Flexibly Mapping Synchronous Groupware Architectures to Distributed Implementations

Tore Urnes
Telenor Research and Development
P.O. Box 83
N-2007 Kjeller
Norway
Tore.Urnes@telenor.com

T.C. Nicholas Graham
Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada
graham@cs.queensu.ca

Abstract

Design-level architectures allow developers to concentrate on the functionality of their groupware application without exposing its detailed implementation as a distributed system. Because they abstract issues of distribution, networking and concurrency control, design-level architectures can be implemented using a range of distributed implementation architectures. This paper shows how the implementation of groupware applications can be guided by the use of *semantics-preserving* architectural annotations. This approach leads to a development cycle that involves first developing the functionality of the application in a local-area context, then tuning its performance by setting architecture annotations. The paper concludes with timing results showing that architectural annotations can dramatically improve the performance of groupware applications.

1 Introduction

In recent years, a number of software architecture styles have been proposed to support the development of synchronous groupware systems. These include *PAC** [3], *ALV* [15], *C2* [25] and the *Clock* architecture style [12]. Such design-level architecture styles allow developers to specify the high-level structure of their applications while abstracting the low-level details of distributed implementation.

Tools supporting the development of groupware applications differ in the architectural abstractions that they present to developers. Low-level toolkits expose the underlying distributed system to the developer. For example, the highly successful *GroupKit* toolkit [21] is based on a fully replicated architecture with no concurrency control. Developers must therefore be aware that different participants' views may become inconsistent [14]. *GroupKit's* low-level implementation approach is part of the reason for its great success – programmers need to know that they are programming a distributed system, but have great control over the implementation of their application.

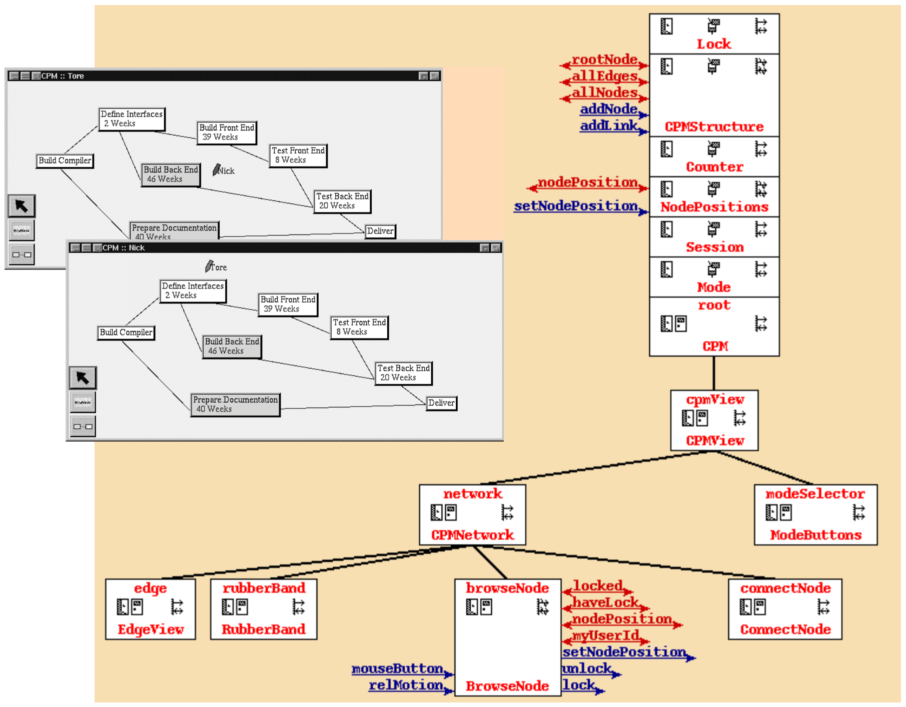

Fig.1. A critical path planning application and its design-level architecture.

High-level toolkits take the approach of automatically implementing a design-level architecture provided by the programmer. This approach simplifies the development of groupware applications by abstracting the details of networking, distribution and concurrency control, but at the cost of implementation flexibility. The *Rendezvous* toolkit [15] demonstrated that it was possible to automatically implement the high-level *ALV* architecture. *Rendezvous* provides only a purely centralized implementation, however, leading to problems with performance. Our own *Weasel* system [11] showed how high-level architectures could be mapped to a semi-replicated distributed implementation. In *Weasel*, the user interface is represented on the client machines and the functional core of the application is implemented on a server. As with *Rendezvous*, however, *Weasel* provided programmers with only one implementation strategy.

