Responsiveness and Consistency Tradeoffs in Interactive Groupware

Sumeer Bhola* sumeerb@cc.gatech.edu

Guruduth Banavar† banavar@watson.ibm.com

Mustaque Ahamad* mustaq@cc.gatech.edu

ABSTRACT

Interactive (or Synchronous) groupware is increasingly being deployed in widely distributed environments. Users of such applications are accustomed to direct manipulation interfaces that require fast response time. The state that enables interaction among distributed users can be replicated to provide acceptable response time in the presence of high communication latencies. We describe and evaluate design choices for protocols that maintain consistency of such state. In particular, we develop workloads which model user actions, identify the metrics important from a user's viewpoint, and do detailed simulations of a number of protocols to evaluate how effective they are in meeting user requirements.

Keywords: Replication, Consistency, Response Time, Performance Evaluation, Workloads.

INTRODUCTION

Simple interactive groupware, like chat, whiteboards, and text editors are becoming commonplace. In the future, we can expect complex groupware like engineering CAD (Computer Aided Design), and DIS (Distributed Interactive Simulation) to be widely available. The interactive nature of such applications requires that the effect of a user's action is seen by himself as well as other users in a timely fashion. However, the problem of providing interactive response time is becoming increasingly difficult as groupware is deployed in a wide-area distributed environment like the Internet, where high communication latencies are common. End-users are increasingly accustomed to direct manipulation user interfaces, which typically require response times on the order of 50-100*ms*. However, due to the fundamental limitation of the speed of light, the round-trip delay to the far side of the planet is at least 200*ms*. In mobile wireless computers, intermittent connectivity can also cause large delays. Also, with modem manufacturers optimizing on bandwidth and not latency [5], it is unlikely that latencies when connecting from home will be substantially reduced any time soon. As a consequence of these limitations, groupware systems have been exploring ways to provide interactive responsiveness independent of network latency.

The interactions across distributed collaborating users are supported by shared state. There is general agreement that replication of shared state has the potential to reduce response time for actions that manipulate this state, since a user's action can be executed on her local replica. In addition, it can reduce bandwidth requirements by batching a user's actions before propagating them to other users [3]. However, state replication leads to the problem of replica consistency due to the possibility of different ordering of updates at different replicas.

We assume a model in which the shared state is composed of a set of objects which are fully replicated at all the collaborating processes/users (processes are instances of the groupware applications). These objects are updated through *atomic updates* that are *issued* by the collaborating processes. An atomic update is a code fragment which can read and write a subset of the shared objects in the set, and is guaranteed to execute atomically at each replica. This model eliminates the need for groupware applications to explicitly use locks to achieve atomicity, and permits more flexibility in employing different consistency protocols.

The protocols that are used by current groupware systems to order updates consistently at all replicas fall into two broad categories: *pessimistic* and *optimistic*. Pessimistic protocols usually delay the execution of an update until it is ordered consistently at all sites. An *optimistic* protocol allows a locally issued update to be executed immediately. In essence, an optimistic protocol makes assumptions about the local replica and later confirms these assumptions. The immediate execution of an update corresponding to a user action provides response time that is independent of communication latencies in the system. However, there is a possibility that optimistic assumptions turn out to be false since two conflicting updates can be executed in different orders at different replicas. When this happens, updates have to be undone, and then redone in the *correct* order to get a consistent replica state.

Optimistic protocols in some form have been used in many groupware research prototypes like Grove [7], ORESTE [10], COAST [15], DECAF [17] and Villa [3]. However, there is no quantitative evaluation of the design

*College of Computing, Georgia Tech, Atlanta, GA 30332.
†IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.

choices. In this paper, our goal is to understand and evaluate the design choices for optimistic and pessimistic protocols for interactive groupware. We identify two contrasting classes of consistency protocols: *Dependent* and *Independent*. The Independent protocol allows both optimistic and pessimistic consistency policies, while the Dependent protocols are pessimistic. They differ in how updates are timestamped to determine a global order, and how they commit updates. Commit is important in pessimism to determine when an update can be executed, and in optimism to know when an update will not need to be undone. To capture conflicts between updates of different users, we define three types of contention, *virtual, real, and user-perceived* which affect the behavior of these protocols. In particular, the following are the main contributions of this paper.

- We extend the user interaction model for graphical interfaces to include the concept of a *lookahead threshold*, which models the ability of the user to continue input actions while the previous actions are being executed. We use this to motivate why the issuing and execution of updates should be decoupled, and to develop a general model of user behavior for the synthetic workloads.
- We develop a detailed simulation of both Independent and Dependent protocols. The simulation allows latency to be varied, and assigns a cost to update execution and undo. It is driven by adaptive synthetic workloads, which allow control over parameters like the *lookahead threshold* and the different kinds of contention. The parameter values we use are motivated by real application scenarios. Metrics such as response time mean and deviation, stall count and stall time, message overhead and jitter count are used to understand the tradeoffs when choosing one of the protocols.

The next section describes the interactivity requirements in more detail and introduces the *lookahead threshold*. Subsequently, we describe the two consistency protocol classes we evaluate and how they are affected by the three types of contention. We then discuss the synthetic workloads, how they map to certain application scenarios, and motivate the metrics important for the evaluation. Next, we discuss the main results and how they can be used to choose the appropriate protocol for a certain situation. We then describe related work and conclude.

INTERACTIVITY REQUIREMENTS

In this section we give a more concrete definition of the interactivity requirements for groupware. For this, we build upon the considerable amount of research already done in single user graphical interfaces (for a good introduction see Collins [6], Shneiderman [16]).

The time interval between a user's input action and its response seen by the same user is the *Input Response*(IR) *time*. The key problem in interactive groupware is to keep the IR time low enough as to not differ significantly from single-user applications. This problem is made difficult by the fact that data consistency must not be overly compromised to achieve an acceptable IR time.

