Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware

T.C. Nicholas Graham *Tore Urnes* *Roy Nejabi*

Department of Computer Science
York University
4700 Keele St., North York
Canada M3J 1P3
{graham,urnes}@cs.yorku.ca

ABSTRACT

The Model View Controller (MVC) architecture has proven to be an effective way of organizing synchronous groupware applications. Distributed implementations of MVC, however, can suffer from poor performance. This paper demonstrates how optimized semi-replication of MVC architectures can lead to good performance over both local and wide area networks. We present a series of optimizations to network communication based on specific communication properties of groupware. These optimizations have been implemented in the Clock groupware development toolkit, allowing programmers to develop applications directly in the high-level MVC style, with Clock automatically providing optimized performance. Timings of an application developed in Clock show that usable speed was obtained in a highly interactive groupware application running between Toronto and Calgary, with a typical latency of 190 ms per round trip message. The paper discusses the tradeoffs involved in the algorithms, and presents timings to demonstrate the effectiveness of the different approaches. The timings show that when running over a wide area network, the best optimization can achieve a factor 60 speedup over the naive implementation of distributed MVC.

KEYWORDS: groupware, groupware toolkits, performance evaluation

1 INTRODUCTION

The implementation of groupware is challenging, due to the need to support a high-level, iterative and incremental development method, while at the same time requiring excellent performance. Considerable research effort has been devoted to finding means of providing high level support for the development of groupware while also providing sufficient performance in the resulting applications.

One promising approach has been to organize groupware applications around derivatives of the *Model-View-Controller* (MVC) architecture [7]. The key idea of MVC is that the data underlying the application (the *model*) is separated from input handling code (the *controller*) and display maintenance code (the *view*). In a groupware application, each user forms a separate view-controller pair, while the model represents the shared data and application functionality. The MVC architecture frees the programmer from issues of maintaining consistency between user's views, even in the presence of relaxed WYSIWIS (what you see is what I see) behaviour. Groupware toolkits based on MVC-like architectures include RendezVous [6], Weasel [4] and GroupIE [11].

Efficient distributed implementation of groupware based on MVC has proved difficult, due to the latency of network communication between the distributed components of the user interface. As will be seen in section 4, our experiments show that in a straightforward implementation of a distributed MVC architecture, as much as 95% of view update time can be spent waiting for the network.

This paper presents techniques for optimizing the implemention of distributed groupware based on the MVC architecture, and demonstrates experimentally how effective each technique is. The three optimizations, *caching*, *request prefetch* and *request presend* identify communication patterns typical of groupware, and provide optimal behaviour for those patterns. Experiments show that request presend, the most effective of the optimizations, leads to usable performance in a highly interactive groupware application running on a standard Internet connection between Toronto and Calgary, a distance of 3,000 km.

The paper is organized as follows. Section 2 presents the MVC architecture, and shows how MVC can be implemented in a distributed setting. Section 3 describes the optimizations. Section 4 then reports experimental performance results when these optimizations are applied to the Clock groupware development tool.

2 MVC AS THE BASIS FOR GROUPWARE

To help motivate how groupware is organized under the MVC architecture, figure 1 shows an example of a syn-

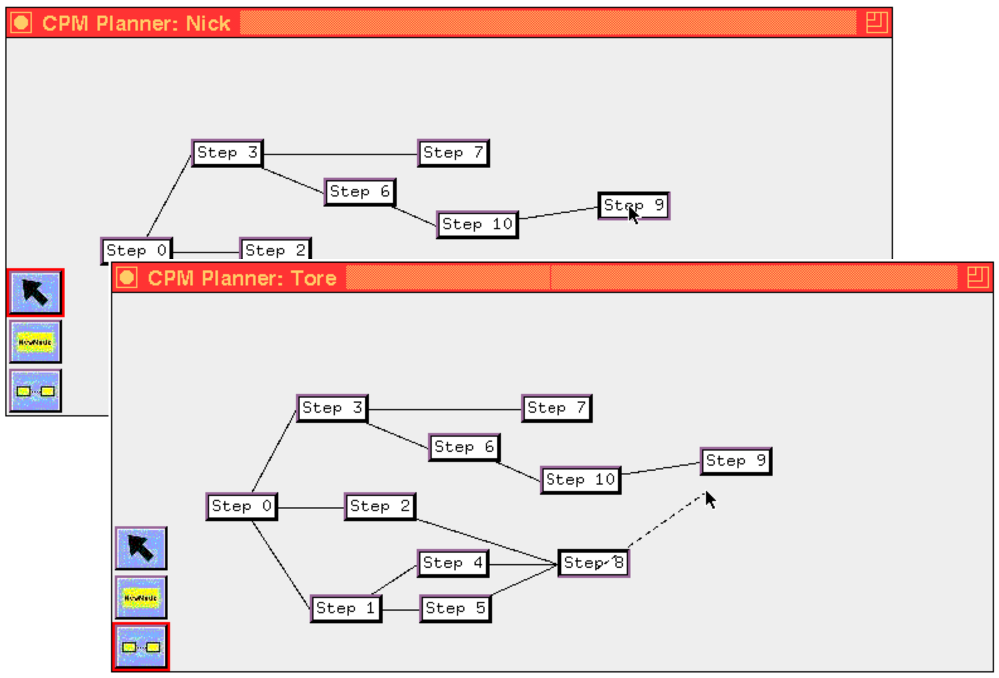

Figure 1: A critical path planner application implemented in Clock. Nick is moving a node while Tore is connecting two nodes.

chronous groupware application, a simplified planning tool based on the critical path method. Any number of users may view and modify a network representing the scheduling of a set of tasks by moving, adding or connecting nodes. The application is relaxed WYSIWIS (what you see is what I see), where each user sees the same network and immediately sees the results of changes made by other users, but where users may select different modes; for example, one user may be connecting while another adds a new node. The application enforces automatic node locking, so that only one user may move any given node at a time. Implementing this application involves maintaining consistent displays for all users, arbitrating concurrent actions, and customizing the appearance and behaviour of each user's view. This application was developed using the Clock groupware development tool, and forms the basis of the experiments described in this paper.