An alternative approach is to combine the advantages of high and low-level toolkits by providing both high level abstractions and the opportunity for low-level tuning. The *Suite* system [4] first demonstrated this approach by showing how a semi-replicated implementation of the model-view-controller (MVC) architecture [18] can be manually tuned by using peer to peer communication to bypass the model. The *GEN* system [19] provides shared objects as a high-level abstraction, and facilities for specifying how these shared objects should be implemented. The *Prospero* system [5] provides a *meta-object protocol*, allowing the developer to specialize the toolkit's mechanisms for managing shared data. *AMF-C* [24] provides groupware frameworks that can be customized following implementation.

In this paper, we demonstrate that it is possible to completely separate the functional design of a groupware application from the design of its distributed implementation. A design level architecture in the style of *PAC* [3] or *Rendezvous* [15] provides a conceptual framework for developing the application's functionality. This architecture is then annotated to guide its implementation as a distributed system. A toolkit provides a default implementation of the architecture, suitable for testing. To achieve production quality performance, architectures are tuned via *semantics-preserving annotations.* These annotations select between a variety of distribution styles, concurrency control methods, caching algorithms and replication strategies.

We have demonstrated this approach of separating the development of an application's functionality from the specification of its distributed implementation, using the *Clock* groupware development toolkit. Clock provides an architecture style based on layered MVC [18]. A distributed implementation of a Clock architecture is considered correct if it adheres to Clock's formal semantics [8]. As with other high-level architecture styles, Clock permits a wide range of implementations. The effect of a semantics-preserving annotation is therefore to specify to the Clock runtime system that a particular implementation is desired. Therefore, annotations do not change the functionality of the application, just its runtime performance.

Annotations can lead to dramatic improvements in the performance of groupware applications. As shown in section 4.1, the Clock implementation of a highly interactive project planning application runs with instantaneous response time, even when the participants are located in Canada and New Zealand. Over a wide area network, the annotated version of this application ran ten times faster than the automatically derived implementation.

This paper is organized as follows. We briefly introduce the *Clock* architecture style and show how this style can be used to implement a simple synchronous groupware application. We then introduce the architecture annotations that are used to control the distributed implementation of the architecture, and show the effects of applying the annotations on the runtime performance of the application.

2 The Clock Architecture Style

Clock architectures are used to design the structure of synchronous groupware applications. Like other groupware architecture styles, *Clock* architectures consist of a hierarchy of components representing the application's compositional structure. *Clock* architectures hide low-level implementation issues such as distribution policies, networking protocols and concurrency control.

Clock has been used to build a number of substantial applications, in our research group and elsewhere. These applications include a user interface design tool [2], a tool for recording design rationale [22], a multiuser web browser [9] and a multiuser video annotation tool [12].

2.1 An Example Groupware Application

To motivate how *Clock* architectures are designed, we use the example of a simple project scheduling application written in *Clock*. As shown in figure 1, the application allows multiple users to simultaneously create nodes in a critical path network, connect them, and rearrange them. The critical path through the network is shown in white. Telepointers allow people to see who else is present in the session, and what they are doing. Each participant sees the effects of other participants' actions in real time. Participants have private copies of the project toolbar, so that, for example, one person can be moving a node while another is connecting two nodes. The application is therefore relaxed WYSIWIS (what you see is what I see).

When a participant clicks on a node in the critical path, he/she implicitly locks the node so that others cannot move it. Thus, two participants can concurrently move different nodes, but cannot concurrently move the same node.

2.2 Clock Architecture for the Critical Path Application

Clock architectures are structured as trees of components. The root of the tree (the *CPM* node) implements the functional core of the application, in this case representing the structure of the project network and the critical path. A set of abstract data types (*ADT's*) is attached to the functional core, representing application data. In figure 1, ADT's implement the positions of the nodes in the project plan (*NodePositions*), the dependencies among the nodes (*CPMStructure*), and what users are currently participating in the project planning session (*Session*).