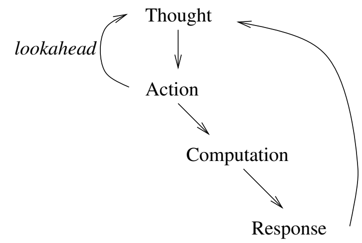

Figure 1: Extended human interaction model

Remote Response (RR) *time* is the time interval between a user's input gesture and its effect seen by a remote user. It is also important to minimize this quantity for interactive groupware, but this is not as critical as low IR time. However, in some scenarios, e.g., two users in the same virtual room involved in a war game, it is important to keep the ratio of RR to IR time small.

The typical model used for user interaction is a cycle of *thought-action-computation-response*. The computation and response part of this cycle is responsible for the IR time. This includes, translating the user *action* into an *update* on the shared state, and execution of this update on the local replica. Card [4] classifies tasks into categories based on the timing of this cycle. *Perceptual tasks* take less than 50-100ms. An example of this is tracking the motion of the mouse pointer. The user handles these tasks without conscious processing.

Due to the very short duration of perceptual tasks, users do not necessarily wait for response from the previous task before going onto the next task. In terms of the human interaction model, we propose that tasks have a *lookahead threshold* (shown in Figure 1), which represents the number of actions that the user is willing to issue before the response to the first one is required. The lookahead threshold for perceptual tasks is expected to be non-zero. As tasks become longer duration this threshold will drop to zero. We now discuss the impact of a non-zero lookahead threshold.

Synchronous versus Asynchronous Processing

A non-zero threshold does not effect most single-user applications, in which response time is totally dependent on local processing. But in collaborative applications, in which IR time may depend on communication latency, it implies that a process can start processing the next user action, if available, before the previous one has executed on the local replica. This asynchrony allows pipelining of the computation required for locally issued actions. Contrast this with a synchronous processing model, in which the process does not begin computing the locally issued action, until the previous action by it has executed on the replica.

To evaluate the impact of synchronous versus asynchronous

Opt

$A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 \dots$

$R_1 R_2 R_3 R_4 R_5 R_6 R_7 R_8 \dots$

SyncPess

$A_1 \quad A_2 \quad A_3 \quad A_4 \dots$

$\xleftrightarrow{l} R_1 \quad R_2 \quad R_3 \quad R_4$

AsyncPess

$A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 \dots$

$\xleftrightarrow{l} R_1 R_2 R_3 R_4 R_5 R_6 R_7 R_8 \dots$

Figure 2: Some Action-Response Patterns

Protocol	Acceptable l (ms)	Unacceptable l (ms)
SyncPess	50	80
AsyncPess	140	240

Table 1: Response Time Expectations

processing, as seen by the end-user, we use a consistency protocol in which every local update is sent to a central server, which totally orders all the updates. The optimistic policy will execute the update locally, before it is sent to the central server and therefore IR time is independent of the network latency. The pessimistic policy waits till the update is returned by the central server, so IR time, say l, depends on the round trip communication delay with the server. These policies can be combined with asynchronous or synchronous processing, to give four choices, SyncOpt, AsyncOpt, SyncPess, and AsyncPess. As synchrony or asynchrony has no impact when we execute optimistically, we are only left with three cases. Figure 2, shows some action-response patterns for the three cases. A_i is the instant when the computation on a user action begins and R_i is the instant when the response is seen locally.

We did an expert evaluation with two tasks, drawing lines and drawing free hand curves, to find the appropriate response time values. Optimism is the perfect case for response time, and we observed that the inter-action (most of the actions were mouse drag events) time was between 20-40ms — clearly a perceptual task. We then asked the experts to provide the minimum l values at which they started noticing the delay in response (acceptable), and when it became very irritating (unacceptable). These are tabulated in table 1. We can see that AsyncPess, which utilizes lookahead to pipeline computation, has a much higher unacceptable latency, implying that lookahead does occur, and can be utilized by the consistency protocol. On the other hand, even a response time of $80ms$ is unacceptable for SyncPess.

More detailed and rigorous user studies using more tasks are necessary to understand how the type of task affects these numbers. However, this preliminary result which shows the significant effect of lookahead, motivates us to use an asynchronous processing model in the rest of the paper. Also, it shows that the value of the threshold can be an important factor when evaluating consistency protocols.

CONSISTENCY PROTOCOLS

The previous section discussed responsiveness. There is also a competing requirement — that of maintaining data consistency. We describe protocols which cover a large design space of protocol choices that can be made for interactive groupware.

We assume a model similar to that in [3], where the shared state is composed of a set of objects that are fully replicated at all the processes in the collaboration. User actions are translated into *atomic updates* by the process, which are then *issued* to the consistency protocol[1]. Each atomic update can read and write a number of objects in the set. The system ensures that all read and write operations of an update are executed atomically at a given replica i.e. operations of two atomic updates do not interleave. Also, as the *lookahead threshold* can be greater than zero, the process can issue atomic updates even when its previously issued updates have not yet been executed locally.

Each update goes through the following stages, issue, timestamp at source, disseminate update with the timestamp to all replicas, execute at each replica. The timestamps define a *partial order* on all the updates issued by the processes. A total order is not essential as concurrent updates could be accessing different objects, and so do not need to be ordered. The partial order must guarantee that any order of execution of the updates that is consistent with this order leads to the same replica state. A pessimistic protocol always executes according to this partial order, while an optimistic protocol may have to reorder execution (using undo,redo), when it notices that it has violated the partial order. An update is said to have *committed* at a replica, when all the updates before it in the partial order have been received and executed. Therefore, a pessimistic protocol only executes an update after it has committed.

