Quality Analysis of Distribution Architectures for Synchronous Groupware

T.C. Nicholas Graham
School of Computing
Queen's University
Kingston, Canada, K7L 4L5
graham@cs.queensu.ca

W. Greg Phillips
Elec. and Comp. Engineering
Royal Military College
Kingston, Canada, K7K 7B4
greg.phillips@rmc.ca

Christopher Wolfe
School of Computing
Queen's University
Kingston, Canada, K7L 4L5
wolfe@cs.queensu.ca

Abstract

This paper identifies a set of distribution architectures for the development of synchronous groupware and provides an analysis of their quality attributes. The architectures and their quality attributes provide insight on how to structure the implementation of synchronous groupware applications, providing developers with precise guidance on the trade-offs between various implementation techniques. In contrast to many proposed architectures for groupware, these architectures have been synthesized through analysis of successful groupware systems whose properties are well-understood.

1. Introduction

Over the last two decades, numerous distribution architectures for synchronous groupware systems have been presented in the research literature [8, 20, 11]. However, there have been few publications addressing the properties of these architectures as deployed in production-quality systems. This makes it difficult for developers to assess which architecture would be most appropriate for what tasks. This relative lack of evaluation is not completely surprising, as until recently very few synchronous groupware applications were widely deployed or used.

In recent years, however, numerous synchronous groupware applications have come into wide use in the areas of communication, online gaming and electronic meetings. This presents us with the opportunity to analyze what implementation designs have worked in practice and to determine their properties.

By its nature, architectural design embodies trade-offs between quality attributes such as performance, security and availability. Ideally, developers would be able to consult a guide detailing the qualities of a range of well-understood architectures, allowing them to choose the approach that best matches their application's requirements. The architectures presented in this paper represent one small step towards this goal.

In this paper, we identify five distribution architectures for synchronous groupware. These architectures all address the basic problem of how to allocate components of the groupware system to different computational nodes, and how those components should communicate.

This paper makes two contributions:

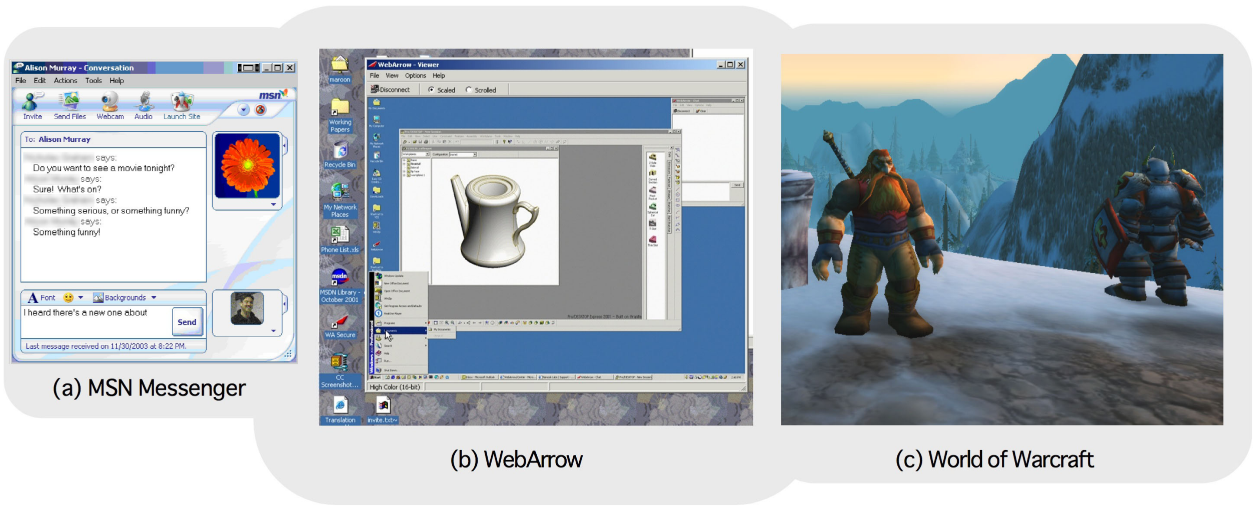
- We have identified an interesting set of distribution architectures used in commercial groupware systems. While some of these approaches have been presented in the past by the research community, this is the first comprehensive attempt to examine architectures that have been deployed in commercial systems and used by thousands of people. Our criterion for the inclusion of a distribution architecture in this presentation is that we needed to find instances of its use in at least three systems that have been deployed and widely used.

- We have analyzed these architectures with respect to a broad set of quality attributes, identifying the tradeoffs inherent in choosing a particular distribution architecture. This differs from earlier analyses, which focus primarily on performance [14]. As we shall see, when other quality requirements are considered, the architecture with the best performance may not always be the correct choice.

We first summarize what synchronous groupware is, then discuss quality attributes of interest in evaluating groupware implementations. Finally, we present the set of distribution architectures that we have identified, and analyze them with respect to these qualities.

2. Synchronous Groupware

Synchronous groupware allows people to communicate or collaborate in real-time. After many years of limited



[redacted] says:
Do you want to see a movie tonight?
[redacted] says:
Sure! What's on?
[redacted] says:
Something serious, or something funny?
[redacted] says:
Something funny!