The architecture tree represents the hierarchical composition of the user interface. The root *CPM* node creates one instance of the user interface (*CPMView*) per participant. The *CPMView* is in turn composed of the project plan (*CPMNetwork*) and the project toolbar (*ModeButtons*). Components communicate via messages. For example, a *BrowseNode* responds to *mouseButton* and *relMotion* user inputs. A *BrowseNode* may make requests (e.g., using *nodePosition* to request the position of the node), and updates to change state (e.g., moving a node with the *setNodePosition* message.)

Clock architectures can be viewed as implementing a layered model-view-controller [18] structure, where the ADT's implement the model, and the components further down the tree implement the view/controllers.

To aid with concurrency control, *Clock* guarantees atomicity of input transactions. An *input transaction* describes the sequence of computation that is required to process a user input, i.e., to read in the input, modify the user interface and application state, and update the views of all users. Therefore, *Clock* implementations may process input transactions concurrently, but only if they can guarantee that the transactions will not conflict. Atomicity of input transactions provides a powerful, low-level concurrency control mechanism from which higher-level concurrency control policies can be implemented.

Clock architectures are developed using the *ClockWorks* visual editor [10]. In this editor, components can be easily added, moved, deleted and grouped into aggregate components. When *Clock* programs run, participants may enter or leave dynamically from any location on the Internet. Participants use a version of the *GroupKit* session manager [21] to enter a session. The current session information (i.e., the names and IP addresses of all participants) is automatically maintained in the *Session* ADT.

By default, *Clock* architectures are implemented using a fully centralized architecture. The complete application runs on a single machine, posting the view of each participant to his/her client machine. *Clock* architectures, however, can be implemented in a wide variety of styles, ranging from fully centralized to fully replicated, with a range of hybrid styles in-between. The next section shows how semantics-preserving architecture annotations can be used to map design-level architectures to a wide range of distributed implementations.

3 Annotations

Developing groupware applications is a challenging task. As with all interactive software, development is iterative, requiring rapid change in response to usability testing. At the same time, developers must contend with the complexities of implementing an efficient distributed system. In *Clock*, we have taken the approach that developers should be able to work first on getting the functionality of their application correct, and then on tuning the application to obtain acceptable performance. As outlined earlier, the *Clock* architecture style permits developers to implement their application without worrying about issues of concurrency control, distribution or networking. The *Clock* toolkit automatically provides a purely centralized implementation of the architecture that correctly implements the application's functionality.

This default implementation, however, typically fails to provide acceptable performance for more than a few users or over a wide-area network. In order to improve performance, the developer can place *annotations* on the architecture. These annotations give hints to the runtime system as to how the architecture should be implemented as a distributed system. Annotations are *semantics-preserving,* meaning that they are allowed to affect the performance, but not the functionality of the application. Annotations are therefore not considered to be part of the program, but rather are information to be used for runtime tuning.

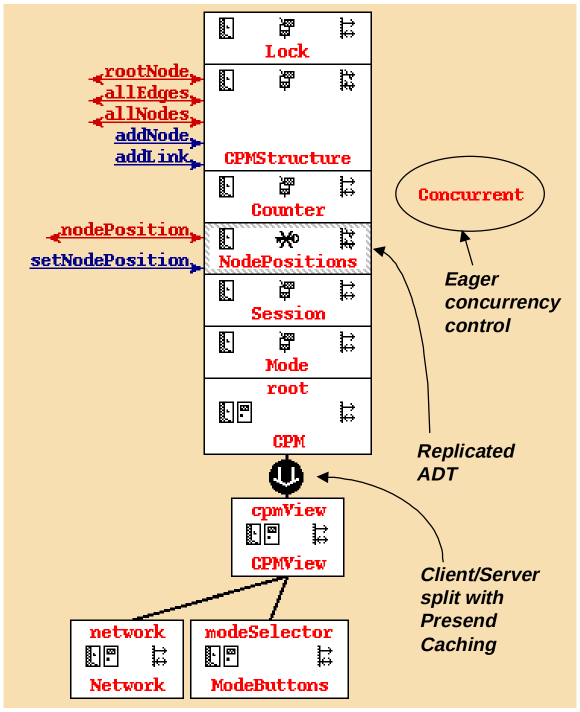

Fig. 2. The architecture of figure 1 annotated to improve performance.