Other than the optimistic, pessimistic choice, the *key* aspect in which consistency protocols differ is: Is there any coordination between processes when timestamping concurrently issued updates ? A protocol that does no coordination has to generate a total order, because it does not know if two concurrent updates need to be ordered, so by default it has to order them. We refer to such protocols as *Independent* protocols, and the one we use generates a total order using Lamport clocks [11]. With the optimistic policy, this protocol is similar to the ORESTE [10] protocol, except that we consider atomic updates. The *Dependent* protocols coordinate the assignment of timestamps by using locks, and therefore can generate a partial order. These protocols attempt to use the locality of access by a user to commit updates quickly, which is important when doing pessimistic execution. Therefore, we restrict our attention to pessimistic versions of the

[1] Sometimes referred to as the 'system'.

Dependent protocols. We consider two types of Dependent protocols, one which uses *predeclared* read, write sets (also referred to as access sets) for the updates and another which uses *real* access sets. There are some tradeoffs there, which we discuss later.

Before discussing the protocols, we define two pairs of read, write sets (r_1, w_1), (r_2, w_2) to be *conflicting* if and only if $r_1 \cap w_2 \neq \emptyset$ or $w_1 \cap w_2 \neq \emptyset$ or $w_1 \cap r_2 \neq \emptyset$.

Independent Protocols

These protocols use a Lamport clock at each process, which is incremented whenever a local update is issued, and is used along with the process number, to timestamp the update. These timestamps define a total order on all the updates. The update is then multicast to every process (including the source). Clocks are also modified whenever a timestamped remote update/message is received, using the usual modification rules. Every update received by a process is put in an uncommitted queue sorted in increasing order of timestamp. Each process also maintains a vector of timestamps which are estimates of the clock values of other processes (inferred from the messages received from those processes), and the minimum value in this vector is used to decide when an update in the uncommitted queue can commit. A committed update is removed from the queue. A timestamped heartbeat message is sent out by a process if it has not sent a message in a while, to ensure liveness of commit.

In the *optimistic* protocol (optInd), a process (say p) immediately executes all received updates. For each uncommitted update (say e), it also maintains the access sets (R_e, W_e) of the last time that update was executed. This information is changed whenever an update is undone and then redone. Suppose p receives an update u, which should have been ordered before a set of updates E, which it has already executed. To conform with the total order, p should undo E, however it cheats a bit. It first executes u, and if its access sets (R_u, W_u) *conflict* with R_e, W_e for any $e \in E$, some of the updates in E need to be undone.

For the *pessimistic* protocol (pessInd), updates are only executed after they commit, therefore no detection of conflicts is necessary.

Dependent Protocols

The Dependent protocols use locks, which are associated with mutually exclusive subsets of the objects. A lock has a version number, and the version numbers form a continuous sequence of integers starting with 0. These version numbers are used to timestamp updates associated with that lock, and this timestamp is used to order these updates. For example, suppose the current version number of lock A is 3, and lock B is 1. Then if user Alice issues an update (say u) which requires both lock A and B, then after acquiring both locks, this update will be timestamped with ((A,4),(B,2)). After timestamping, the locks are released, this update is multicast to others, and also inserted in the local uncommitted queue. Update u will be removed from this queue and executed only when update(s) with timestamps (A,3) and (B,1) have been executed. Note, that locks are only useful for timestamping, and are *not required* for committing an update. In our simulation of this protocol, we did not want to choose a specific lock management protocol, so we made a simplifying assumption. We assume that the process requesting the lock is omniscient and knows who last requested that lock. Therefore, it can send the lock request directly to that process, which eliminates the need to forward lock requests. Suppose a process just acquired lock X with version number i, then the updates with timestamp (X, j) where $j \leq i$ have been received or are on their way. Once such updates arrive, the process can commit its local update which has timestamp $(X, i+1)$. To ensure no contradictory ordering information, locks that are acquired to timestamp an update are not released until the update is fully timestamped. This is similar to the 2-phase locking scheme used in databases.

We consider two types of dependent protocols, which differ in how the locks that need to be acquired for an update are determined. Dep1 uses predeclared access sets which are provided when the update was issued, while Dep2 actually executes the update and based on the objects that need to be read or written, acquires the locks on demand. The advantage of using real information is that concurrency is enhanced, because predeclared access sets may include more objects than are actually accessed, however deadlocks can occur if not used very carefully.

Contention and Semantic Information

Due to social protocols mediating between users, in most collaboration scenarios where users are issuing actions concurrently, the concurrency is itself not a problem. However, it can cause problems for the consistency protocols. We define three kinds of contention which impact how these protocols perform. Two updates are *virtually contending* (VC), when their predicted read, write sets conflict. Virtual contention limits concurrency for Dep1 . *Real Contention* (RC) between two updates means that their real read, write sets (the read,write sets when they are executed consistent with the partial order) are conflicting. Real contention limits the concurrency for Dep2 and will cause undos in optInd . *User perceived contention* (UPC) between two updates means that correcting a misordering between the two results in a state which cannot be reasonably explained from the previously observed state. User perceived contention causes *jitter* in the user's interface (or view, in MVC terminology). From the above definitions it is clear that for a pair of updates, UPC⇒RC⇒VC. Note that when using optInd , a user does not notice real contention that is not user perceived, because techniques like double buffering can hide the intermediate states of the view when undoing and redoing updates.

