A Comparative Analysis of Groupware Application Protocols

Mark O. Pendergast
Washington State University[1]
mop@tricity.wsu.edu

Abstract

Two of the most difficult problems faced by developers of synchronous groupware applications are the handling of multiple session connections and the maintenance of replicated data. Protocols and algorithms to solve these problems have evolved over the years as developers gained experience and network standards were developed and enriched. This paper analyzes the efficiency of three common application level protocols used for the development of groupware systems. These include central sequencing, distributed operations, and independent-immutable objects. In order to perform the analysis, nine measures of network and processor efficiency were developed. The result of the analysis showed the independent object method to be superior in overall efficiency. Central sequencing is recommended for applications that are not conducive to the use of independent objects.

Keywords: GSS, Groupware, Multicast, LAN, Broadcast, CSCW, and Protocol Analysis.

1 Introduction

Groupware also referred to Computer-Supported Cooperative Work (CSCW) has been defined as *"Computer-based systems that support two or more users engaged in a common task (or goal) and that provide an interface to a shared environment"* [2]. Groupware applications are typically categorized as being either asynchronous or synchronous (or real-time). Examples of asynchronous applications include Email, file servers, electronic bulletin boards, news groups, shared databases, and World Wide Web information archives. These applications allow users to share and exchange information, but do not support real-time interaction. On the other hand, synchronous applications provide mechanisms for sharing data and seeing the changes made by other participants as they occur. This is referred to as What-You-See-Is-What-I-See (WYSIWIS) capability [17]. Different applications support different levels of WYSIWIS. Applications that support a strict level of WYSIWIS allow all users to see the same screen image at all times. This includes cursor movements by other users, new characters or graphics symbols added, and synchronization of scrolling. Other applications provide a relaxed level of WYSIWIS support. These applications allow users to work on different segments of a shared document/design, all copies of the document are kept up-to-date in real-time and should users work on the same portion of the document, their changes would appear on each other's screens in real-time. Synchronous applications include multi-user text editing tools, voting, conferencing, graphical design, sketching, and brainstorming. For more in depth discussion of groupware applications I refer the reader to [4][6][8][15][19]. The techniques pioneered groupware researchers have been transferred to commercial products such as Proshare (TM Intel), Lotus Notes (TM Lotus Corporation), GroupSystems (TM Ventana Corporation), Windows for Workgroups (TM Microsoft) and LiveBoard (Xerox).

The technology that supports synchronous groupware often takes the form of intricate, multi-user software applications running in a distributed environment. This distributed environment typically consists of UNIX workstations or personal computers communicating over a local or wide area network. Data is stored on file or database server and temporarily replicated on stations while the groupware applications are active. TCP/IP (Berkeley and WINSOCK), NETBIOS, and IPX/SPX protocols are commonly employed. These protocols provide reliable point-to-point session connections and broadcast datagram connections. These commercial products do not support reliable, sequenced broadcast messages ([1] and [13] describe reliable broadcast protocols that could be put to good use in groupware applications).

[1] Washington State University, Richland, WA 99352 USA **Email**: mop@tricity.wsu.edu

Two of the most difficult problems faced by developers of synchronous groupware are the handling of multiple session connections and the maintenance of replicated data [16]. The mechanism that the groupware developer selects for solving one problem often dictates the solution that must be employed for the other. If data integrity is maintained by sequencing all changes through a central station, what happens when that station exits the collaborative session? Does another station take over or is the collaboration of all stations terminated? When a user starts a collaborative application, how does the application know what other users are currently active with the same data? What happens on these machines while the new user establishes session connections to the other users and downloads data from the central server? Finally, how is information and view consistency guaranteed?

Mechanisms to solve these problems have evolved over the years as developers gained more experience and network standards were developed and enriched. Three classes of models for handling multiple sessions and replicated data have emerged. These include *central sequencing*, *distributed operations*, and *independent-immutable objects*. *Central sequencing* requires all changes to shared data to be in the form of transactions ordered by a single station. *Distributed operations* require that changes be in the form of transactions that are broadcast along with "state" information to all stations. Special application dependent algorithms then apply the transactions. *Independent-immutable objects* also require changes to be in the form of transactions, sequencing of the transactions and special algorithms to manipulate transaction and state information are not necessary. Instead, these applications rely on the independence inherent in the data to resolve conflicts. These models were chosen for analysis due to their diverse approaches for solving collaboration problems. Central sequencing in the form of client/server applications and sequenced transaction applications is the most prevalent form of collaborative application. This groupware mainstay has been challenged by groupware researchers developing more efficient alternatives, namely operation transformation and immutable object paradigms.