I heard there's a new one about

Last message received on 11/30/2003 at 8:22 PM.

Figure 1. Examples of synchronous groupware applications

commercial success, in recent years synchronous groupware has become widely used in at least three application areas:

- *Communication Tools:* Tools such as MSN Messenger and Skype allow people to communicate at a distance, either by textual chat, or more recently through voice over IP and video. These tools have subscriberships in the millions, and for many people, have become as important as email and the telephone.

- *Multiplayer Games:* Games that allow people to collaborate and compete have become enormously popular. Services such as GameSpy and Xbox Live allow millions of people to connect and play together. Massively multiplayer games such as World of Warcraft allow thousands of people to meet, socialize and adventure together in the same persistent virtual world.

- *Electronic Meeting Tools:* Tools allowing remote presentations and online meetings are becoming increasingly used. Examples of commercially successful products include WebEx, GoToMeeting, and WebArrow.

Representative examples of these tools in action can be found in figure 1. *MSN Messenger* is a ubiquitously deployed instant messaging tool featuring text, video and voice communications. *WebArrow* is an electronic meeting system, allowing participants to work together on shared documents. *World of Warcraft* is a massively multiplayer online game allowing thousands of players to socialize, collaborate and compete in a shared virtual world.

3. Qualities of Synchronous Groupware Systems

The design of software systems involves managing tradeoffs. One solution may reduce bandwidth costs while impacting usability. Another may give great security, at the cost of deployment headaches for the system's users. To make informed choices, designers must first specify the requirements of their system, and rank which of its desired qualities are the most important. *Quality attributes* provide a vocabulary for discussing these requirements [3]. There has been considerable work in analyzing the performance properties of distribution architectures for groupware, both informally [8, 11, 20] and formally [14]. However, the CSCW field has been weak in identifying and clarifying design tradeoffs over the wider spectrum of quality attributes such as performance, usability, security, availability and scalability.

Our first step is to identify quality attributes that are particularly useful for discussing design tradeoffs in synchronous groupware systems. We will then (in section 5) be able to show the tradeoffs between distribution architectures in terms of these quality attributes.

3.1. Performance

Synchronous groupware applications are normally deployed over the Internet, and so suffer from the problems of network latency, limited network bandwidth, and lossy message delivery. These problems all result in performance penalties.

There are numerous measures that capture aspects of the performance of groupware systems. Some of the more critical are:

- *Feedback time, feedthrough time:* Feedback time measures the time from a user performing an action to seeing the result of that action. Feedthrough time measures the time from a user performing an action to other users seeing the result.

- *Frame rate:* Applications such as video conferencing and 3D games are often judged by the frequency with which they update the display.

- *Bandwidth consumption:* Some distribution models require large available bandwidth to operate. This may have consequences on the expense of operating the groupware application, or on the ability to run it at all in some environments. High bandwidth requirements can negatively impact feedthrough time and frame rate.

3.2. Usability

The usability of a groupware system has an enormous effect on its success [1]. While not all usability issues are architectural in nature, some are heavily influenced by architectural choices. Among these are:

- *Ease of deployment:* This measures how difficult it is for users to configure their environments so that they can take part in a groupware session. Issues include whether users need to install software or involve the local IT department.

- *Fidelity:* Groupware applications provide participants with a view of some shared context. The degree to which that view is a correct and up-to-date representation of the shared context is the application's *fidelity.*

- *Collaboration awareness:* Groupware applications vary significantly in their support for collaboration, particularly in the restrictiveness of their coordination models, whether they support pure or relaxed WYSIWIS[1] views, and how well they provide group awareness. The degree to which groupware applications are designed for collaboration has been termed *collaboration awareness* [8].

3.3. Security

Security refers to the preservation of confidentiality and integrity of the information manipulated in a groupware application, including the apparent identities of the users. Security is of great importance to corporate users of groupware, who are concerned about theft of sensitive data or eavesdropping on conferences. Perhaps surprisingly, it is also important in multiplayer games: players have shown astonishing tenacity in their pursuit of ways of cheating [28], particularly in the case of massively multiplayer games, where in-game assets often possess significant real-world value.

[1]WYSIWIS: *What You See Is What I See*

3.4. Availability

Availability measures the percentage of time that the groupware system is up and available for use. In the commercial world, people expect groupware applications to have a level of availability similar to that of their telephone. Online games are offered as services, and therefore have high uptime requirements from their player base. Availability is normally measured via this equation:

$$\frac{\text{mean time to failure}}{\text{mean time to failure} + \text{mean time to repair}}$$

This implies that availability may be improved either by increasing mean time to failure, or by decreasing mean time to repair.

3.5. Scalability

Some groupware applications have stringent scalability requirements. For example, a remote presentation system may need to accommodate hundreds of attendees. A massively multiplayer online game may need to support thousands of concurrent players in the same virtual world. The infrastructure of an instant messaging system may need to support millions of users.