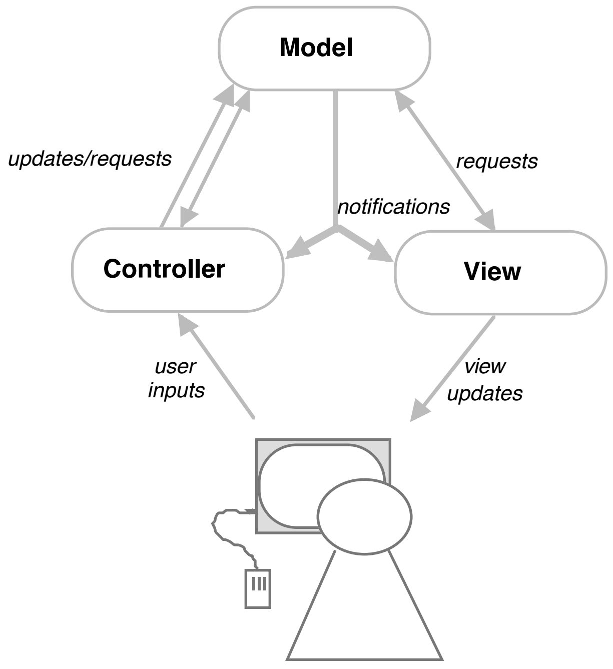

Figure 2: The MVC Architecture.

Figure 2 shows the MVC architecture as implemented by Clock. Applications are split into three parts – a *controller* responsible for input handling, a *view* responsible for output, and a *model* implementing the underlying application and data. The controller translates user inputs into updates to the model state. When its state has been modified, the model notifies the view that view updates may be necessary. The view then recomputes what (if any) display updates must be made. This architecture frees the model from details of how views are updated, and frees the controller from having to determine which views must be modified as a result of user inputs. Modern user interface toolkits derived from MVC include layered extensions such as PAC [2], and constraint-based approaches such as Garnet [8].

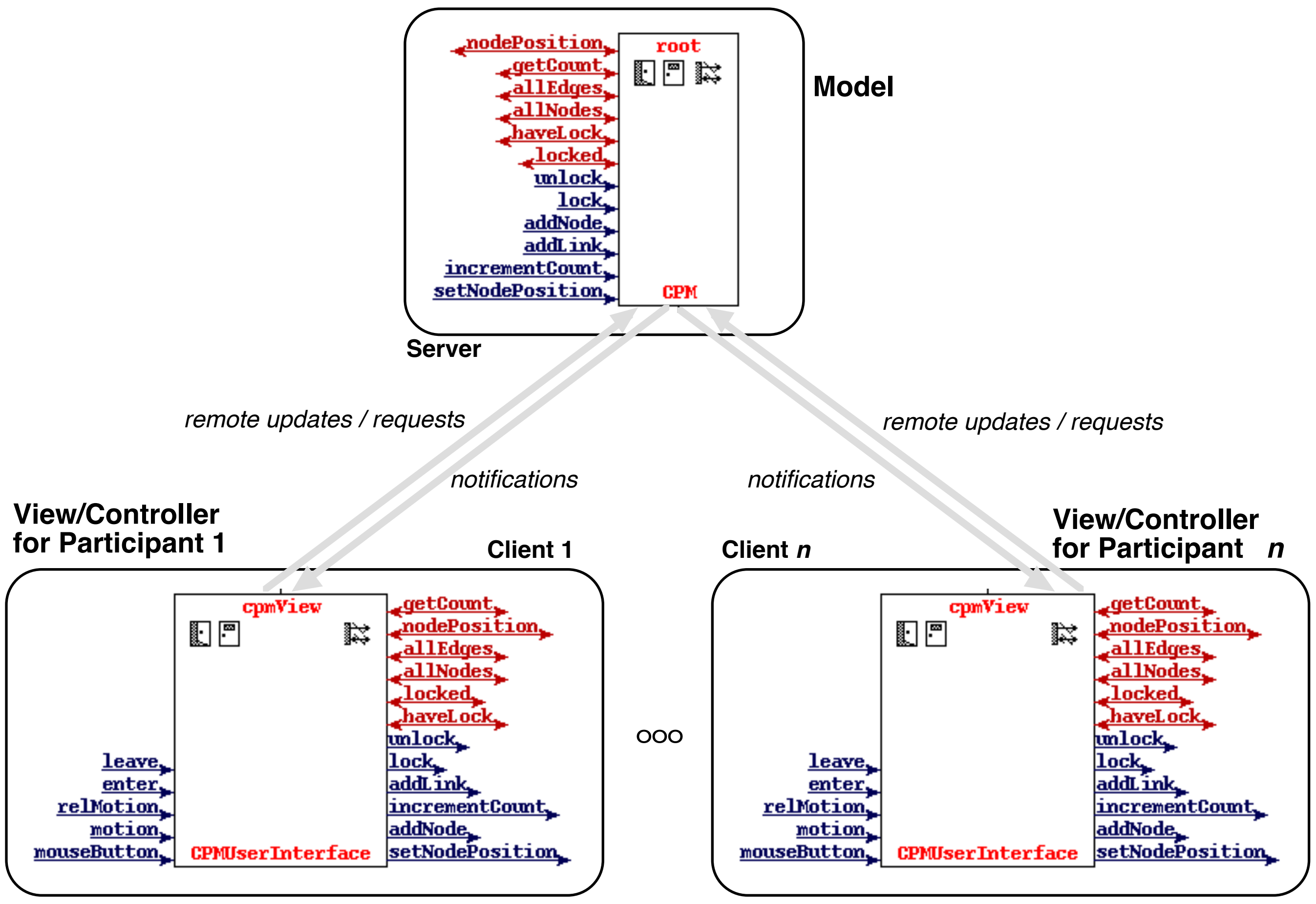

Figure 3: The semi-replicated implementation of MVC in Clock. The model is implemented on a centralized server, while each user's view/controller is implemented on his/her client machine. The left side of the model is annotated to show the methods it provides, while the right side of the view/controllers are annotated to show the methods they invoke. In Clock, controllers are implemented via event rules, and views are implemented via constraints.