These approaches impose different loads on the network and processor, and have different levels of intricacy for developers to contend with. This paper performs an in depth examination of the three models, comparing them with regard to intricacy, network efficiency, and processor load. The next section of this paper establishes the criteria for evaluation, following that is a definition and evaluation of each model. The paper concludes with a summary of the evaluation and how the results can be applied in groupware development and research.

2. Defining Comparison Measures

In order to define measures to compare the three approaches it is first necessary to resolve what it is they that really are in computer system terminology. From a networking point of view, the three approaches contain elements of the upper three layers of the ISO/OSI reference model, namely, Session, Presentation, and Application. From that viewpoint they should be considered *network protocols* and evaluated in terms of network throughput, response time, and utilization. However, intrinsic to these approaches are mechanisms for handing data, processing user requests, and applying transactions. Therefore, it is necessary to evaluate these approaches from an algorithmic point of view. This requires the approaches to be evaluated in terms of finiteness, definiteness, and effectiveness [10]. The following paragraphs detail the measures used for comparison from a network and algorithmic viewpoint.

2.1 Network Protocol Measures

Introductory data communications courses teach us that network performance should be measured in terms of availability, response time, throughput, and utilization. These measures are well defined for evaluating lower level protocols (Transport, Network, and Data link) where the primary concern is the efficient use of the transmission medium. To apply these same measures to groupware Application level protocols requires some adaptations.

Availability in the context of computer networks is defined as the percent of time a resource is available to users, or the mean-time-between-failures(MTBF) divided by the sum of the mean-time-between-failures and the mean-time-to-repair(MTTR). Failure at the application level is caused by either failure at one of the lower protocol layers or by poor implementation of the application (coding errors). If all three techniques being compared in this paper rely on the same communications platform and they are all equally robust, then differentiating factor is the amount of time it takes the technique to recover from a network failure (MTTR). The time to recover is a function of the number of connections that have to be restored, the amount of data to be downloaded, and the number of control messages that have to be sent to re-synchronize all stations. Another factor affecting availability is the amount of time it takes a user to join a collaborative session. This time is both the initialization time for the new user and any interruption imposed on existing users. As with recovery, the number of connections that must be established, the amount of data to be downloaded, and the number of control messages required are all-important factors. An initialization or recovery time that is too long could preclude the use of the groupware application for casual, spontaneous interactions.

Throughput and response times are a combined measure of the network's ability to transmit data, the amount of data to be transmitted, and the amount of processor time required for handling the data. From the network perspective, response times for these collaborative models are determined by the number of messages required to get a transaction to all the participating stations. Utilization has been defined as being both a measure of how much of a resource is being used and how efficiently the resource is being used. For the purposes of this paper, utilization translates into three factors, how large are the transaction messages, how many messages must be sent, and how much of each message is overhead required by the algorithm as compared to data entered by the user. Total processor overhead is a function of the number of messages sent, length of the messages, and the amount of data manipulation that must occur for each message. Section 2.2 gives more details on the data manipulation aspects of the collaborative models.

Condensing availability, throughput, response time, and utilization factors into a single list give us:

- *Messages required for recovery* - the number of messages required for the application to self-recover when a station exits the session or a connection is lost. ($M_{recovery}$)
- *Messages required for initialization* - the number of messages required when a station joins a session. ($M_{initialization}$)
- *Messages per transaction* - the number of messages sent network wide to get a transaction to all stations participating in the activity. ($M_{transaction}$)
- *Message size* - number of bytes required for each transaction message. This includes both user entered data and algorithm overhead. (M_{size})
- *Message Efficiency* - a measure of how efficient messages used for sending transactions are. This is calculated as the ratio between application data to overall message size. In other words, what portion of the message is not overhead required by the algorithm. ($M_{efficiency}$)