Two questions that arise from the above definitions are, (1) in what scenarios are the three kinds of contention not equiv-

Parameters	Text Editing	Engineering
N (number of users)	4	4
$threshold$	$0, 1, 2, 3$	3
update interval(ms)	$(400, 200)$	$(100, 50)$
numUpdates	1000	1000
latency (ms)	$100, \ldots, 1000$	$30, \ldots, 300$
heartbeat multiple	$1, 2, 3, 4$	3

Table 2: Parameter Values

alent, and (2) how can semantics of the shared data and updates be used to reduce these scenarios. Data-dependent access patterns, which occur, for example, when objects constrain the state of other objects, can cause VC which is not RC. One situation in which RC happens, is when the lock (or object) granularity does not match the dynamic division of work the participants in a collaboration agree upon. Techniques which exploit the type information of the objects can reduce undoes in optimistic protocols[9], and increase concurrency in pessimistic protocols used for groupware [13]. In optInd , undo and redo is completely local (unlike [9]), and potentially very fast, and because we do not expose the intermediate states to the user, semantic information will probably not improve the performance of optInd significantly. For Dep2 , techniques like multi-granularity locking [2], can increase concurrency, however there are two problems which may limit improvements, or even worsen the situation. Firstly, in scenarios we consider with data-dependent access patterns, the granularity of what will be accessed by an update is not known beforehand, and so too many locks may be acquired. Secondly, having many locks may reduce the locality of acquiring locks by a process. For example, in the first sequence lock table given in [13], which can be used for text editors, insertion of each character causes a new lock to be acquired. The performance impact of semantic techniques on end-users needs more investigation, but is outside the scope of this paper.

Another situation which causes real read-write contention occurs when the objects represent some physical entity, and there are physical constraints between them. For example, for collision detection in DIS, updates to the position of every entity require the positions of other entities in the neighborhood to be read. However, the read value is discarded whenever collisions haven't occured.

We consider all three types of contention, and a range of values for each, when doing the evaluation.

EVALUATION SETUP

Workload

One of the main problems in evaluating the consistency protocols is that user input (both *what* is input, and *when* it is generated), can change with varying response time. We are not aware of any traces of user interaction with complex collaborative applications which we can use to evaluate these protocols. As a first step in real evaluation of such proto-

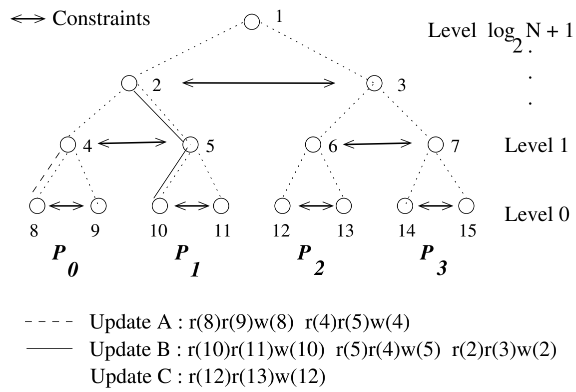

Figure 3: Forest Workload

cols we construct adaptive distributed synthetic workloads. These workloads are adaptive in that they use the lookahead *threshold* to change the timing of the user input based on the response seen. A threshold of 0 means that a process does not issue a new update until the previous one has been locally executed i.e. each process only has 1 outstanding request at any time. Therefore a threshold of t means that a process can have $t + 1$ outstanding requests, and it stalls until the current lookahead falls below $t + 1$. A uniform distribution with a certain mean and range is used to model the *inter-update interval*. *numUpdates* is the total number of updates issued by each process and *latency* is the one-way communication latency between processes. The *heartbeat multiple* gives the inter-heartbeat interval for the Independent protocols as a multiple of the *latency*.

The key characteristic of a workload is the number and type of objects in the shared set, and the access patterns of the updates issued by each process. We use two realistic collaboration scenarios to motivate these characteristics.

Text Editing *N=4* users are editing a document which is divided into sections, each of which is modeled as a shared object. Assume that the Dependent protocols use a lock per section. Depending on the situation, each user is editing his own section, or two users are editing different parts of the same section. To capture this behavior, we vary a parameter h from $0 \ldots 1$, where *h=0* implies no contention of any sort, while *h=1* means that there is real contention between pairs of users. There is *no* user perceived contention. The range of values of the other parameters is given in table 2.

Engineering Design This scenario is motivated by an example collaboration in the SHASTRA system [1], and involves the design of a complex physical model.

Although it can be scaled to higher N values, we illustrate it with *N=4*. All the vertices in figure 3 represent shared objects. The leaf vertices represent primitive models which are combined hierarchically to create more complex models. Two models with the same parent model constrain each

other, and therefore a write of one has to read the other. For example, an update which writes model 5 must read model 4. In this scenario, each process reads and writes two primitive models. For example, process P_1 reads and writes model 10 and model 11. With a certain probability p, a write of a model at level i, results in a write of its parent at level $i+1$. Also, we can control the number of levels in the workload i.e. a maximum level of 2 means that there are 14 model objects, with model 2 and 3 not constraining each other. The number of the highest level is denoted by $h+1$. The value of h controls the amount of virtual contention, and is varied between $0, 1, 2$. As each update has the potential to write its ancestor at level $h+1$, for Dep1 we use $N/2^h$ locks. For Dep2 , there are $N/2^k$ locks for level k, with no additional lock required for level $h+1$. As children of the same parent share a lock, and every process moves from the bottom to the top when acquiring locks, there is no possibility of deadlock. p controls how much of the virtual contention is real and is varied between $0.0, 0.2, \ldots, 1.0$. In figure 3, with *h=2*, updates A and B cause real contention, while A and C, or B and C are only virtually contending. Note that because Dep1 uses predeclared access sets, we have used locks which are coarse grained as compared to the locks used for Dep2 .
This workload allows virtual and user perceived contention to be modeled in an elegant way. The range of values of the other parameters are given in table 2.

Performance Metrics

We consider the following metrics.

1. Local response time (mean and deviation) : Along with a low mean, a low deviation is also important to avoid user surprise due to wide variation in response time.