In order to validate this two step approach to developing groupware applications, we have provided a set of annotations in *Clock*. Figure 2 shows the architecture of the critical path planner with annotations. Annotations are used to control the distribution of the application across multiple machines, the caching algorithms employed, replication of ADT's, and the concurrency control strategy.

The remainder of this section explains the effects of these annotations, motivating that a large space of possible implementations can be derived from a single *Clock* architecture. The timing results presented in the section 4 demonstrate that different choices of annotations can have a dramatic effect on the runtime performance of an application, particularly when running in a wide area context.

3.1 Annotations for Distribution

At runtime, an architecture leads to a set of code and data that must be distributed among the machines used to support the groupware session. One possible implementation is *purely centralized* (as used in *Rendezvous* [15]), in which all code and data is represented on a central server, while displays are posted to participants machines. Another common implementation is *semi-replication*, as used in *Weasel*

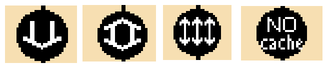

Fig. 3. Annotations specifying caching strategies: presend, prefetch, simple caching and no caching [13].

[11] and *Suite* [4]. In semi-replicated implementations, the shared context (or *functional core*) of the application is represented on a server machine, while each participant's user interface is represented on his/her own machine. Semi-replication has the advantage of supporting improved scalability [11], as the load of an increasing number of numbers of participants is distributed to the participants' own machines. However, semi-replication brings increased communication costs in that the user interfaces must communicate over the network in order to access data in the shared context. Which approach is best depends on the cost of network communication, the number of participants in the groupware session, the speed of the server, and the communication patterns of the application itself.

In Clock, annotations are provided which allow the architecture to be split between a server and client machines. As shown in figure 2, this annotation is attached to links in the architecture tree. Everything above the annotation is represented on the server machine; everything below is represented on the client machine. All update and request messages are automatically routed over the network boundary, transparently to the programmer. Developers can therefore easily experiment with different client-server split points without any reprogramming. Normally, the client-server split is made such that the shared context is placed on the server and the user interface on the client machines.

In summary, by using the client-server split annotation, developers can specify how their code is to be split among the machines being used in the groupware session. The annotations are changed simply by pointing and clicking in the *ClockWorks* environment [10]. Changing the client-server split requires no programming whatsoever.

3.2 Annotations for Caching

The primary disadvantage of semi-replicated implementation is that client machines must communicate over a network to request data from the shared context. These requests may be synchronous, requiring the client to block. Even in a local area context, clients may spend most of their time blocking [13]. Caching algorithms can be employed to reduce the number of remote requests that clients make. *Simple caching* records what requests have been made in the past, and what responses were received; if the client makes a request, the cache is consulted before making a remote request to the server. As is shown in section 4, simple caching can dramatically improve response times. More sophisticated schemes such as *prefetch* and *presend* caching [13] attempt to predict future requests and asynchronously service the requests so that the responses will be available in the cache when the requests are actually made.

These schemes carry costs, however: simple caching costs memory at the client to maintain the cache entries, and introduces extra computation in order to test whether requests are already in the cache and to maintain cache coherence. Prefetch and presend caching introduce computation to predict future requests, and can tie up the server and network in computing and communicating requests that may never be made. The correct choice of caching algorithm therefore depends on network and machine speeds and the characteristics of the application itself.

Clock provides four annotations to control the caching strategy (figure 3). These annotations correspond to providing no, simple, prefetch and presend caching. To change the caching strategy, programmers simply click on the annotation, and select the desired strategy from a dialogue box.

3.3 Annotations for Concurrency Control

Participants in a groupware session can perform actions concurrently. The system must react in some reasonable way when participants perform conflicting actions.

Numerous concurrency control schemes for groupware have been proposed. These schemes trade off three properties: speed of resolving local reads/writes, intuitiveness of handling conflicts, and burden placed on the groupware application developer. Some of these tradeoffs are summarized by Greenberg and Marwood [14]. Pessimistic schemes (such as the use of locks) determine whether it is safe to access shared data before the access is made. The overhead of these checks means that users suffer a penalty of accessing shared data even if all the shared data is available locally on their own machine. Pessimistic schemes guarantee, however, that concurrent inputs will never lead to inconsistent state or unintuitive undoing of user actions.