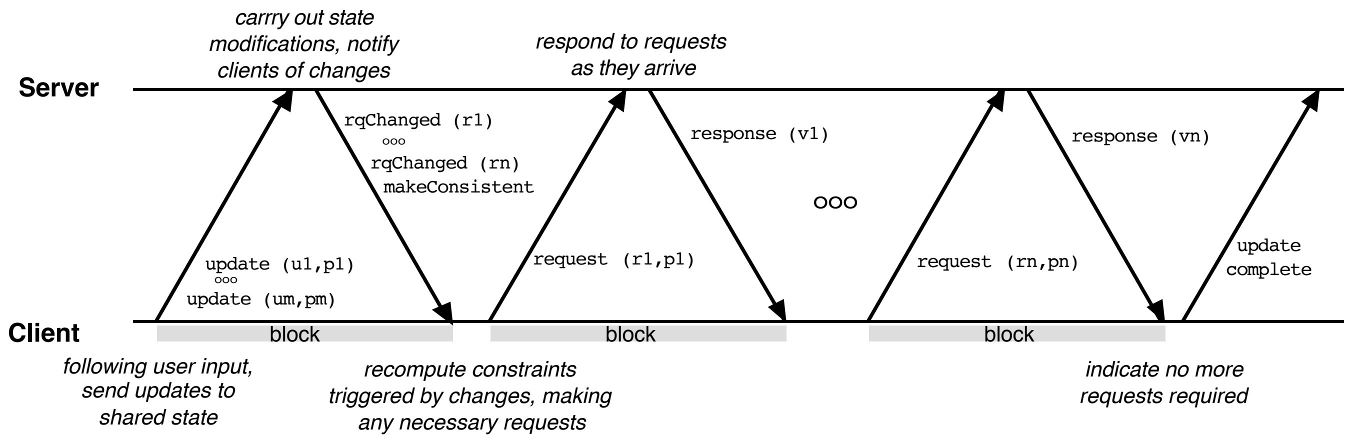
carrry out state modifications, notify clients of changes
respond to requests as they arrive
Server
rqChanged (r1)
ooo
rqChanged (rn)
makeConsistent
response (v1)
ooo
response (vn)
update (u1,p1)
ooo
update (um,pm)
request (r1,p1)
request (rn,pn)
update complete
Client
block
block
block
following user input, send updates to shared state
recompute constraints triggered by changes, making any necessary requests
indicate no more requests required

Figure 4: Naive protocol for updating views in response to changes in the model.

MVC provides an excellent conceptual organization for programming groupware. Figure 3 shows how the planner application of figure 1 is organized in Clock. The model implements all data and operations shared by the users of the planner. The shared information includes the structure of the network, the positions of the nodes and locking information. The operations provided by the model include *request methods* (shown with a horizontal double-headed arrow) for querying the shared state, and *update methods* (shown with a horizontal single-headed arrow) for modifying the shared state. Example requests include *nodePosition* (obtain the position of a node), *allNodes* (obtain the list of all nodes) and *allEdges* (obtain the list of all edges). Example updates include *setNodePosition*, *addNode* and *addLink*.

A view/controller pair is dynamically provided for each participant in the current groupware session. Each view/controller pair is responsible for implementing the user interface of that participant. A controller may update the shared state by invoking one of the update methods provided by the model. When the model state is modified, the model notifies the views of *all* participants that display updates may be required. The views can then update themselves in parallel, making necessary requests to the model.

This architecture provides a good separation of concerns. The model contains no information on how views are computed. This avoids the complexity of centrally keeping track of a dynamically changing number of customized views. Since views contain no shared data, bringing latecomers into a session is also straightforward. Since the model automatically triggers view computation in the different views, controllers do not need to be aware of how the inputs of one user affect the displays of other users. This approach is similar to the organization of groupware applications in the RendezVous ALV model [6], where constraints (rather than requests) are used to propagate information from the model to the view/controllers, and to the program organization under GroupIE [11], which follows a more traditional MVC organization.

The MVC architecture is also straightforward to implement in a distributed setting. In what has come to be called a hybrid centralized-replicated architecture (or *semi-replicated* architecture [4]), the model is implemented on a central server machine, while the view/controller pair for each user is implemented on the user's local client machine. This approach aids scalability, since the main work of implementing the user interface of each user is distributed to the clients.

The Clock run-time system maps high-level architecture specifications in the MVC style into distributed implementations, where all requests, updates and notifications are automatically implemented over the network. This allows programmers to operate within a high-level programming framework, leaving the run-time system to automatically take care of issues of distribution, network communication, and concurrency control.

The serious drawback of semi-replicated architectures is that the networked implementation of requests from the view to the model quickly dominate view update time. As will be seen in section 4, our experiments show that with the naive implementation of remote requests (i.e., simply making requests when necessary and blocking until the response is received), performance is unacceptably slow even in a local-area context, and completely unusable in a wide-area context. In computing a view over a wide area network, as much as 95% of the client's time can be spent blocked, waiting for responses to requests.

To address this problem, we have developed and tested three algorithms for optimizing network communication. As summarized in section 4, when running on a standard Internet wide area network between Toronto and Calgary, the best optimization gives up to a factor 60 improvement over the naive approach, making the example application acceptably responsive. These optimizations have been built into the Clock groupware development tool. This means that Clock programmers do not have to optimize communication themselves – they simply write code directly in the MVC style, and leave optimization to the run-time system.