2. Local commit time (mean and deviation) : For the pessimistic protocols the commit time is the same as the response time. A low commit time is also good for an optimistic protocol as it reduces the possibility that an update may be undone a long time after it was initially performed.

3. Jitter Count : This is only important for the optimistic protocol. This counts the number of undos that were user perceived.

4. Number of messages of each type : Important for measuring the communication cost of the protocols. Other than the messages to disseminate the updates, which are the same for each protocol, the *extra messages* for the Independent protocols are due to heartbeat messages, and those in the Dependent protocols are due to lock request and reply messages.

5. Stalled count and Stalled time : Stalled count measures the number of times the user had to stall because the previous $threshold+1$ updates had not been locally performed. Stalled time is the mean time for which the user stalled.

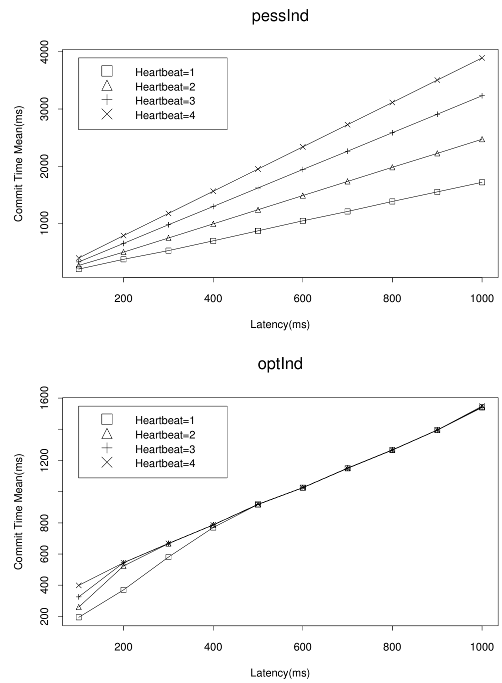

Figure 4: Commit time vs. Latency. Threshold=0

6. Total execution time : Measures how long the users took to finish their tasks.

RESULTS

We simulated the protocols with both the text editing and engineering design workload. In this section we summarize some of the interesting results.

Text Editing

As RC=VC and each update only needs one lock, Dep1 and Dep2 behave the same, and will be referred to as Dep. Due to the large number of parameters, we highlight the effect of the *threshold* and *heartbeat multiple* and then fix their values to reasonable levels. A scatter plot of the commit time versus the threshold, for all the protocols, showed that it was *usually* not affected by the lookahead threshold value. This may be expected as the threshold value affects when an update is issued, but should not affect the commit time of an already issued update. However, when *threshold=0* for pessInd , the commit time was much larger than when *threshold* > 0. Also, it was much larger than the commit time for optInd with *threshold=0*. To explain this, figure 4 shows the effect of heartbeat multiple and latency on commit time for pessInd and optInd when *threshold=0*. For optInd , the effect of heartbeat frequency decreases as latency is increased. This is because the inter-update interval is much lower than the latency so heartbeats are not really needed. However, as pessInd executes updates pessimistically, it stalls after issuing each update, and hence heartbeat messages are very important to pick up the slack in message fre-

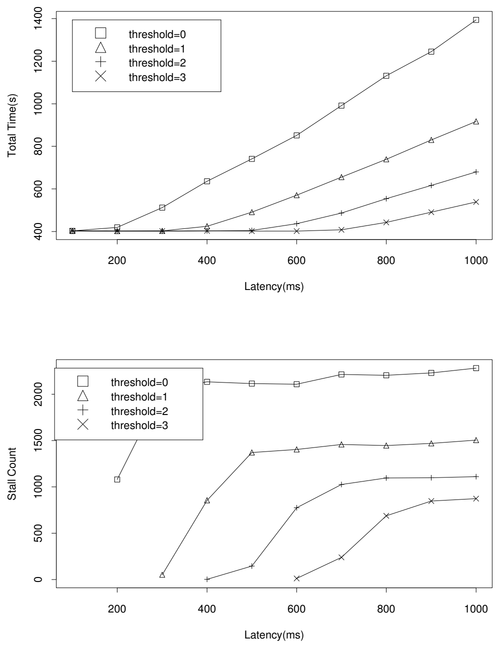

Figure 5: Total time and Stall count for Dep ,h=1

quency. It can be seen that with *heartbeat=1* the commit is almost as fast as that of optInd , however in this case the number of heartbeat messages was of the order of the total number of updates. This gives us the following observation.

O 1 *When* threshold=0*, increasing the heartbeat rate decreases the response time and total time for* pessInd *significantly. However,* optInd *is unaffected by heartbeat rate at high latencies.*

Next, we looked at how threshold affected the total time of the task. As expected, it did not affect optInd , and Dep when *h=0*, but had a strong effect on pessInd , and Dep when *h=1*. Figure 5 shows the effect of threshold and latency on Dep when *h=1*. As expected, the total time for the task decreases with increasing threshold value. Increase in total time with latency is due to increasing number of stalls (up to a certain maximum, dependent on the threshold value) and increasing time for each stall. For *threshold=3*, no process stalls until *latency=600ms* and therefore the total time taken remains constant until then.

O 2 *In a task with a high value of threshold, the user can perform well even at high latencies, because of the lower total number of stalls and the smaller mean time for each stall.*

The results pertaining to varying contention are better examined in the engineering design experiments.