Optimistic schemes are based on the assumption that conflicts are rare, and that it is therefore preferable to detect and repair conflicts after they have occurred. Our own *eager* concurrency control strategy uses a transparent rollback scheme [26]. Other approaches require the programmer to provide correct *rollback* functionality to undo erroneous actions (such as in Bayou [6]) or *operation transforms* that allow updates to be applied in different orders at different sites [7]. Optimistic approaches allow local operations to proceed without consulting with other sites, but may lead to unintuitive user interface behavior when user actions are undone or transformed. Optimistic approaches may require the programmer to provide special purpose code to detect and/or repair conflicts, increasing the development effort.

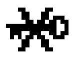

Fig. 4. Annotation specifying replication.

It is usually preferable to use a pessimistic scheme if acceptable performance can be obtained and move to an optimistic scheme as network latencies degrade. The choice of concurrency control schemes ultimately depends on the characteristics of the groupware application, the speed of the available network, and the availability of programmer time to devote to customizing concurrency control support.

In order to demonstrate that multiple concurrency control schemes can be supported within the same toolkit, we have included two algorithms in *Clock*: a locking scheme and the optimistic *eager* [26] concurrency control scheme. As shown in figure 3, programmers may select which scheme to use by clicking on the *locking/concurrent* annotation in the architecture.

3.4 Annotations for Replication

Groupware applications may be implemented with shared data represented on a central server [4,11,15], or replicated to the machines of the participants [1,14,17,21]. The primary benefit of replicating data is that response time can be improved, as local inputs can be processed without communicating with other machines. As discussed in the last section, however, replication requires sophisticated concurrency control schemes to ensure that replicas remain consistent. Additionally, replication may not be possible for some sorts of data (e.g., files or proprietary data) [20].

In *Clock*, we provide a flexible approach to replication, where developers can choose to replicate individual ADT's (figure 4). This way, the developer can choose to replicate those ADT's for which there will be a performance improvement, and centralize those for which no improvement will result. Replicated ADT's in *Clock* have no concurrency control associated with them. It is therefore incumbent on the developer to ensure that replicated ADT's have the property that applying updates in different orders will not cause consistency problems.

For example, in figure 3, the *NodePositions* ADT is replicated. This ADT keeps track of the positions of the nodes in the critical path network. The developer of this application knows that in practice, only one participant can be moving a given node at any given time. (Recall that clicking on a node locks the node.) It is therefore safe to turn off concurrency control on this ADT. In our experience, many applications contain important ADT's where concurrency control can be safely turned off, allowing replicated implementation with no overhead. As will be seen in the next section, replicating such crucial ADT's can result in dramatic speedups.

Copies of replicated ADT's are maintained on the server. The server is responsible for multicasting updates to all replicas, and maintaining a central copy of the current ADT state that can be used to allow late joiners to enter a session.

The use of annotations for replication differs from the use of annotations we have seen up to now. Annotations for distribution, caching and concurrency control are optimizations that can be applied to any architecture. By specifying those ADT's where concurrency control can be safely turned off, replication annotations allow the developer to specify application-dependent knowledge that the runtime system could not deduce. That is, the developer is specifying that in this case, the replication annotation is semantics-preserving. As argued by Edwards et al. [6], application-specific knowledge can dramatically improve the performance of concurrency control.

Replicated ADT's can be combined with centralized locks, allowing programmers to develop concurrency control strategies that are customized for their application. For example, in the critical path planner, the *Lock* ADT assigns locks for nodes. This ADT is centralized, so that requests for locks will be serialized. Manual locking of parts of the shared artifact can allow safe replication of parts of the shared context without the overhead of concurrency control.

4 Implementation

Annotations are automatically implemented by the Clock runtime system. Clock applications are divided into a server and a client part. The server is responsible for maintaining shared data, implementing concurrency control and notifying clients of changes in the shared context. Clients are responsible for implementing the user interface of a single user.

Clients include a cache and a replica store, while the server contains a concurrency control unit and a server side cache. Annotations are implemented by making a runtime decision as to how these cache, replica and concurrency control units are used.

Figure 5 shows the implementation of the critical path planner following application of the annotations of figure 3. This implementation is derived automatically by the Clock runtime system from the annotated design-level architecture. The server contains the ADT's implementing the structure of the critical path plan. The client replica store contains a copy of the *NodePositions* ADT.