2.2 Computer Algorithm Measures

Classical algorithm analysis as described by Knuth [10] requires that an algorithm be judged on three fundamental characteristics, *finiteness, definiteness*, and *effectiveness*. *Finiteness* is a measure of not only whether an algorithm terminates after a given number of steps, but how many steps or cycles are necessary for it to complete. This behavior of an algorithm is typically measured in terms of an approximation of its upper bound, referred to as "big-oh" or O(f(n)). An algorithm that completes in O(n) performs a number of steps proportional to the "n" items being processed, $O(n^2)$ performs a number of steps proportional to "n" squared. *Definiteness* is a measure of how precise and unambiguous the algorithm is. Will it work in a predictable, consistent manner each time? Can the algorithm be implemented using existing programming languages? Finally, *effectiveness* refers to how well the algorithm actually works for the problem at hand. What compromises to function or performance have to be made to implement it, and can it be applied to a larger class of problems?

In order to judge the groupware approaches on the basis of *finiteness, definiteness,* and *effectiveness,* I will use the following, somewhat more quantifiable, measures:

- *Transaction response time, local* - the amount time required for the results of a transaction to appear on the station which generated it. This consists of processor time required to create a transaction on the originating station and any delay imposed due to message transmission. Measured in terms of big-oh. (T_{local})
- *Transaction processing time* - the amount of processor time required to implement a transaction on a station other than the originating station. Measured in terms of big-oh. ($T_{process}$)
- *Intricacy of transaction creation and processing* - this is a subjective measure of the amount effort it would take to implement the algorithm. This can be characterized by the *number of states necessary to implement the algorithm* as well as the difficulty in applying the algorithm to a given application.
- *Generalizeability of algorithm across applications* - this is a subjective measure of the algorithm's ability to be applied to a variety of group applications and its suitability for being implemented in a re-useable, object-oriented manner. A rating of *High* indicates that it could be implemented for any application and requires little or no intricate algorithm development. *Fair* indicates that it could be implemented for most applications with some

amount of algorithm development. *Poor* indicates that it would not work for all applications or would impose constraints on the functionality of the application.

The following section presents an examination of each of the three approaches using the measures defined for network and algorithm efficiency.

3. Model Definition and Analysis

The three models for handling multiple sessions and replicated data that are analyzed in this section are: central sequencing, distributed operations, and independent-immutable objects. The following sections define and analyze each algorithm with respect to the criteria established in section 2.

3.1 Central Sequencing Model

The Central Sequencing Model relies on a data and/or transaction server in order to maintain data integrity. This model has been successfully implemented for a variety of applications. There are several ways in which the central model has been implemented: *terminal linking*, *centralized data*, and *distributed data*. Under *terminal linking*, the application runs on the central node, mouse and keyboard inputs from all users are routed to this node, the application's outputs are broadcast to back to all users. This linking can be performed via software or hardware (daisy chaining of keyboards). The major problem with terminal linking is that the application is forced to use a strict WYSIWIS, i.e. individual views are not possible and users cannot work concurrently. This approach does allow the sharing of applications that were not specifically designed for group work [6]. *Centralized data* implementations are similar to database and client-server applications. One copy of the data exists on a server, users send transactions and read data from the server. This implementation is less popular as it increases read access times and makes real-time view updates difficult to perform. The third, and most popular form for synchronous groupware applications is the *distributed data model*. Under this model the data is resident on all stations, and the central node sequences operations. When a user makes a change, the operation is sent to the central node, that broadcasts it to all users using a multicast tree approach [11]. The operation is not applied locally until it is received back from the central node. This ensures that all stations have consistent data, however, it does not guarantee that the users' intentions will always be carried out. For example, two users attempt to correct a spelling error in a shared document by inserting a missing character, the result of this operation would result in an unintended extra character. Both users then delete this extra character, resulting in an unintended deletion. Local response times can be impacted due to the round trip delay imposed by the sequencing of operations . Variations of this algorithm apply locally generated operations immediately, then redo them in correct sequence when they arrive back from the central node [5].

