pp. 238–247.

[15] Catherine Hamon and Arthur M. Keller, "Two–Level Caching of Composite Object Views of Relational Databases", in *Proceedings of the 11th International Conference on Data Engineering*, Taipei, Taiwan, Mar. 1995, pp. 428–437.

[16] Kevin Wilkinson and Marie-Anne Neimat, "Maintaining Consistency of Client–Cached Data", in *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Queensland, Australia, Aug. 1990, pp. 122–133.

[17] Mark F. Hornick and Stanley B. Zdonik, "A Shared, Segmented Memory System for an Object–Oriented Database", *ACM Transactions on Office Information Systems*, vol. 5, no. 1, 1987.