The clients and server communicate via the Clock protocol [26], a networking protocol running over TCP/IP. The Clock protocol has been formally specified using the PROMELA protocol specification language, and validated through simulation and model checking [16], as well as through implementation in the Clock toolkit. Full details of the Clock protocol are provided elsewhere [26]. To give a flavour of the protocol, we consider how the protocol implements requests made by the view:

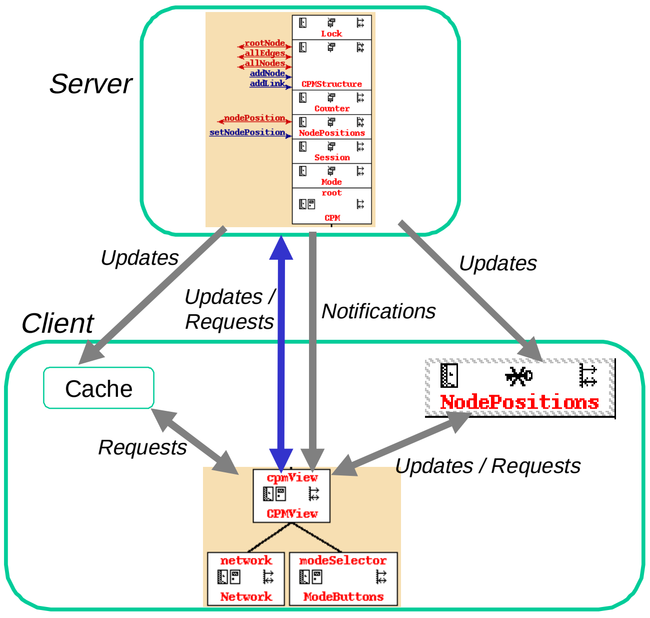

Fig. 5. Implementation architecture resulting from annotations of figure 2.

Requests to the shared context are first routed through the cache and the replica store, and only sent to the server if necessary. In practice, almost all requests are handled locally. Consider, for example, that during view computation, a client makes a request *r* with parameters *p*. The view computation unit issues a message *request(rqId,r,p)*, where *rqId* is some unique id. The Clock runtime system routes this message first to the cache; if this request has been cached (with value *v*), a response message *response(rqId,v)* is returned to the view computation unit. If the value has not been cached, the request message is forwarded to the replica store. If one of the replicated ADT's is capable of handling the request, the response is computed, and the *response(rqId,v)* message is returned. Finally, if the replica store does not contain an ADT capable of handling the request, the *request(rqId,r,p)* message is forwarded to the server, which computes and returns the *response(rqId,v)* message. The view computation code is not aware of where the response is generated; view code simply issues the *request* message, and handles the *response* when it is returned. Therefore, distribution, caching and replication decisions do not impact view computation code. Similarly, the implementation of input-handling code is not impacted by changes in the distribution architecture.

Communication from server to client is also handled through the Clock protocol. Whenever actions by one user change the shared context, the caches and replica

	1ms Latency	10ms Latency	350ms Latency
Centralized	71±2 ms	126±2 ms	992±119 ms
Semi-Replicated	1,245 ±60 ms	24,523 ±2277 ms	∞
Presend Cache	131±4 ms	256±12 ms	∞
Eager Conc. Ctl	123±5 ms	193±15 ms	543±21 ms
Replication	89±3 ms	86±2 ms	86±2 ms

Fig. 6. Results of successively applying annotations to the architecture of figure 1. The timing results show the response time of moving a node in the critical path planner, when two users are simultaneously moving nodes. Times labeled as "∞" were too long to be measured. The given ranges specify a 90% confidence interval based on 275 samples.

stores on other clients may become out of date. The server asynchronously sends updates to the other clients. The level of detail provided by these updates depends on what form of server-side caching has been selected [13].