3 OPTIMIZING DISTRIBUTED MVC

In order to understand the optimizations to distributed MVC, it is first necessary to look at why a naive implementation is slow. Consider the sequence of communication that is required when a user performs an update to the model. The update must be sent from the controller (implemented on the client machine) to the model (implemented on the server machine.) The model responds by informing the client that its view may be out of date. The view is then recomputed, potentially requiring information from the server. Figure 4 shows the messages exchanged between a client and server in updating the model and then computing a new view. Following a user update, the client generates a series of updates $u_1(p_1), \ldots, u_n(p_n)$ which are sent to the server. The server carries out the updates, and then notifies each of the clients of request values that may have changed (through the messages `rqChanged` $(r_1), \ldots,$`rqChanged` (r_n).) The clients then recompute those parts of their views that may be affected by the change, generating requests for the server. Following each request, the client blocks until the response is returned.

As the distance between the client and server increases, network latency increases, and the time spent blocking quickly dominates the view update time. The optimizations presented in the next three subsections all aim to reduce the time clients spend waiting for the network, at the cost of performing extra work in both the client and server to avoid or optimize communication. The timing results of section 4 show that even in the relatively low-latency case of a local area network, these optimizations all improve performance.

In order to implement this protocol correctly, some kind of concurrency control scheme is required [5]. Any scheme can be used; for now, the Clock system uses a simple server locking scheme that in effect serializes user updates. Ongoing work is directed at more sophisticated schemes for ensuring the atomicity of updates without necessarily requiring central locking.

3.1 Optimization 1: Request Caching

The idea behind request caching is to dynamically collect the values of remote requests in a cache. As long as we can prove that the value of the request has not changed since we last made it, we can use the local cache value rather than the remote value. As will be shown in section 4, this optimization alone improves performance by a factor of seven in our example application.

Figure 5 shows how a request cache is constructed. Whenever a client makes a request, the request and its value are entered into a cache list attached to the client. For example, here the position of node 1 was last reported to be the coordinate $(50, 50)$. When subsequent requests are made, they are compared to the requests in the cache. If the request is available in the cache, the local result is used, saving the cost of a remote request.

The server is responsible for invalidating cache entries. Whenever an update is made to server state (step 1 of figure 6), all requests implemented by the handling component potentially change value. The server then informs the clients

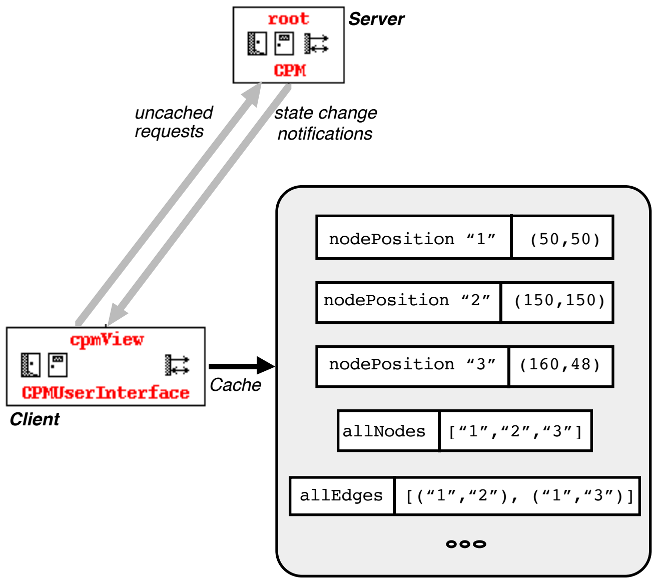

Figure 5: Cache optimization: remote requests can be reduced by maintaining a request cache in the client.

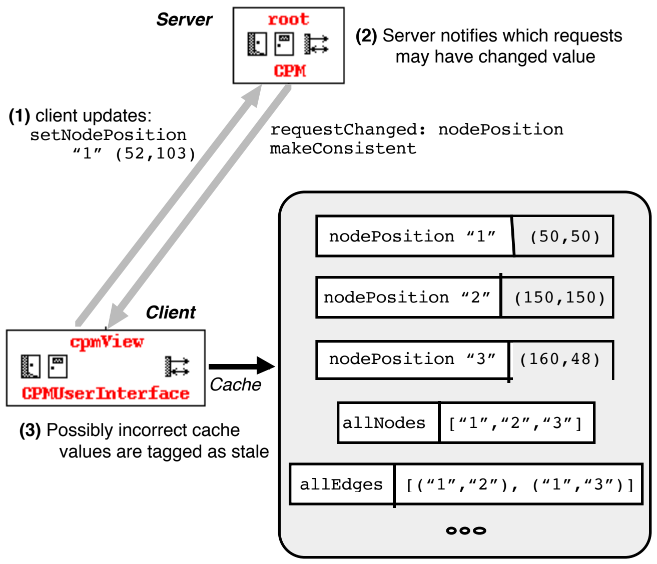
Server
root
CPM
(2) Server notifies which requests may have changed value
(1) client updates: setNodePosition "1" (52,103)
requestChanged: nodePosition makeConsistent
Client
cpmView
CPMUserInterface
Cache
(3) Possibly incorrect cache values are tagged as stale
nodePosition "1" (50,50)
nodePosition "2" (150,150)
nodePosition "3" (160,48)
allNodes ["1","2","3"]
allEdges [("1","2"), ("1","3")]
ooo

Figure 6: When a node position is changed (i.e., node "1" is moved to position $(52, 103)$), the server notifies that client that the "nodePosition" request has changed. The client tags all "nodePosition" entries in the cache as stale (shown by greying out the cache values).