Implementation

Figure 1 represents the state diagram for implementing central sequencing with distributed data. When an application is executed it is initially in State 1 (Initializing). The application session is not known by the network, the data object has not been loaded. As before, the application must open a multicast connection. If this is the first instance of the application, i.e. the first user to enter a session, the station assumes the role of central server and session data is loaded from an existing object on the server disk or a new object is created. After which the application goes to State 3 (Ready). If this is not the first instance, then the application must initiate a download sequence.

Session data is downloaded from the central station by first sending an **download_req**, then going to State 2, Download In Progress. The server then responds with **download_data** messages and a **download_complete** message. Once the **download_complete** message is received, the station goes to State 3 (Ready). While in State 2, all messages are discarded except the **download_data** and **download_complete**. Since the download request message is sequenced by the server along with other requests, it is not necessary to place the entire system in quiescence before downloading a station. If the amount of data to be downloaded is large, then it may make sense for the central station to freeze other sites in order to prohibit an excessive number of requests from stacking up in its input queue while the download is in progress.

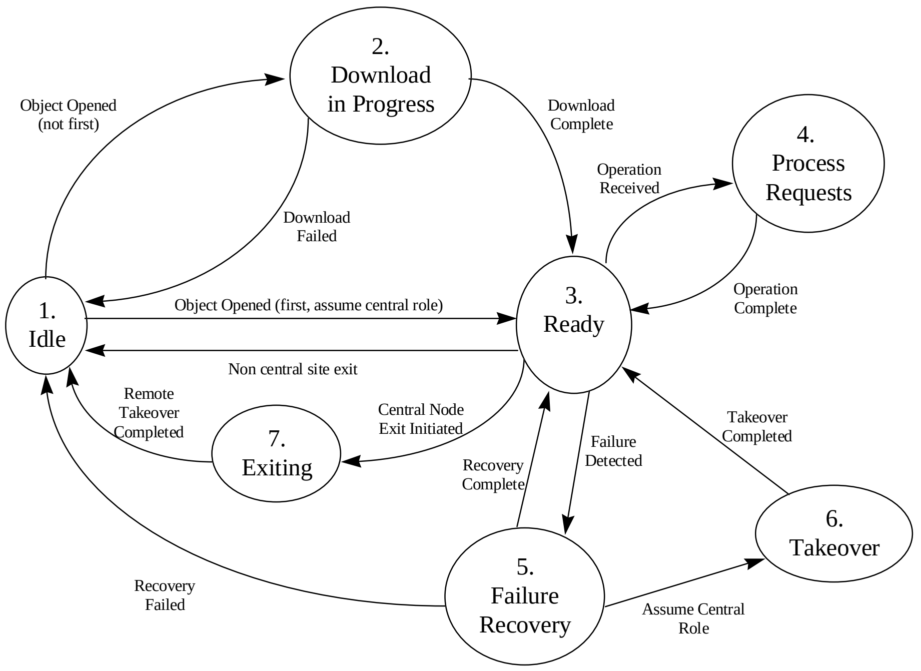

Figure 1 Implementing Central Sequencing

While in State 3 (Ready), sites receive and process operation request messages (state 4), close requests, and failure messages. Since all operations are sequenced by the central host they may be executed as soon as they arrive. The most recent operation is saved for failure recovery operations. Failure of a site that is not the central node requires no action on the part of the application running on other sites. The application will be notified of the failure via a **failure_detect** message and can take the opportunity to update active user lists and to clear gesture markers or user icons from the screen. If the central node fails, then the session must go through a failure recovery process.