4.1 Effect of Applying the Annotations

We now consider how the use of annotations can greatly improve the performance of groupware applications. Figure 6 shows the results of successively applying annotations to the architecture of figure 1. These numbers were obtained through a set of experiments on both local and wide-area networks. All experiments used TCP/IP over the standard Internet. The experiments measured the response time of moving a node in the critical path planner in a two user session where both users are simultaneously moving nodes. The response time is defined as the time from which the user performs an input action (i.e., moving the mouse) to the time at which the display is updated. The local area experiments used three PC's, PII 300MHz, running Linux, with a round-trip latency of 1 ms between the machines. The first wide area experiments involved PC clients at Queen's University, and a SparcStation 10 server located at York University, with a 10 ms latency between the server and client machines. (Queen's and York Universities are separated by 250 km.) The second set of wide area experiments used two PC clients located at Queen's University, Kingston, Canada, and a Sun Ultra 1 server located at the University of Wellington, New Zealand; the round-trip latency between client and server was 350 ms.

As shown in figure 6, the default centralized implementation gives acceptable performance in the local area context, but slows in the wide area context to an unusable 1-second response time. Simply performing a client/server split gives unacceptably poor performance, even in the local area context (1.25 seconds response time.) Adding presend caching gives a substantial speedup, bringing response times to usable levels, except in the widest area context. Moving to eager concurrency control brings statistically insignificant improvement in the local area context. Over wide area, the overhead of obtaining locks is higher, and therefore eager concurrency control has a more significant effect. Finally, replicating the *NodePositions* ADT brings a significant speedup in both local and wide areas.

5 Analysis

The timing results presented in the last section show that annotating an architecture can result in dramatic speedup of applications. In the wide area case, the annotations made the difference between unusable and instantaneous response time, even in the case when two users are simultaneously interacting with the application. The best implementation is hybrid, combining centralized and replicated data and selective application of concurrency control. Such a hybrid implementation would be hard to derive automatically, but was easy to derive through architectural annotations.

According to Shneiderman [23], humans perceive response times of approximately 50 – 150 ms as instantaneous. The annotated implementation achieved response times in this range, even with participants as far separated as Canada and New Zealand. We have obtained similar results from other applications we have developed in *Clock* [26], such as a collaborative video annotator [12] and the *GroupScape* web browser [9].

We have shown that annotations allow programmers to develop applications without being concerned with their distributed implementation, then later tune the application to its distributed context. One possible criticism of design-level architectures is that they may be biased towards a client/server implementation, where the shared context is represented on the server machine. Even when components are replicated, a centralized component is involved in multicasting the client updates. In fact, annotations allow us to move very close to a pure replicated model if we choose. Replicated implementations typically must retain some central component to help in concurrency control, maintain session information, and deal with late joiners. If all ADT's from the shared context are replicated and caching is turned off, the server devolves to simply maintaining session information, dealing with late joiners, and multicasting updates between client machines. Most pure replicated implementations handle these functions through some centralized service.

The approach of using annotations has allowed us to experiment with a range of implementation algorithms to gauge their effectiveness. We have found presend caching [13] to be generally highly successful despite the burden it places on the server machine. Optimistic concurrency control [26] is generally preferable to locking, although in the local area context, its overheads may be as large as its benefits. It is important to be able to turn off these optimizations, however, as they are based on heuristics which may not be appropriate to every application.

The two-stage approach of first implementing a design-level architecture and then tuning it works well. We have had little difficulty tuning architectures after their completion, and in gaining significant speedups through this tuning.

6 Conclusions

In this paper, we have argued that it is possible to separate the design-level architecture for synchronous groupware systems from its ultimate implementation. We have shown how a single design-level architecture can be mapped to a wide range of implementation architectures, ranging from centralized to replicated. We have argued that in practice, hybrid architectures combining some centralized and some replicated data may be the most effective.

To validate these ideas, we presented our experiences with the *Clock* groupware development toolkit, in which design-level architectures can be optimized through semantics-preserving annotations. We showed how the application of annotations can lead to a hybrid implementation architecture, tuned to the specific properties of the groupware application and the hardware environment in which it is running.

The major shortcoming of our approach is that the annotations do not permit applications to be tuned on a per-client basis. We are continuing research into generalizing the annotation concept, developing hybrid concurrency control and caching algorithms, and dynamic reconfiguration of implementation architectures.

Acknowledgements

The work described in this paper was supported by the Natural Science and Engineering Research Council. Thanks also to Saul Greenberg and Mark Roseman for permitting us to use the *GroupKit* session manager [21].