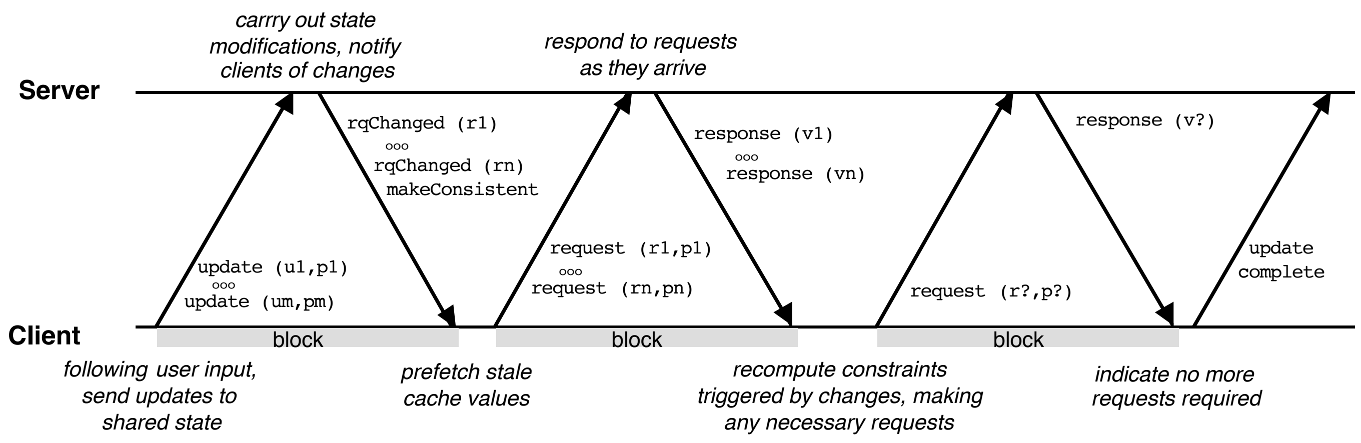
carrry out state modifications, notify clients of changes
respond to requests as they arrive
Server
rqChanged (r1)
ooo
rqChanged (rn)
makeConsistent
response (v1)
ooo
response (vn)
response (v?)
update (u1,p1)
ooo
update (um,pm)
request (r1,p1)
ooo
request (rn,pn)
request (r?,p?)
update complete
Client
block
block
block
following user input, send updates to shared state
prefetch stale cache values
recompute constraints triggered by changes, making any necessary requests
indicate no more requests required

Figure 7: Protocol for updating views including request prefetching.

of these potential changes (step 2), causing the clients to tag all cache entries for these requests as "stale" (step 3). The results attached to stale cache entries are potentially incorrect, and therefore may no longer be used.

This algorithm is simple and conservative: the fact that data flow analysis is used to predict which requests may be out of date results in all stale cache entries being identified, but can invalidate many cache entries whose values have not changed. Despite this conservatism, section 4 shows that the algorithm produces substantial savings over both local and wide area networks.

In Clock, caching is also used as the basis of an algorithm for incremental display recomputation, similar to lazy constraint evaluation [12]. The savings brought from incremental display recomputation are insignificant, however, compared to the savings in networking.

Caching can be seen as a form of data replication, in that data of interest to clients is represented locally. Request caching differs from standard replicated approaches (e.g., as used in GroupKit [10]), since only data that is actually used by clients is replicated, as opposed to the complete shared context.

3.2 Optimization 2: Request Prefetch

The motivation behind the second optimization is that we could dramatically reduce the time waiting for request results if we could predict exactly what requests were to be made, make them all at once, and then compute the updated view. This way, instead of having to wait for each request value as we make the request, we collapse all latency into a single request/response pair consisting of all requests and responses.

Predicting exactly what requests will be made in updating a view is in general uncomputable. However, a reasonable guess can be obtained by simply looking at the client's cache – in practice, chances are high that a view will make exactly the same set of requests this time as it made last time it was evaluated. Therefore, when evaluating a client's view, all stale requests made by the client are first prefetched.

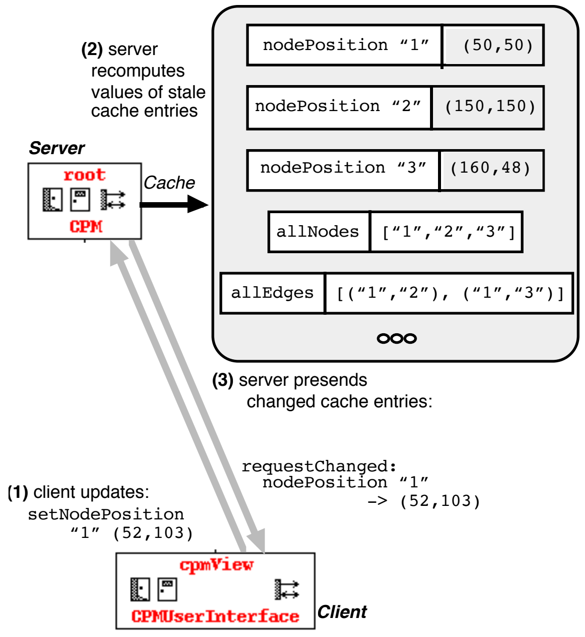
(2) server recomputes values of stale cache entries
Server
root
CPM
Cache
nodePosition "1" (50,50)
nodePosition "2" (150,150)
nodePosition "3" (160,48)
allNodes ["1","2","3"]
allEdges [("1","2"), ("1","3")]
ooo
(3) server presends changed cache entries:
requestChanged: nodePosition "1" -> (52,103)
(1) client updates: setNodePosition "1" (52,103)
cpmView
CPMUserInterface
Client

Figure 8: Presend optimization: By maintaining a request cache in the server, prefetches can be anticipated.

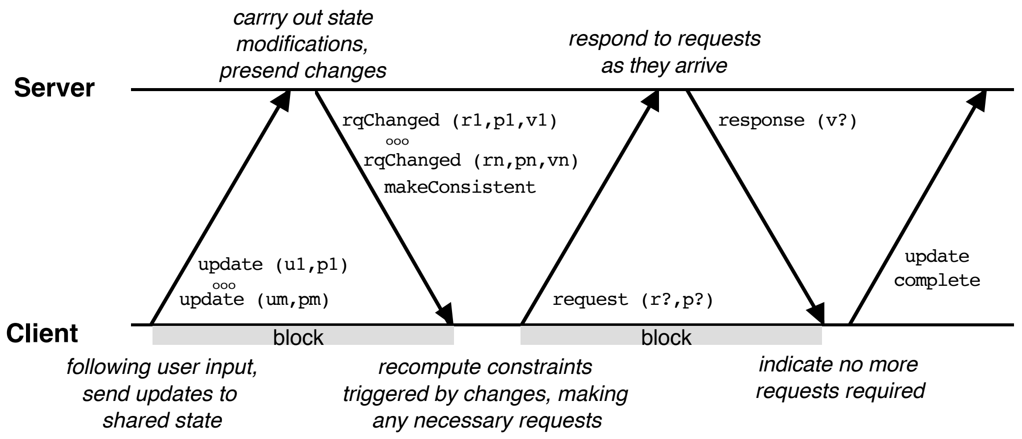
carrry out state modifications, presend changes
respond to requests as they arrive
Server
rqChanged (r1,p1,v1)
ooo
rqChanged (rn,pn,vn)
makeConsistent
response (v?)
update (u1,p1)
ooo
update (um,pm)
request (r?,p?)
update complete
Client
block
block
following user input, send updates to shared state
recompute constraints triggered by changes, making any necessary requests
indicate no more requests required