4. The Workspace Notation

In section 5 we introduce distribution architectures supporting the development and distributed implementation of synchronous groupware systems. To present the architectures, we use the Workspace Architecture Notation [21], a precise notation explicitly designed to express the architectures of groupware systems. The subset of the notation used in this paper is shown in figure 2 and described below.

Workspace architectures are deployed on behalf of *users* and consist of *components*, *connectors* between these components, and the *nodes* on which the components reside.

There are three kinds of component in the notation. An *actor* has its own thread, and is capable of initiating activity. *Reactors* react to their environment; they do not initiate activity, but respond to stimuli from other actors or reactors. *Stores* are passive components used to store data.

Components may be anchored on *nodes*, which represent computational platforms such as computers.

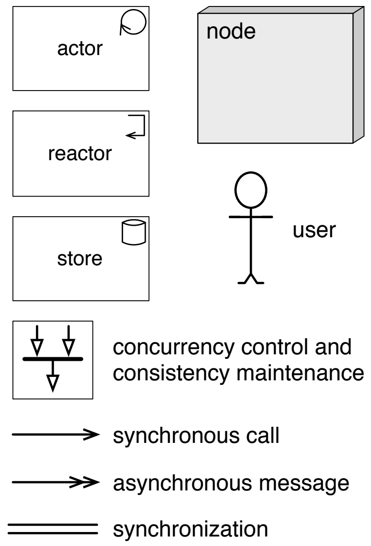


Figure 2. The workspace notation

Components may be connected by connectors. A *call connector* supports synchronous calls. A *subscription connector* permits asynchronous messages to be sent from one or more components to zero or more other components. A *synchronization connector* ensures that two components are observationally equivalent (i.e., synchronized.) All three connector types may span node boundaries.

Finally, a *concurrency control and consistency maintenance* unit (or *CCCM*) is responsible for ordering operations to provide an application-dependent level of consistency. CCCMs may be connected to one another to allow concurrency control over actions on a set of components. Where architectures include replicated data, the CCCM components enact the required replica consistency maintenance algorithms.

5. Distribution Architectures

In this section, we present five distribution architectures for synchronous groupware. The architectures address the core problems of sharing data between session participants, allowing participants to communicate, and connecting participants together. To the best of our knowledge, these architectures capture the best practice of development of synchronous groupware; each has been observed in the development of several real groupware systems that are in daily use by thousands of people.

We identified these architectures by examining the implementation design of existing systems. Our knowledge of the designs was gained from a variety of available sources, including published papers, direct observation of the systems' behaviours, and technical presentations by the systems' developers.

	Performance	Deployability	Fidelity	Collab'n Awareness	Security	Availability	Scalability
Centralized core, thick client	◔	×	◔	√	√		◔
Generic thin client	×	√	◔	×		×	
Centralized mixer with broadcaster	√		◔				√
Replicated input broadcasting	◔	×		√	×	√	×
Replicated state synchronization	◔	×		√	×	√	×

Figure 3. Distribution architectures for the development of synchronous groupware

Figure 3 lists these architectures and shows the quality attributes of greatest relevance to each. We now discuss each distribution architecture in detail.

5.1. Centralized Core, Thick Client

Figure 4 shows the *centralized core, thick client* distribution architecture. This architecture is particularly useful for the development of high-security, collaboration-aware applications, and can be tuned for high scalability.

This architecture places the application's functional core on a server and allows multiple "thick clients" to connect to it. The functional core takes care of all issues related to the application's semantics, while the thick client is responsible for all aspects of presentation.

The client and functional core operate asynchronously. The client sends non-blocking messages to the functional core, either to notify it of the user's actions or to request information. The functional core provides updates to the client in response to other users' actions or to computations taking place in the functional core itself.

Each client contains a local context whose purpose is to cache information from the functional core and to support client-side prediction. The local context is used to reduce the frequency with which the client is required to wait for information from the functional core.

All messages to the functional core flow through a CCCM unit whose purpose is to serialize incoming messages, ensuring that messages are processed in the order in which they arrive.

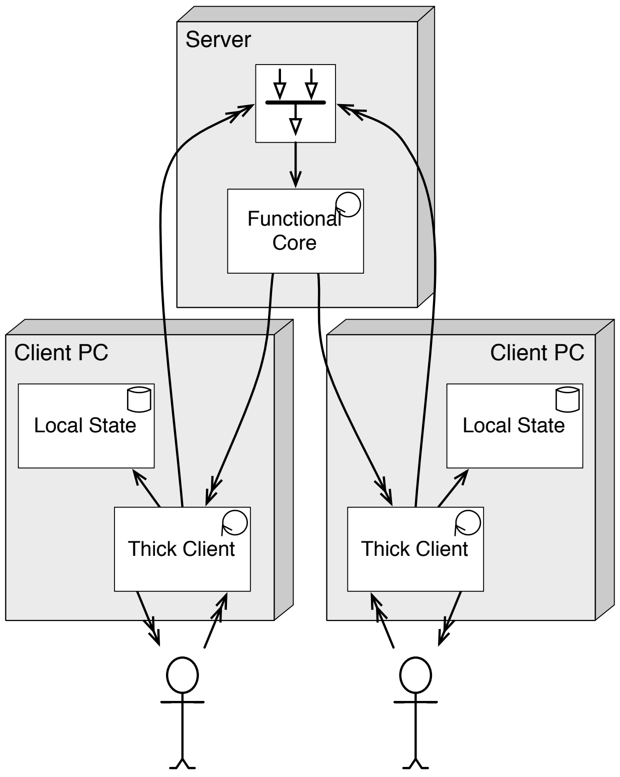


Figure 4. Centralized core, thick client

5.1.1 Examples

This architecture is used widely in multiplayer video games, ranging from games that support on order of ten players (e.g., Half-life [5] and Halo 2 [7]) to massively-multiplayer games supporting thousands of concurrent players (e.g., EverQuest[2], Lineage [17] and Star Wars: Galaxies [22]).