Failures are reported to application level procedures when a session abnormally closes or when a transmission cannot be completed. If the central node failed while it was in the process of broadcasting an operation to other sites, then it is likely that some of the sites received the last operation and some did not. Therefore, the recovery process must verify that all sites have received the same number of operations. Since all sites have a copy of the data (plus or minus the last operation), any of them are eligible to take over as the central node. Each station maintains a list of active stations by *station id*, the simplest method of determining the new central node is to choose the station with the largest (or smallest) id. In the case of a wide area network, then the station that can communicate over the shortest path to all other stations is chosen. When an application receives a **failure_detect** notification of the central node, it first determines the station id of the new central node. It then sends a **takeover_request** message to the new central node and moves to State 5 (Failure Recovery). The new central node will learn of the failure of the central node either by receiving a **failure_detect** notification or a **takeover_request** message. In either case it assumes the role of the central node and initiates the recovery process by broadcasting a **takeover_inprogress** message to all sites and assumes State 6 (Takeover). It then waits for all sites to respond with a **recovery_inprogress** message (containing the last operation performed). As **recovery_inprogress** messages are received by the new central node, the operations contained in each are checked against the last operation completed. If they are different, then the operation is executed. Once all **recovery_inprogress** messages have been received then the new central node broadcasts a **recovery_complete** message to all nodes and returns to State 3. This message contains the last operation. When **recovery_complete** message is received the operation is checked against the last operation completed at that node. If they are different the operation is executed. The site then returns to State 3. A recovery timer is used to prevent sites from staying in States 5,6 indefinitely. If the timer expires, then multiple failures are assumed and a close/exit sequence is performed. This recovery process verifies that all sites are able to communicate with the new central node and all sites have completed the same number of operations.

When a non-central node site closes, then no special processing need be done. When the user of the central node exits, then it is necessary for the central node to determine which site will take over and to initiate the takeover operation. When the exit request is received, the "old" central node determines the new central node via site id, broadcasts an

takeover_req message to all nodes, then goes to State 7 (Exiting) . This message designates the id of the new central node. When this message is received, non-central node sites will discontinue sending operations to the old central node and begin sending to the new one. The new central node will then assume its new role and begin sequencing and broadcasting incoming requests. Both central and non central nodes will respond to the **takeover_req** message by sending a **takeover_ack** to the old central node. While the old central node is in State 7 (Exiting) it may receive operations from stations that haven't processed the **takeover_req** message yet. Instead of broadcasting these operations to all sites, it merely forwards them to the new central node. They are then sequenced with other messages. Once **takeover_ack** messages have been received from all sites, then the old central node can safely exit.

Quiescencing (idling) the system is not necessary for checking pointing of data, recovery from failures, or exiting of the central node. Since each station maintains connections with all active participants, central node takeovers are easy to perform. Some messages that are sent between stations do not require sequencing (gesture messages, view coordination messages, etc.). These messages can be broadcast directly to those stations, thereby reducing the overhead on the central node.

Analysis

Table 1 presents a summary of performance measures of the central sequencing algorithm. The strength of approach is its ability to process local and remote requests in constant time and the fact that it doesn't require quiescence for check pointing, and failure recovery. The total number of transmissions per operation is roughly equivalent to that required by the other approaches, N versus N-1 (N= number of concurrent users). Network transmissions are both a strength and a weakness of the approach. On the plus side, each site is responsible for only for transmitting an operation to the central node, not to all sites. On the minus side, each operation must pass through the central node, creating a bottleneck and increasing the response time for operations generated locally (2T, T = transmission time).

Transaction efficiency is calculated as the size (in bytes) of the operation divided by the total size of the request. A request was defined as < id, O_i,>, where the *station id* is 2 bytes long and the size of the operation (O_i) is application dependent. Using a simple text editor as an example, the operation would require 1 byte for operation type (insert, delete), 1 for the character, and 4 for the position (6 bytes total). The total message size is 8 bytes, 2 of which is overhead of the site id. Efficiency is 75%. O_i will be larger for more intricate applications and operations. Therefore, 75% should be viewed as a lower bound.

A final advantage of implementing the sequencing model, as opposed to the distributed operation model is intricacy. Applications are not required to maintain and search request logs and queues, operations do not have to be transformed, and failure recovery operations are less intricate. This is evident by the fact that only 7 states are required to implement sequencing as compared with 9 for distributed operations. The delay for executing local operations can be eliminated by employing techniques such as reversible execution [2].

3.2 Distributed Operation Model

The best groupware example of a Distributed Operation Model is the Distributed Operation Transformation Algorithm (dOPT) described by Ellis [3]. The dOPT algorithm allows an application to maintain data on multiple sites without having to use locks or a central sequencing station. Thus operations can be performed immediately on their originating sites, lock/unlock overhead is eliminated, and the bottleneck created by a sequencing station is removed. Under the dOPT algorithm it is not necessary to sequence operations if you :

1) Do not execute an operation O_j (j = originating station) on another station until you execute all operations executed by station j prior to O_j..
2) You transform operation O_j to take into account any operation executed on the other station that had not been executed on station j.