Figure 9: Protocol for updating views including request presending.

Figure 7 shows how request prefetching is performed. Assume the client has a cache with some stale entries (as in figure 6.) The client then sends the stale requests to the server in a batch. The server sends back all responses in a batch, allowing the cache to be updated. The view is computed; if any additional requests are required, they are made synchronously.

Our experiments show that in practice, the cache is a good predictor for what requests will be made in updating a view. When a user moves a network node, for example, the cache perfectly predicts what requests will be made, so that no additional synchronous requests are required. This speeds up client view updates by a factor of 1.5 to 5 over caching alone.

Request prefetching is a protocol optimized around knowledge that we are implementing a groupware application. In other application domains, it would not necessarily be the case that just because a request was made in the past, it is worth prefetching the request prior to future computation. As shown by our experiments, however, in groupware the same requests tend to be made repeatedly. Intuitively, this is to be expected – groupware programs typically involve some shared artifacts that are manipulated by different users. As the artifacts are modified, the view makes a similar sequence of calls to derive an updated display. In addition to the application formally tested in this paper, our experience has been that other applications also have this property that past requests are a good predictor for future requests.

3.3 Optimization 3: Request Presend

A refinement to the prefetch optimization of section 3.2 is *request presend*. The motivation behind presending is to collapse the three steps of request change notification, request prefetching and response into a single step. Figure 8 shows how presending works. In addition to the client, the server also maintains a request cache, recording what requests have been made, and who made them. Based on this cache, the server can compute what requests a client would make if it were to perform a prefetch operation. The server then anticipates the prefetch, and presends the information without having to be asked.

That is, when a client updates the server state, the server determines which cache entries are stale. The server then recomputes the values of stale cache entries to determine which requests have actually changed value. The server then sends change notifications, including the new request values, to the clients. The clients update their local caches, gaining the same information as would be obtained in a prefetch operation.

Figure 9 shows the updated protocol for request presending. From the protocol, it can be seen that the three messages required for request prefetching have been collapsed into one. Despite the extra work of maintaining caches at both the server and client side, the next section shows that this optimization provides no worse performance over a local area network, while providing a factor two speedup over the prefetch optimization when run over a wide area network. The factor two speedup is not surprising, given that the protocol collapses two synchronous requests into one.

Request presending can be seen as addressing the fundamental problem of MVC, that change notification messages do not adequately specify what has changed in the model, requiring the view to request more information. Presending allows the model to guess what changes are of interest to the view, and send the updated values together with the change notification message. This can be seen as an automatic realization of the Suite [3] toolkit's relaxed MVC approach, where programmers augment update messages with information as to what changed in the model.

Systems such as Garnet [8] and RendezVous [6] implement the MVC model using a more restricted form of constraints than used in Clock. These systems permit values to be constrained to other values, but not to the values of *methods*. For example, in the critical path planning application (figure 3), the view depends on the value of the *nodePosition* method: whenever *nodePosition "1"* changes value, the view must be updated. In restricted constraint systems, this dependence would have to be expressed differently, ei-

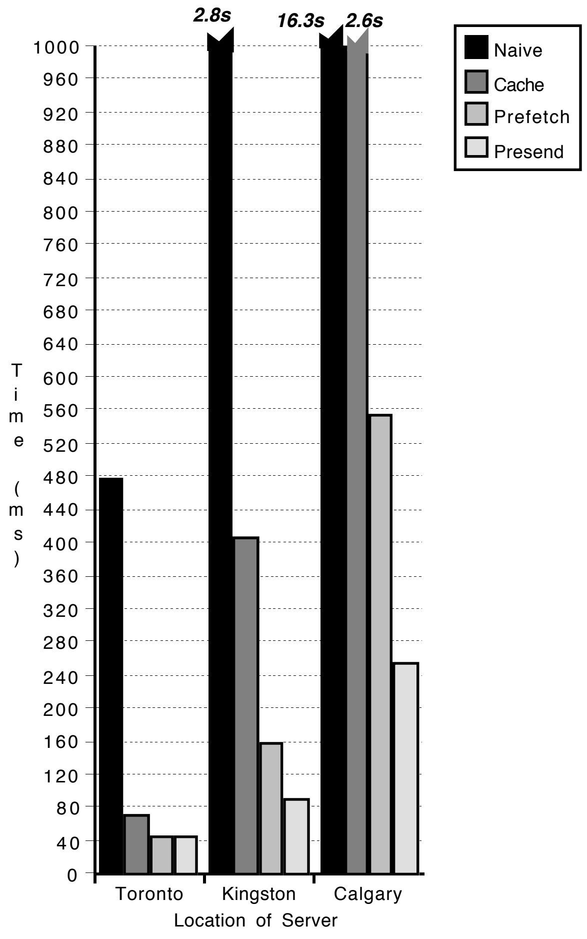

Figure 10: Timing results comparing the four algorithms with a client in Toronto and the server at three different sites.

ther by linking the view directly to a data structure containing all node positions, or by introducing intermediate value slots representing the values of each of the node positions. The former approach violates information hiding by revealing the data structure's format to the view, while the second places the burden on the programmer of having to dynamically create the intermediate objects and links to them. These restricted constraints have the advantage that the presend optimization can be implemented more easily than we have presented here, since the restriction to first order values simplifies data flow analysis used to predict what values will be needed in the future. However, the restrictions also imply a loss of notational flexibility for the programmer. Bharat and Hudson's *Doppler* algorithm [1] is also based on this restricted form of constraint.