The operation of this architecture is well illustrated by *EverQuest*. There, the server is responsible for all gameplay decisions, including positioning of players' units, resolution of combat events, activities of AI-controlled characters and inter-player trades. The client is responsible for managing user interactions, the 3D rendering and lighting of game objects, animation, and physical effects of such as falling or skidding. Thus, the client performs significant computation, while the server retains control over the actual gameplay.

Interaction between the client and server is via high-level messages. For example, if the client wishes to inform the server that the player has moved, it sends a message of the form

move(id, pos)

[2]EverQuest's protocol has not been published, but has been re-engineered by the game's fans, and is documented in the source of the ShowEQ utility [23].

Similarly, if the server wishes to inform the client of the movement of a non-player character, it sends a message of the form

move(id, pos, velocity, heading, animation)

which specifies that the object is to be moved to the given new position and the given animation is to be played.

In the latter example, the velocity and heading are useful for client-side prediction of the current location. Using the *dead reckoning* technique [16], the client extrapolates positions of the game's mobile entities from their last reported position, heading and velocity. This allows positional updates to be transmitted less frequently, improving bandwidth consumption at the cost of some degree of fidelity. (Lee states that 70% of bandwidth use in a massively multiplayer game is related to movement [17].)

EverQuest is supported by 1,500 servers located in a managed network operations centre, and allows up to 2,500 players to simultaneously interact in a single instance of the EverQuest world [15]. In contrast, the *Halo 2* game [7] locates the functional core on one of the client computers, allowing small groups to play without requiring any central infrastructure. Both games have excellent availability; EverQuest through server redundancy, and Halo 2 through a mechanism for migrating the functional core in case of failure of the hosting client.

5.1.2 Qualities and Tradeoffs

The centralized core, thick client architecture is appropriate when *security* is of concern, since key application data and application logic are controlled by the server. The client may be responsible for significant computation, but does not have direct access to sensitive data. This is the fundamental reason why this architecture is used for multiplayer games.

This architecture has the *scalability* advantage that clients communicate only with one other node (the server), reducing *bandwidth requirements* as compared to replicated approaches (see sections 5.4 and 5.5.) However, the centralized functional core can itself become a scalability bottleneck; this is mitigated in massively multiplayer games by distributing the load of the functional core over several physical servers [4].

The server forms a single point of failure, possibly compromising *availability*. This problem can be addressed by failover mechanisms such as redundancy in EverQuest's server farm and migration of the functional core in Halo 2.

Feedback time is generally poor with this architecture, since all user inputs have to flow to the server before the client can be updated. In practice, the use of the Local State to perform client-side prediction ameliorates this problem, possibly at the cost of *fidelity*.

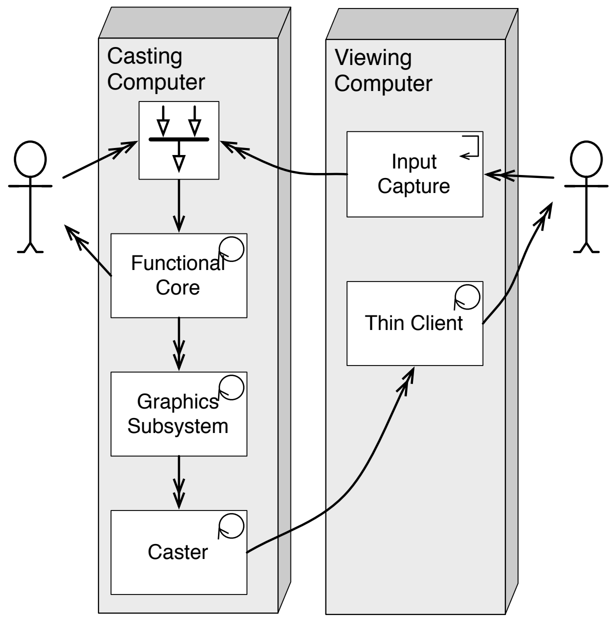


Figure 5. Generic thin client

5.2. Generic Thin Client

Figure 5 shows the *generic thin client* distribution architecture. This approach is widely used in electronic meeting systems such as WebEx [27], GoToMeeting [10] and WebArrow [26], due to its ease of *deployment*.