Engineering Design

We fixed the *threshold=3* and *heartbeat=3*. With no VC, i.e. *h=0*, the performance of optInd and Dep1 is equivalent, as neither communicates with other processes before executing

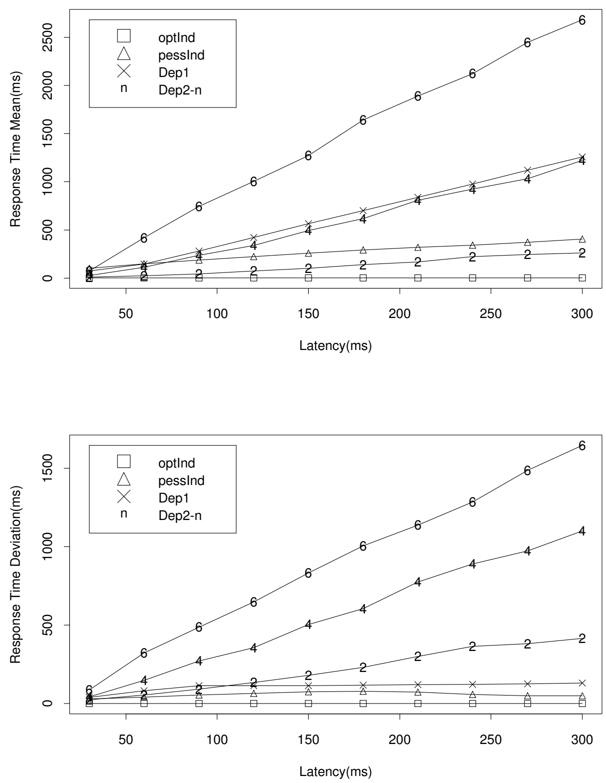

Figure 6: Responsiveness vs. Latency. h=2

a local update. Similarly, when $h = 0 \vee p = 0$, i.e. there is no RC, optInd and Dep2 are equivalent. In the plots shown, Dep2-n refers to the Dep2 protocol when $p = 0.n$. The values of p, h, i.e. RC and VC, have no effect on pessInd . In the following discussion, *low* VC refers to *h=1*, *high* VC to *h=2*, *low* RC to $p = 0.2, 0.4$ and *high* RC to $p > 0.4$. We first examine metrics in which p, h do not affect optInd [2]. Figure 6, shows that pessInd gives a lower response time than Dep1 beyond a latency of $60ms$. Dep2 with $p \leq 0.2$ is better than pessInd , however it deteriorates quickly with increasing real contention. In fact, at high real contention ($p > 0.4$), Dep2 performs much worse than Dep1 , because it is acquiring multiple locks in sequence for every update, while Dep1 only acquires one lock of coarser granularity. This gives us the following observation.

O 3 *When* VC *is high and* RC *is low,* Dep2 *has lower mean response time than* Dep1 *and* pessInd *. However, with high* RC*,* Dep2 *can have higher mean response time than* Dep1 *and* pessInd *, and this difference increases with increasing latency.*

Looking at the standard deviation of response time, we see that it is lower for pessInd than both Dep1 and Dep2 . Also, the deviation for pessInd and Dep1 is relatively in-

[2] Although undo and redo do impact these metric values, the effect was insignificant.

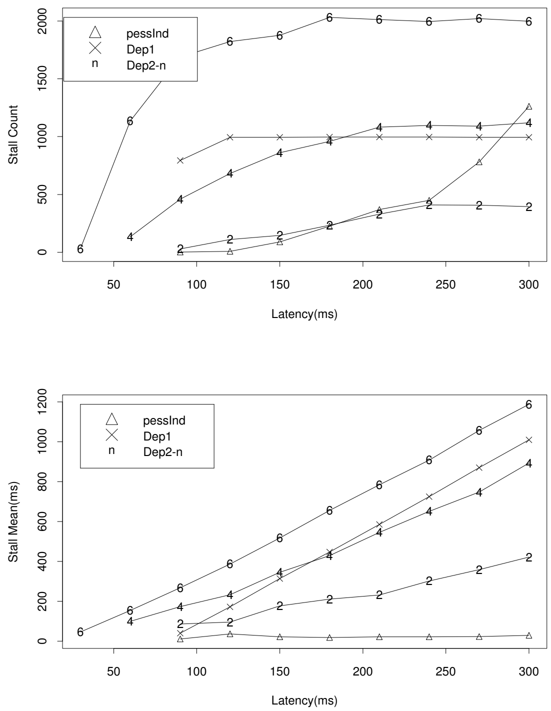

Figure 7: Stall vs. Latency. h=2

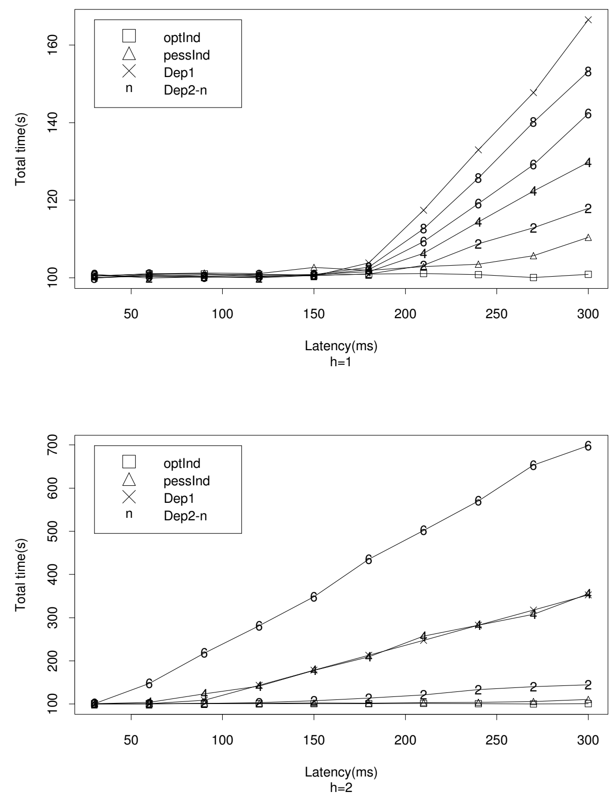

Figure 8: Total time vs. Latency

dependent of latency, even though the variation in latency is increasing with increasing mean latency. This can be attributed to the sum of many independent random variables, like the latency and inter-update interval, having a lower deviation than each of them alone. However, with $p = 0.2$, Dep2 has a higher deviation than both Dep1 and pessInd, even at very low latencies, because for each update it may have to acquire anywhere between 0 and 2 locks.