4 EXPERIMENTAL RESULTS

In order to test the properties of the algorithms described in section 3, we carried out a series of experiments. The first experiments were designed to demonstrate the relative performance of the different communication algorithms, ranging over local and wide area networks. The second experiments were designed to show how well each of the algorithms scales as additional participants are added to the session. The experiments clearly show that presending is the best approach in both local and wide area contexts, providing both the best overall performance and the best scalability of all algorithms presented. The experiments further show that when applying presend, the example application is very responsive between Toronto and Kingston, and usable between Toronto and Calgary.

	Naive	Cache	Prefetch	Presend
Toronto	478± 18	71± 4	48± 1	46± 2
Kingston	2,820± 77	407± 24	159± 5	88± 3
Calgary	16,307±1459	2,649±306	556±85	256±45

Figure 11: The raw times underlying the graph of figure 10. The times are in milli-seconds, and express a 90% confidence interval based on 100 samples.

4.1 Algorithm Speeds over Wide Area

For each of the four algorithms and for local and wide area networks, the first experiment measures how quickly user inputs translate into view updates. To achieve this, we timed how long it takes for the system to respond to a user moving a node in the critical path network of figure 1. Precisely, we measured the real time in milli-seconds from the moment the client starts processing user input to the time the updated view has been displayed. Some concurrency control overhead common to all views is not included in the times. (This is because our experiments are not aiming to test the effectiveness of our concurrency control algorithm; in fact Clock currently uses a naive central locking scheme for concurrency control. Typical times to obtain a lock are 1 ms in the local area context, and up to 200 ms in the wide area context.) In the experiment, the client machine was always a Sun SparcStation 10 (75 MHz SuperSparc processor) located at York University in Toronto. We timed the same operation using server machines at three different locations: at York University in Toronto (SGI Indy, 174 MHz MIPS R4400), at Queen's University in Kingston, 260 km from Toronto (SGI Indy, 100 MHz MIPS R4000), and at the University of Calgary, 3,000 km from Toronto (SGI Indy, 133 MHz MIPS R4600.) Therefore, the view and controller were located in Toronto, while the model was located up to 3,000 km away. 100 samples were taken of each optimization level at each location. The local area experiments were carried out over ethernet. The links to Kingston and Calgary were via the internet. At the time of the experiments, the latency of a round-trip message was approximately 30 ms between Toronto and Kingston, and 190 ms between Toronto and Calgary.

Figure 10 shows the results of this experiment. Figure 11 shows the actual numbers obtained, and provides a 90% confidence interval. These results demonstrate that the rankings of the methods within each location are statistically significant, with the exception that the difference between the prefetch and presend algorithms is insignificant in the local area context.

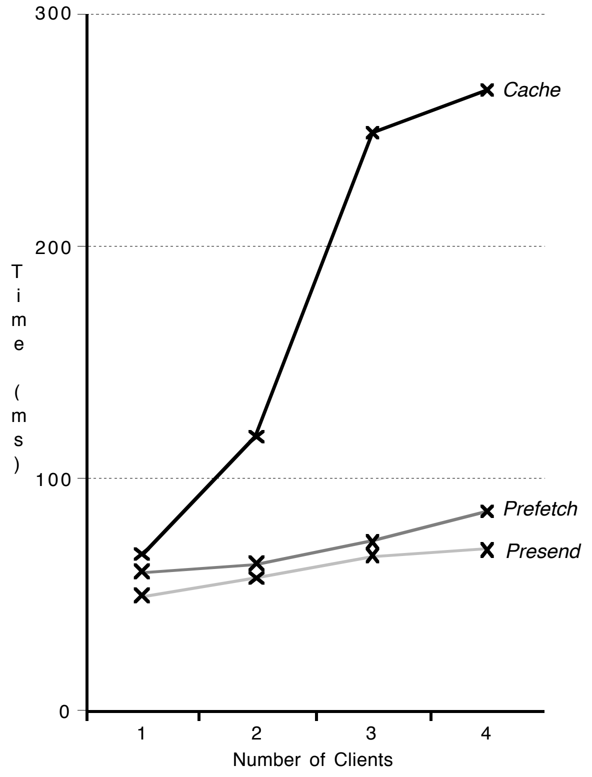

Figure 12: The time to update the views of 1 – 4 clients using the three optimizations.

The results show that in the Toronto and Kingston locations, display updates are basically instantaneous when using the presend optimization. This is a very positive result, given that the application is running over the internet between two locations separated by 260 km. The 1/4 second update time between Toronto and Calgary is no longer instantaneous, but remains usable.

4.2 Algorithm Scalability

The second set of experiments was designed to demonstrate the scalability of the different algorithms, since potentially an algorithm designed to work well with one client might behave poorly with multiple clients. These experiments show that the presend algorithm, in addition to being the fastest algorithm for one user, also posesses the best scalability.

	Naive	Cache	Prefetch	Presend
1 Client	519±62	68± 2	61±1	50±1
2 Clients	750±72	118±40	64±1	58±2
3 Clients	867±82	250±70	74±1	67±2
4 Clients	1,150±82	267±70	86±1	70±2

Figure 13: The raw times underlying the graph of figure 12. The times are in milli-seconds, and express a 90% confidence interval based on 100 samples.

The experiments were performed with the server (SparcStation 10) and all clients (SGI Indy) located at York University. We repeated the experiment of moving a node in the critical path example of figure 1, except that this time the node was moving simultaneously on the displays of up to four other participants. We measured at the server the real time elapsed from the time a user requested permission to perform an update, to the time that all clients reported the update had been reflected on their display. This time is slightly longer than that measured in section 4.1, since the measurements were at the server and therefore included the time for concurrency control arbitration. (The client and server machines were also swapped with the faster machine now acting as server instead of client.) 100 samples were taken for each of one through four clients.