In this architecture, one meeting participant runs an application (or set of applications) to be used collaboratively. The other participants run a *generic thin client* that provides a mirror of the application. A "caster" (view broadcaster) hooks into the graphics subsystem on the casting computer and passes along changes in the application view to the thin client, which then displays the changes. The thin client is generic in the sense that it requires no knowledge of the underlying application that is being displayed; it simply processes display updates.

User inputs on the viewing computers are trapped by an input capture component and forwarded to the casting computer.[3] On the casting computer, a CCCM unit serializes the inputs of all users, typically filtering them according to a floor-control scheme, and delivers the inputs to the application.

5.2.1 Examples

WebEx [27] uses the generic thin client architecture to support application sharing within electronic meetings. With application sharing, any number of participants can view an application hosted on one participant's computer. A special *mirror driver* is installed on the casting computer that intercepts changes to the display device, allowing the caster unit to send display updates to each of the thin clients for display. Since the literal contents of the application's window are being broadcast, WebEx is restricted to being pure WYSIWIS.

Only one participant can control the application at a time. WebEx uses a floor control interface allowing the meeting's moderator to determine which participant's inputs will be injected into the application. A CCCM unit is responsible for sending the controlling participant's inputs to the application on the casting computer.

Other popular applications of this form include WebArrow [26] and GoToMeeting [10].

5.2.2 Qualities and Tradeoffs

The primary advantage of this architecture is *deployability*, since the application that is being shared is required only on the casting computer, not on the viewing computer. This allows a group to collaborate using standard, collaboration-transparent applications, with only the additional installation of the thin client on participants' computers plus the caster on the casting computer. Commercial tools based on this architecture typically bundle the caster and thin client into a small web-deployable download.

This approach provides excellent *feedback time* and *feedthrough time* for the user of the casting computer, since all computations take place locally and the floor-control concurrency control imposes no overheads. For viewers, the situation is worse, as inputs must travel to the casting computer and outputs must return to the viewing computers in the form of screen images, normally as compressed bitmap data. For applications involving rich graphics and animation, this overhead may be significant.

When bandwidth is at a premium, the viewers' *frame rate* is typically reduced, so updates are propagated more slowly. This may be ameliorated somewhat by use of the *centralized mixer and broadcaster* architecture (see section 5.3).

Availability is a concern with this architecture since the casting computer is a single point of failure: if it becomes unavailable, the collaborative session cannot continue. For high availability, it may be better to consider replicated architectures (sections 5.4 and 5.5) or a centralized architecture with managed hosting (section 5.1).

5.3. Centralized Mixer with Broadcaster

Figure 6 shows the *centralized mixer with broadcaster* distribution architecture. This architecture addresses situations where inputs from a set of session participants need

[3]The input capture component may need to retrieve information from the thin client in order to, e.g., compute relative mouse scaling.

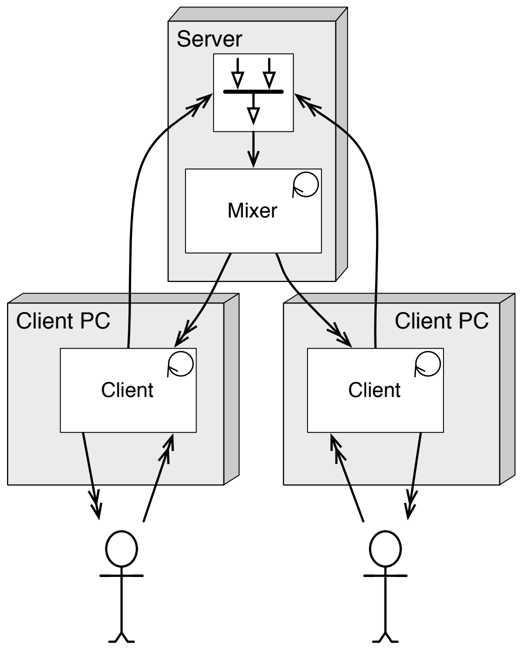


Figure 6. Centralized mixer with broadcaster

to be mixed in some way and then rebroadcast to other participants. The mixed data may be identical for all or may be customized for individual participants or groups of participants. This architecture is commonly found in communication tools such as Skype [2], MSN Messenger [18] and TeamSpeak [25], and in tools based on the generic thin client architecture (section 5.2.)

As the figure indicates, one or more clients send data to the server computer, where a CCCM unit serializes the data (if necessary) and the data is passed to the mixer. The mixer then broadcasts one or more streams of mixed data to the clients.

A common variant of this architecture locates the mixer on one of the client PCs.

5.3.1 Examples

The *Skype* internet telephony tool [2] uses the centralized mixer with broadcaster distribution architecture. Each participant in a call runs a Skype client which encodes the participant's voice and plays back the conversation of the other participants. The clients send their participants' voice data to a mixer, located on a "super-node", which is simply the computer of some Skype user.[4] The mixer creates and sends a custom sound stream to each participant. This stream includes the voices of all other participants; the participant's own voice is omitted to prevent the sensation of echo.

[4]The "super-node" owner is not necessarily involved in the session.