In order to satisfy these conditions, each station is required to maintain a state vector S_i, request log, request queue, and transformation matrix. The state vector indicates the operations performed on that station which originated from other stations. The request log keeps track of operations performed at a given site and is referenced when transforming operations. The transformation matrix determines how an operation must be modified to take into account operations that were performed on the receiving station, but not yet performed on the sending station. The transformation algorithms also have mechanisms to help preserve the users' intentions when conflicts occur (redundant insertions or deletions). When a user performs an operation, e.g. inserts or deletes a character, the operation is built into a request that contains the current state vector of the site, the site id, the operation, and the operation's priority $< id, S_i, O_i, P_i >$. The request is then queued for execution on the local site and broadcast to all other sites. When a request is received, the current state vector at that site is compared to the state vector of the request, the operation is either held on the request queue, executed immediately, or transformed before execution. Quiescence, i.e. an idle state, is enforced periodically to allow the request queue to be reset and the data to be saved to disk. For a complete discussion of the dOPT algorithm reference [3]. Sun[18] describes an alternative transformation algorithm.

Implementation

Figure 2 presents a generic state diagram for implementing a distributed application with dOPT. When an application is executed it is initially in State 1 (Initializing). The application session is not known by the network, the data object has not been loaded. While in this state the application must create a multicast session. If this is the first instance of the application, i.e. the first user to enter a session, then session data is loaded from an existing object on the server disk or a new object is created. After which the application goes to State 4 (Ready). If this is not the first instance, then the application must initiate a download sequence.