Figure 12 shows the results of these experiments. (The figures for the naive implementation are not included since they dwarf those of the other algorithms.) The raw numbers are shown in figure 13, demonstrating that the differences between the algorithms are statistically significant.

The graph of figure 12 clearly shows that both the prefetch and presend approaches give excellent scalability, at least up to the four clients measured. Both algorithms scale roughly equally, with presend remaining the faster of the two.

5 RELATED WORK

Our main intention in presenting these optimizations is to demonstrate that it is possible to implement efficiently distributed groupware structured using the MVC architecture. We believe that MVC and its derivative architectures provide a natural way of implementing groupware, allowing developers to spend more time investigating groupware design and less time programming. The MVC organization for groupware was investigated in the RendezVous tool [6], using constraints as the basis of notification of change in shared data. RendezVous was implemented purely centralized, and hence suffered scalability problems.

Bharat and Hudson [1] have also examined distributed constraints as the basis for groupware implementation, and have presented the *Doppler* algorithm for optimized concurrent evaluation of distributed constraints. The primary contribution of Doppler is to maximize the concurrent processing of updates performed by different users, as opposed to the problem we have tackled, of trying to process a single user action as quickly as possible.

Another approach to implementing groupware is *full replication*, as supported (for example) by the GroupKit [10] toolkit. In this approach, every user has a complete copy of the application and its data. State updates are broadcast, and processed locally by each user's machine. Display updates in response to local changes are immediate, since all necessary data is available locally. Full replication can be more scalable than our semi-replicated approach, since there is no contention for a shared server machine. Full replication is not always possible, however: applications may depend on large data bases or media servers that reside in only one location and cannot readily be replicated, or may depend on proprietary data which may be queried but not replicated.

For some applications, processing updates to shared state may be computationally expensive, requiring special, centralized hardware. Therefore, our semi-replicated approach is a necessary alternative when full replication is not possible. We are currently investigating whether performance gains can be realized from a hybrid scheme in which some shared data is centralized and some replicated.

Another approach to reducing network communication is the use of optimistic concurrency control, such as in the *Jupiter* collaboration system [9]. In Jupiter, users maintain some shared state locally. By using the local state, immediate feedback can be provided in response to user actions without any need for network communication at all. Conflicting actions can only be detected after the fact, and are arbitrated by a server. Pessimistic concurrency control such as used in Clock gives perfect synchronization of conflicting actions, and is therefore preferable if the network can support it. Optimistic concurrency control is preferable in situations where latency is too high to support the pessimistic approach. Our optimizations can be seen as extending the distance over which the pessimistic approach is practical.

We are currently examining how information present in client caches can be exploited to permit more optimistic concurrency control, allowing clients to compute views based on current cache values. We believe this has the potential in most cases to eliminate the concurrency control overheads reported in section 4 while still preserving pessimistic semantics.

6 CONCLUSIONS

This paper has demonstrated that it is possible to efficiently implement groupware using an architecture based on MVC. Programmers can therefore develop software using a model that hides the difficult problems of network communication, distribution and consistency maintenance among different users, and still receive reasonable performance in a wide area setting. The paper has presented three algorithms for optimizing network communication, and demonstrated through timings how much improvement each optimization brings.

7 ACKNOWLEDGEMENTS

Clock and *ClockWorks* were developed by the authors, as well as Catherine Morton and Gekun Song. This work was carried out at the York Laboratory for Computer Systems Research, and was partially supported by the Natural Sciences and Engineering Research Council, the Information Technology Research Centre, and the Royal Norwegian Research Council. We would like to thank Jim Cordy of Queen's University at Kingston and Saul Greenberg of the University of Calgary for the use of their equipment in the experiments described in this paper.

REFERENCES

1. Krishna A. Bharat and Scott E. Hudson. Supporting distributed, concurrent, one-way constraints in user interface applications. In *Proceedings of the Eigth Annual Symposium on User Interface Software and Technology (UIST'95)*, pages 121–132. ACM Press, November 1995.

2. Joelle Coutaz. The construction of user interfaces and the object paradigm. In *Proceedings of ECOOP '87*, pages 121–130, 1987.

3. P. Dewan and R. Choudhary. A High-Level and Flexible Framework for Implementing Multiuser User Interfaces. *ACM Transactions on Information Systems*, 10(4):345–380, October 1992.

4. T.C. Nicholas Graham and Tore Urnes. Relational views as a model for automatic distributed implementation of multi-user applications. In *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work (Toronto, Oct. 1992)*, pages 59–66, 1992.

5. Saul Greenberg and David Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW'94)*, pages 207–217. ACM Press, October 1994.

6. Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F. Patterson, and Wayne Wilner. The Rendezvous language and architecture for constructing multiuser applications. *ACM Transactions on Computer-Human Interaction*, 1(2):81–125, June 1994.

7. Glen E. Krasner and Stephen T. Pope. A cookbook for using the Model-View-Controller interface paradigm. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

8. Brad A. Myers, Dario A. Guise, and Brad Vander Zanden. Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods. In *Proceedings of OOPSLA'92*, pages 184–200. ACM Press, October 1992.

9. David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the Eigth Annual Symposium on User Interface Software and Technology (UIST'95)*, pages 111–120. ACM Press, November 1995.

10. Mark Roseman and Saul Greenberg. Building real time groupware with GroupKit, a groupware toolkit. *ACM Transactions on Computer Human Interaction*, 3(1):66–106, March 1996.

11. T. Rüdebusch. *CSCW: generische Unterstützung von Teamarbeit in verteilten DV-Systemen*. Doctoral dissertation, Deutscher Universitäts-Verlag GmbH, Wiesbaden, ISBN 3-8244-2043-0, University of Karlsruhe, Germany (in German), 1993.

12. Brad Vander Zanden. An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Transactions on Programming Languages and Systems*, 16(1):30–72, January 1996.