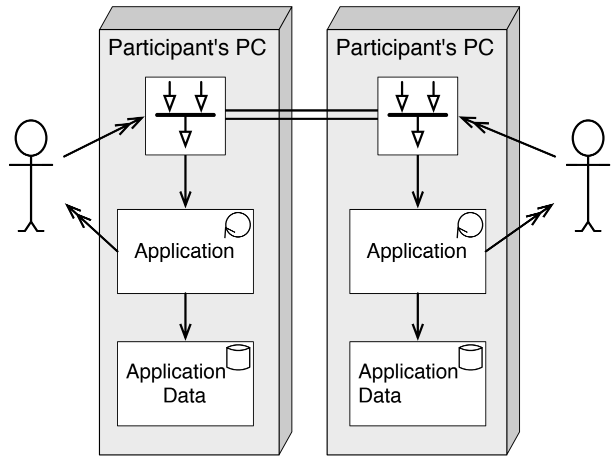


Figure 7. Replicated input broadcasting

The *WebArrow* electronic meeting system [26] uses this architecture for all its supported communication modalities, including voice, text chat and screen data. Screen data is sent by the casting computer of the generic thin client architecture (see section 5.2) to the mixer, which is located on a managed server. The mixer provides a stream of screen data to each of the viewers, possibly providing streams with different frame rates in order to account for varying bandwidth and processing power available to each client.

5.3.2 Qualities and Tradeoffs

The main benefits of the centralized mixer with broadcaster architecture are reduction of *bandwidth consumption* and increased control in the tradeoff between *fidelity* and *feedthrough time*.

The architecture reduces bandwidth consumption when compared to replicated approaches (sections 5.4 and 5.5), since each client communicates with only one other node, rather than requiring each peer to communicate with each other peer. As with other centralized architectures (section 5.1), this aids *scalability*, as bandwidth use grows linearly with the number of clients, not quadratically. (This is only true as long as the server's bandwidth and processing power does not become a bottleneck.) However, the architecture may have a negative effect on *availability*, since the centralized mixer represents a single point of failure.

Relatedly, the architecture allows the selective reduction of *fidelity* in order to improve feedthrough time. Clients can be grouped by their available bandwidth and processing time. Different qualities of stream (e.g., varying frame rate in video and application sharing or frequency range in voice transmission) can be provided to each group.

5.4. Replicated Input Broadcasting

Figure 7 shows the *replicated input broadcasting* distribution architecture. For some classes of application, this approach provides a low-bandwidth mechanism for synchronizing the views of multiple users. Also, this architecture provides an easy way of creating a collaborative version of certain existing single-user applications.

In this approach, each participant has a separate instance of the shared application on his/her PC. The instances are synchronized by broadcasting each participant's inputs to the other instances so that they can be applied locally. A CCCM unit is responsible for broadcasting the inputs and ensuring that they are applied in the same order to each instance of the application.

If the application is a reactor (that is, does not initiate its own activity, as with most word processors), the CCCM needs only to serialize inputs, applying them in the same order at each host. If the application is an actor (that is, it carries out its own activities in parallel with the player, as with a game), the CCCM needs to ensure that inputs are applied to all instances at the same point in their execution. This requires that the application provide a mechanism to permit barrier synchronization.

5.4.1 Examples

Microsoft's *Groove* [13] combines synchronous and asynchronous collaboration. Users may collaborate in real-time via a suite of tools such as a shared whiteboard. In addition, users can collaboratively edit Word documents.

Groove's collaborative Word editing is enabled via replicated input broadcasting. Both participants must run Word. Each user's inputs are broadcast to the other, and enacted on both the local and remote instances of word. An explicit floor control policy is used to ensure that only one participant at a time provides input to the application. The collaboration is purely WYSIWIS.

Age of Empires 2 [6] is a highly successful real-time strategy game, released in 2000, and still popular despite the release of its sequel in 2005. In the game, players develop a city through the ages while fighting neighbouring cities. Players may simultaneously perform inputs. The game application itself is an actor, as AI routines constantly advance gameplay even in the absence of user inputs.

The CCCM units on each client cooperate to gather user inputs and inject them into the game at the same point. The game is paused on each peer before the inputs are applied. The game then continues running until the next synchronization point. In an interesting tradeoff, inputs are delayed before their application. This allows higher *frame rate*, as concurrency control is resolved asynchronously with game operation, at the cost of worsened *feedback time*.

In addition to Age of Empires, this approach is used in numerous other real-time strategy games [12].

5.4.2 Qualities and Tradeoffs

The principal benefits of this approach are that it in some cases, it provides lower *bandwidth use* than other replicated approaches (section 5.5) and that, as a replicated architecture, it provides good *availability*.

In general, however, *feedback time* is delayed by the CCCM's distributed process for selecting the order in which updates are applied. In the special case of floor control (as in the Groove example above), feedback time is optimized at the cost of requiring participants to take turns.

The decision between using this architecture versus replicated state synchronization (section 5.5) depends largely on the *bandwidth consumption* of the two approaches. In Age of Empires, the messages required to encode user inputs are far smaller than the those required to compute state changes [6]. Conversely, far less data would be required to capture the changes in a word processor document than to encode the sequence of mouse and keyboard actions required to enact the changes. Additionally, this architecture is restricted to pure WYSIWIS views; if relaxed WYSIWIS is preferred, replicated state synchronization is a better choice.