References

1. J. Begole, C.A. Struble, C.A. Shaffer, and R.B. Smith. Transparent Sharing of Java Applets: A Replicated Approach. *UIST '97*, pages 55-64, 1997.
2. J. Brown and S. Marshall, Sharing Human-Computer Interaction and Software Engineering Design Artifacts. In *Proc. OZCHI'98*, Dec. 1998.
3. G. Calvary, J. Coutaz, and L. Nigay. From Single-User Architectural Design to PAC*: a Generic Software Architecture Model for CSCW. In *Proc. CHI '97*, pages 242-249. ACM Press, 1997.
4. P. Dewan and R. Choudhary. A High-Level and Flexible Framework for Implementing Multiuser User Interfaces. *ACM TOIS*, 10(4):345-380, Oct. 1992.
5. P. Dourish. Consistency Guarantees: Exploiting Application Semantics in a Collaboration Toolkit. In *Proc. ACM CSCW*, 1996.
6. W.K. Edwards, E.D. Mynatt, K. Petersen, M.J. Spreitzer, D.B. Terry, and M.M. Theimer. Designing and Implementing Asynchronous Collaborative Applications with Bayou. In *Proc. ACM UIST '97.* ACM Press, 1997.
7. C.A. Ellis and S.J. Gibbs. Concurrency Control in Groupware Systems. In *Proc.*

SIGMOD '89, pages 399-407. ACM Press, 1989.

8. T.C.N. Graham. *Declarative Development of Interactive Systems.* Volume 243 of Berichte der GMD. Munich: R. Oldenbourg Verlag, July 1995.
9. T.C.N. Graham. GroupScape: Integrating Synchronous Groupware and the World Wide Web. In *Proc. INTERACT'97,* pp. 547-554, July 1997.
10. T.C.N. Graham, C.A. Morton, and T. Urnes. ClockWorks: Visual Programming of Component-Based Software Architectures. *J. Visual Lang. & Comp.,* 7(2):175-196, June 1996.
11. T.C.N. Graham and T. Urnes. Relational Views as a Model for Automatic Distributed Implementation of Multi-User Applications, *CSCW'92,* 59-66, 1992.
12. T.C.N. Graham and T. Urnes. Integrating Support for Temporal Media into an Architecture for Graphical User Interfaces. In *Proc. ICSE '97,* 1997.
13. T.C.N. Graham, T. Urnes, and R. Nejabi. Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware. In *Proc. UIST '96,* pp. 1-10, 1996.
14. S. Greenberg and D. Marwood. Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface. *CSCW '94,* 207-217, 1994.
15. R.D. Hill, T. Brinck, S.L. Rohall, J.F. Patterson and W. Wilner. The Rendezvous Language and Architecture for Constructing Multi-User Applications. *ACM TOCHI,* 1(2):81-125, June 1994.
16. G.J. Holzmann. Design and Validation of Computer Protocols. Prentice Hall, 1990.
17. T. Kindberg, G. Coulouris, J. Dollimore, and J. Heikkinen. Sharing Objects over the Internet: the Mushroom Approach. In *Proc. IEEE Global Internet '96,* 1996.
18. G.E. Krasner and S.T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *JOOP,* 1(3):26-49, Aug./Sept. 1988.
19. T. O'Grady. Flexible Data Sharing in a Groupware Toolkit. M.Sc. Thesis, Department of Computer Science, University of Calgary, 1996.
20. J.F. Patterson, M. Day, and J. Kucan. Notification Servers for Synchronous Groupware. In *Proc. ACM CSCW '96,* pages 122-129. ACM Press, 1996.
21. M. Roseman and S. Greenberg. Building Real Time Groupware with GroupKit, A Groupware Toolkit. *ACM TOCHI,* 3(1):66-106, March 1996.
22. M. Sage and C. Johnson, Pragmatic Formal Design: A Case Study in Integrating Formal Methods into the HCI Development Cycle. In *Proc. DSVIS'98,* 1998.
23. B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction, Third Edition.* Addison Wesley, 1998.
24. F. Tarpin-Bernard, B. David and P. Primet, Frameworks and patterns for synchronous groupware: AMF-C approach, *EHCI'98,* 1998.
25. R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead Jr., J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Trans. SW Eng.,* 22(6), June 1996.
26. T. Urnes. *Efficiently Implementing Synchronous Groupware.* Ph.D. Thesis, Department of Computer Science, York University, October 1998.