O 4 *Even with high* VC *and low* RC*,* Dep2 *can have a higher response time deviation than* Dep1 *and* pessInd*, and this difference increases with increasing latency.*

Figure 7 shows how the stall count and mean vary with latency. The stall count of optInd, Dep1 with no VC, and Dep2 with no RC is zero. The stall count for pessInd rapidly increases after *latency=240ms*, because the lookahead can no longer suppress the effect of the latency. However its stall mean is relatively stable and is around $25ms$. In our user model, in which the user stalls when her lookahead exceeds the threshold, the relative importance of stall count and stall mean needs to be investigated i.e. does the user want to wait more number of times or wait longer each time. An alternative user model, the *no-stall model*, may be more appropriate in certain situations. In this model the number of stalls are minimized by using the past response time information to increase the inter-update interval. In our model, the stall count for Dep1 stabilizes at $N * numUpdates/(threshold + 1)$.

O 5 *The stall count of the Dependent protocols is limited by the* threshold*, while that of* pessInd *keeps increasing with latency. However, the mean time for the stall is independent of latency for* pessInd*, but keeps increasing for the Dependent protocols.*

Figure 8 shows how the total time varies with latency with *h=1,2*. Even though the response time mean for pessInd was usually higher than the Dependent protocols with low contention, the total time does not show this. This is because the lower stall mean of pessInd compensates.

O 6 *For low* VC *and* RC *and* $latency \leq 180ms$*, the choice of the protocol has negligible effect on the total time taken to perform a task. However, with high* VC *and low* RC*, the Dependent protocols perform significantly worse than the Independent protocols.*

Note that we have not modeled the increasing errors, or user frustration that may occur when response time increases, which can impact the total time. Therefore, acceptable response time values for the task need to be considered along with this total time information when choosing a protocol.

From figure 9, we can see how many extra messages are sent per second, by each of the protocols. For the Independent protocols at a very low latency, the heartbeat interval ($3 * latency$) is lower than the inter-update interval, so a lot of extra heartbeat messages are sent. Otherwise, the Inde-

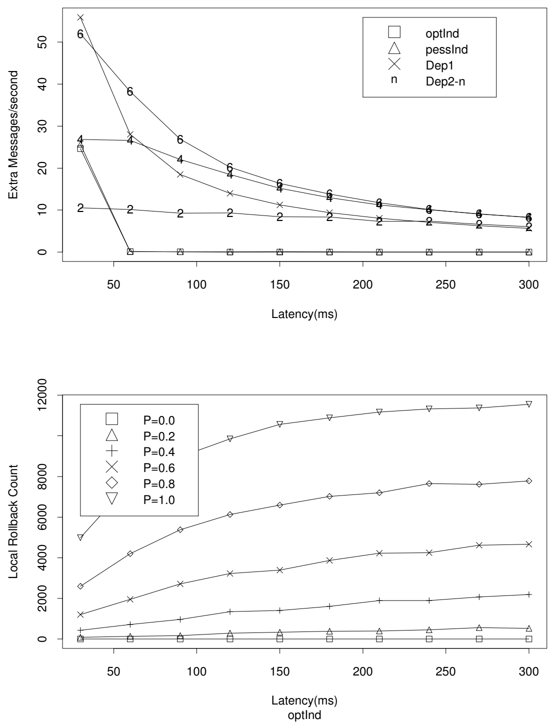

Figure 9: Extra Messages, Rollback count. h=2

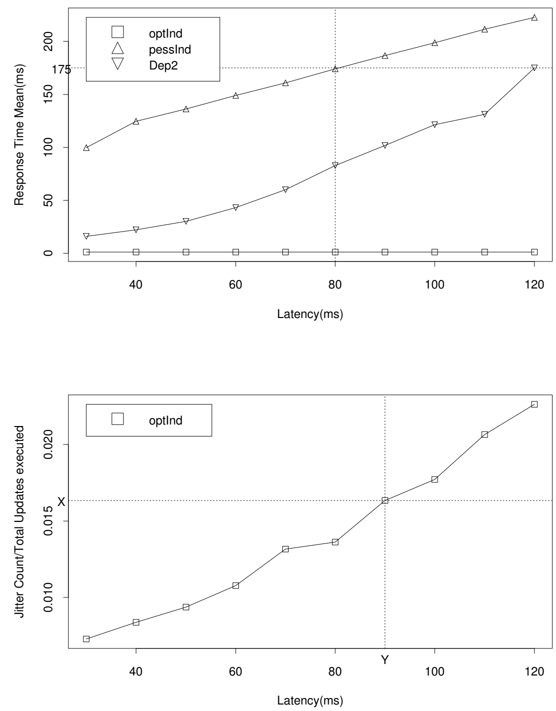

Figure 10: Protocol comparison. h=2,p=0.3

pendent protocols have 0 extra messages. The extra messages for the Dependent protocols slowly decline as latency increases, as the total time to perform the task is increasing. Also, the same figure gives the number of rollbacks/undos for optInd as the latency and RC increases. Note that how many of these rollbacks cause user perceived contention (jitter) is very dependent on the collaboration scenario. However, the amount of jitter that is tolerable, is a very important factor in determining whether optInd is useful for a certain collaboration scenario.