This approach has good *availability* properties, in that the failure of one peer does not prohibit other peers from continuing. Some care is required with turn-taking protocols to ensure that the failed peer does not indefinitely hold the right to input.

A significant drawback of this approach is that it provides effectively no *security*, as all participants must be trusted with all application data.

The *scalability* of this approach is limited by available bandwidth, since each peer broadcasts its changes to all other peers.

This distribution architecture requires all participants to install the complete application, thus is less easily *deployable* than other approaches (section 5.2.) Deployment may be further complicated by the presence of firewalls, making peer-to-peer communication difficult. Centralized approaches (sections 5.1 and 5.3) are significantly easier to deploy in the presence of firewalls.

5.5. Replicated State Synchronization

Figure 8 shows the *replicated state synchronization* distribution architecture. For some kinds of application, this approach provides a low-bandwidth mechanism for synchronizing relaxed WYSIWIS applications. Here, participants interact directly with the application. The application's state is divided into a *shared context*, containing data

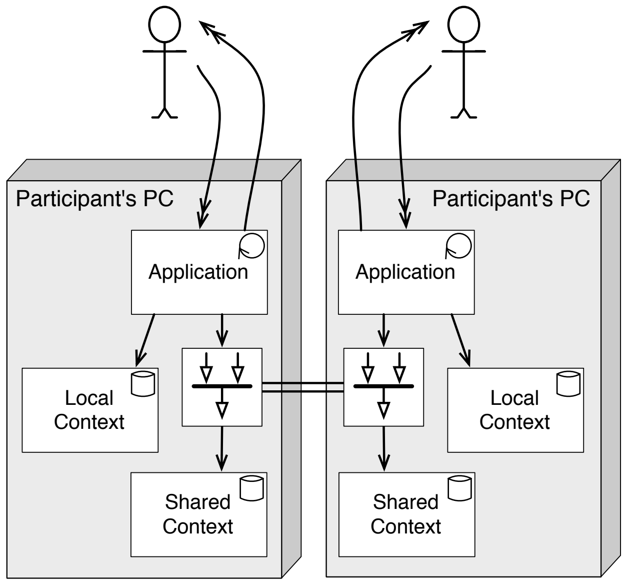


Figure 8. Replicated State Synchronization

common to all participants and a *local context*, which stores data specific to the given participant.

The shared contexts of the different participants are synchronized. All updates to the shared context pass through a CCCM unit whose task is to broadcast the updates to other participants and resolve conflicts in applying the updates. The CCCMs may use any of a range of pessimistic and optimistic concurrency control schemes [11].

5.5.1 Examples

The replicated state synchronization architecture has been applied in surprisingly diverse contexts. We present examples of its use in a shared text editor, a real-time strategy game, and a network of military command and control systems.

SubEthaEdit [24] is a relaxed WYSIWIS shared text editor. Any number of participants can edit the same document in real-time, seeing each others' changes as they are made. Participants can scroll the document separately. Awareness of other participants' actions is provided via telepointers, colour-coding, and scroll bar-based awareness widgets. The *shared context* stores the contents of the document and the awareness information. The *local context* stores the participant's own settings, such as font selection and cursor position.

The real-time strategy game *NetStorm: Islands at War* [12] replicates game state amongst all players' Shared Context components, and broadcasts commands that reflect the actions of game units. As with Age of Empires 2 (section 5.5.1), the application of user inputs to the game must be synchronized via a barrier-wait mechanism.

Command and control systems support cooperation in the real-time deployment of military assets. Systems developed under the *Multilateral Interoperability Programme* [19] follow the replicated state synchronization distribution architecture. The shared context contains information from each country's system, in that country's proprietary format. The applications are heterogeneous, separately developed by each country. The synchronization, in addition to its usual duties of concurrency control and consistency maintenance, is therefore also responsible for translating updates to a common form that can be applied to all replicas. Consistency is maintained via a dOPT-like algorithm [9].

5.5.2 Qualities and Tradeoffs

The replicated state synchronization distribution architecture is particularly useful in cases where *availability* is important. As with other replicated approaches (see section 5.4), it is possible for participants to continue in case of failure of a network link or another participant, as the application and its data are available on each local node.

As discussed in section 5.4.2, *bandwidth consumption* of this approach may be significant, depending on the amount and frequency of data change. The replicated input broadcasting architecture (section 5.4) may be more appropriate depending on the cost of transmitting inputs versus state changes.

The performance of this architecture largely depends on the chosen CCCM approach. In general, locking approaches penalize *feedback time*, while optimistic approaches reduce *fidelity* in cases when conflicts result in rollbacks or fixups [11]. *Scalability* is also influenced by the choice of CCCM; locking scales poorly, while optimistic approaches scale well when conflicts are infrequent.

This architecture shares the *scalability* and *security* concerns of other replicated approaches (see section 5.4).

6. Conclusion

To build effective groupware systems, developers need to be able to understand the tradeoffs involved in the choice of competing distribution architectures. To aid this, we have identified quality attributes of interest to groupware developers, and have shown the strengths and weaknesses of architectures found in groupware applications in use by thousands of people with respect to these attribues.

The importance of these quality attributes and their effect on system design is evident in the design of several of the applications surveyed in this paper. For example, WebArrow is primarily aimed at group meetings which may be organized at short notice for diverse participants. This scenario puts a premium on ease of deployment; however, scalability and availability are less important since group sizes are generally small. This suggests that the generic thin client architecture would be appropriate, and this is what WebArrow uses. One constraint of the generic thin client is that it can be bandwidth intensive due to the size of display updates; WebArrow resolves this though the use of the centralized mixer with broadcaster.

Previous work in this area has focused almost exclusively on performance. By taking considering a wider range of quality attributes of a number of commercially successful distribution architectures, we have provided further assistance to developers in making informed choices for future designs.

7. Acknowledgements

We gratefully acknowledge the support of the Natural Science and Engineering Research Council of Canada and of the Canadian Department of National Defence's Directorate of Land Command Systems Program Management. We would like to thank our colleagues in the EQUIS lab for their stimulating discussion around these topics.

References

[1] K. Baker, S. Greenberg, and C. Gutwin. Empirical development of a heuristic evaluation methodology for shared workspace groupware. In *Proc. ACM Conference on Computer-Supported Cooperative Work*, pages 96–105. ACM Press, 2002.

[2] S. Baset and H. Shulzrinne. An analysis of the Skype peer-to-peer internet telephony protocol. In *IEEE Infocom*, 2006. To appear.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, 1998. ISBN 0-201-19930-0.

[4] J. Beardsley. Seamless servers: The case for and against. In T. Alexander, editor, *Massively Multiplayer Game Development*, pages 213–227. Charles River Media, 2003.

[5] Y. W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization. In *Game Developers Conference*, 2001.

[6] P. Bettner and M. Terrano. 1500 archers on a 28.8: Network programming in Age of Empires and beyond. In *Game Developers Conference*, 2001.

[7] C. Butcher and B. House. Recreating the LAN party online: The networking and social infrastructure of Halo 2. In *Game Developers Conference*, 2005.

[8] P. Dewan. Architectures for collaborative applications. In M. Beaudouin-Lafon, editor, *Computer Supported Cooperative Work*. John Wiley & Sons Ltd., Jan. 1999. ISBN 0-471-96736-X.

[9] C. Ellis and S. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM Conference on the Management of Data* (SIGMOD '89, Seattle, WA, USA, May 2–4), pages 399–407. ACM Press, 1989.

[10] GoToMeeting. `www.gotomeeting.com`.

[11] S. Greenberg and D. Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '94, Chapel Hill, NC, USA, Oct. 22–26), pages 207–217. ACM Press, 1994.

[12] J. Greer and Z. B. Simpson. Minimizing latency in real-time strategy games. In D. Treglia, editor, *Game Programming Gems 3*, pages 488–495. Charles River Media, 2002.

[13] Groove. `www.groove.net`.

[14] S. Junuzovic, G. Chung, and P. Dewan. Formally analyzing two-user centralized and replicated architectures. In *Proc. ECSCW '05*, pages 83–102. Springer-Verlag, 2005.

[15] D. Kushner. Enineering EverQuest: Online gaming demands heavyweight data centers. *IEEE Spectrum*, 42(7):34–39, July 2005.

[16] F. Laramée. Dead reckoning in sports and strategy games. In S. Rabin, editor, *AI Game Programming Wisdom 2*, pages 499–504. Charles River Media, 2004.

[17] J. Lee. Considerations for movement and physics in MMP games. In T. Alexander, editor, *Massively Multiplayer Game Development*, pages 275–289. Charles River Media, 2003.

[18] M. Mintz. MSN Messenger Protocol. `http://www.hypothetic.org/docs/msn`.

[19] NATO Mutilateral Interoperability Programme. *Multilateral Interoperability Programme Technical Interface Design Plan*, version 2.5, December 2005. Available from `www.mip-site.org`.

[20] W.G. Phillips. Architectures for synchronous groupware. Technical Report 1999-425, Queen's University, Kingston, Ontario, Canada, May 1999. Available from `www.cs.queensu.ca`.

[21] W.G. Phillips, T.C.N. Graham, and C. Wolfe. A calculus for the refinement and evolution of multi-user mobile applications. In *Proceedings of the Twelfth International Workshop on Design, Specification and Verification of Interactive Systems* (DSV-IS '05), LNCS, pages 137–148. Springer-Verlag, 2005.

[22] J. Randall. Scaling multiplayer servers. In D. Treglia, editor, *Game Programming Gems 3*, pages 520–533. Charles River Media, 2002.

[23] ShowEQ open-source project. `http://www.showeq.net`.

[24] SubEthaEdit. Available from `www.codingmonkeys.de/subethaedit`.

[25] TeamSpeak. `http://www.goteamspeak.com`.

[26] WebArrow. `www.webarrow.com`.

[27] WebEx. `www.webex.com`.

[28] J. Yan and B. Randell. A systematic classification of cheating in online games. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–9, New York, NY, USA, 2005. ACM Press.