Discussion

An important question is, given the situation, how do we apply the preceding information to decide which protocol to use. We illustrate with an example. Suppose we know that *h=2,p=0.3*, that the latency can vary between $30ms$ and $120ms$, and 50% of the rollbacks cause jitter. Figure 10 shows the mean response time for optInd, pessInd, Dep2 , and the fraction of the total updates executed which caused jitter, which we call the *jitter fraction*. If the maximum tolerable jitter fraction is greater than 0.022, optInd is clearly the best because of its instantaneous response. If the maximum tolerable jitter fraction is a value x, less than 0.022, we have a choice to make. Let y be the latency value corresponding to this value x. We can use optInd when the latency is less than y, and then dynamically switch to pessInd . This can be easily done because the choice to switch from optInd to pessInd and vice versa can be made *locally* by each process. However, if we also cannot tolerate a mean response time greater than $175ms$, we will have to use Dep2 after a latency threshold of $80ms$ is reached. As it is hard to dynamically switch between Dependent and Independent protocols without coordination between all the processes, which may not be possible, we may then have to use Dep2 for the whole collaboration.

RELATED WORK

Greenberg and Marwood [8] qualitatively discuss the impact of different concurrency control schemes, including optimistic and pessimistic protocols, on certain collaboration scenarios. They observe that social protocol between participants make conflicts rare, and in many cases it is possible for users to notice conflicts and manually repair them. For the next generation of collaborative applications, we expect that conflicts usually occur due to secondary effects of user actions, for example, automatic constraint evaluation in engineering design, and collision detection in distributed interactive simulation. In such scenarios, conflicts will be more common, and we cannot expect the user to manually fix the state.

For replicated state, two types of consistency protocols have typically been used. Explicit locking approaches use locks for atomicity and concurrency control, like DistView [14]. Ordering based approaches need an associated commit protocol. Our Independent protocol is similar to ORESTE [10]. DECAF [17] uses a primary copy commit protocol, which is a dependent protocol. The advantage is that an update issuing site only needs to talk to the primary copy sites for the update to commit. By strategically placing the primary copy sites, the commit for certain sites can be made faster. The disadvantage is that the protocol uses the primary copies to accept or reject the timestamp assigned by the update issuer, so there is no bound on how many times the guess for a certain update may be rejected. A protocol in which the primary copies assign timestamps can also be constructed, and our dependent protocol is a dynamic variation of this, in which the primary copies can move.

Finally, there has been some work in understanding how poor response time affects human performance for single user tasks. MacKenzie and Ware [12] use a target acquisition task in which the user has to move the mouse cursor into a rectangular target. They see that increasing the response time from $8.3ms$ to $225ms$ (for an asynchronous processing model) increased the time taken to perform the task by 64% and the error rate by 214%. This shows the importance of low response time for human performance.

CONCLUSION

The conflicting goals of fast response and consistency are difficult to meet for interactive groupware deployed in widely distributed environments. We have developed workload models and quantitatively evaluated a number of choices for consistency protocols that can be used to meet the response time and consistency needs of such applications.

For such high latency environments, the concept of a *lookahead threshold* for user actions is very important for a true evaluation. This models the users ability to issue a number of actions without waiting for the first action to complete. We have used two classes of protocols that can be used to consistently execute the user updates at all sites. With a novel workload model, we have evaluated these consistency protocols along a number of dimensions.

Our results show that when lookahead is high and contention is low, the pessimistic protocols perform significantly better than expected. The pessimistic dependent and optimistic independent protocols are the best when there is no contention. However, with a small amount of real or virtual contention, the dependent protocols have a much higher response time deviation than the pessimistic independent protocol. When there is user perceived contention, the optimistic independent protocol could have problems due to high jitter. Based on the parameters of the operating environment, like latency and contention, and the bounds required on the performance metrics, we have shown how our plots could be used to choose the appropriate protocol.

Future work involves using detailed user studies to further refine our workloads, and to better understand the relative importance of the metrics we measured. Also, better consistency protocols, which utilize application semantics, and adapt to changes in the external and collaboration environment, need to be designed and evaluated.

REFERENCES

1 C. Bajaj and V. Anupam. Collaborative multimedia in scientific design. *IEEE Multimedia*, 1(2):39–49, 1994.

2 P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

3 S. Bhola, B. Mukherjee, S. Doddapaneni, and M. Ahamad. Flexible batching and consistency mechanisms for building interactive groupware applications. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*, 1998. To appear.

4 S. Card, T. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.

5 S. Cheshire. Latency and the quest for interactivity. Commissioned by Volpe Welty Asset Management, L.L.C., November 1996.

6 D. Collins. *Designing Object-Oriented User Interfaces*. Benjamin/Cummings Publishing Company, 1995.

7 C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD'89*, pages 399–407, 1989.

8 S. Greenberg and D. Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the Fifth ACM Conference on Computer Supported Cooperative Work (CSCW)*, 1994.

9 M. Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1), March 1990.

10 A. Karsenty and M. Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proceedings of the 13th ICDCS*, pages 195–202, 1993.

11 L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

12 I. S. MacKenzie and C. Ware. Lag as a determinant of human performance in iteractive systems. In *INTERCHI'93 Proceedings*, 1993.

13 J. Munson and P. Dewan. A concurrency control framework for collaborative systems. In *Proceedings of the 6th CSCW*, 1996.

14 A. Prakash and H. S. Shim. Distview: Support for building efficient collaborative applications using replicated objects. In *Proceedings of the 5th CSCW*, 1994.

15 C. Schuckmann, L. Kirchner, J. Schummer, and J. M. Haake. Designing object-oriented synchronous groupware with COAST. In *ACM CSCW'96*, 1996.

16 B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 2nd edition, 1992.

17 R. Strom, G. Banavar, K. Miller, A. Prakash, and M. Ward. Concurrency control and view notification algorithms for collaborative replicated objects. In *Proceedings of the 17th ICDCS*, 1997.