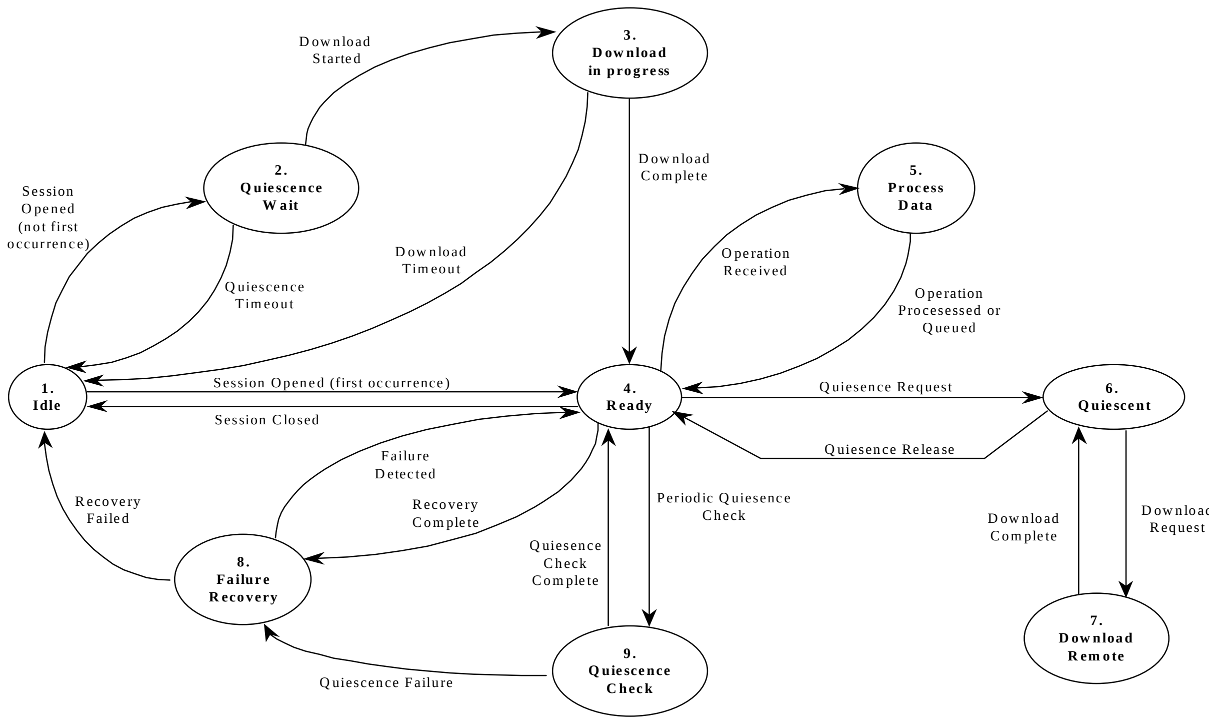

Figure 2 Distributed Operation Algorithm Implementation

In order to download data from another site it is first necessary to enforce quiescence on all the sites [3]. This insures that all queued operations have been completed, all local operation logs and state vectors have been reset, and the data set is identical on all sites. While in quiescence, inputs from users are not accepted (although they may queue up in type-ahead buffers). In order to put the session in a quiescent state the application broadcasts a **freeze_req** message, then assumes State 2 (Quiescence Wait). Upon reception of the **freeze_req** each site will suspend processing of user inputs for the corresponding session, increment their freeze request count, and reply with an **freeze_ack** message as soon as their pending request queue is empty. These stations then enter State 6 (Quiescence Hold) until they receive a download or **thaw_request** message. Once all sites have replied with an **freeze_ack**, the application then sends an **download_req** to one of the active sites and assumes State 3 (Download in Progress). The selected site enters State 7 (Downloading) and begins sending data using the **download_data** message class. The last block sent piggybacks a **download_complete** message, following which the site returns to State 6. If the data to be downloaded is in one contiguous block, then the download can be performed with one transmit operation. When the download complete message is received, the initializing application broadcasts a **thaw_request** message (freeing all sites in State 6), starts the periodic quiescence timer, and goes to State 4 (Ready). At this point all sites should be in State 4, with logs and state vectors that have been reset, and identical data sets. All wait and hold states have a corresponding time-out event that is initiated when entering the state. If the time-out occurs then the sites return to their previous states (READY), in the case of an application that is initializing, the time-out indicates an initialization failure and the applications fails.

The initialization of a second site while the download sequence is in progress poses no problems. Sites that are already in State 6 respond immediately with an **freeze_ack** message, and update their freeze request count, and reset their time-out events. Sites that are in State 7, reply with an **downloading** message and update their freeze request count and time-out events. The **downloading** message tells the initializing site that this site is frozen and if possible, it should request a download from a different site. When **thaw_request** messages are received, the freeze request count is decremented, and if zero, the site returns to State 4.

While in State 4 (Ready), applications can receive one of five events: operation requests, freeze requests, periodic quiescence checks, failure detection, or close requests. Operation requests can originate locally, or from remote sites. When a new request is received, the application enters State 5 and processes the new request and request queue as described by [3]. The processing of freeze requests has already been discussed. The other three events require some discussion.

Periodic quiescence checks are necessary in order to prevent the operation log from growing too large, for check pointing (saving to disk), and as a means to insure that all sites are synchronized. A system timer is used to initiate the quiescence process. When the timer event is generated on a station, that station broadcasts a **freeze_req** message to all stations and enters State 9 (Quiescence Check). Stations receiving the **freeze_req** disable user input to the session, enter State 6 (Quiescence Hold), and reply with an **freeze_ack** message once their pending request queue is empty. This may require waiting for messages from other stations. Once all stations have replied with an **freeze_ack** message, the initiating station may save data to disk, reset the state vectors and operation log, restart the quiescence timer, broadcast an **thaw_request** message, and return to State 4. Stations receiving the **thaw_request** message will reset their operation logs, state vectors, and quiescence timers, then return to State 4. Again, timers are used to prevent the stations from staying in State 7 indefinitely and the initiating station from staying in State 9 indefinitely. If a time-out occurs while in State 7, the site returns to State 4 without resetting state vectors or operation logs. If a time-out occurs while in State 9 failure recovery operations are initiated (State 8).

Failure recovery for an application using the dOPT algorithm is a non-trivial task. Failure can occur at any time and more likely than not, sites will not be quiescent at the time. The host communication systems detect site failures in two ways, one when a transmit to that site has failed (timed out), and the other when the transport level (TCP/IP, IPX/SPX, NETBIOS, etc.) reports a broken connection. Broken connections are detected by the transport level as a result of receiving bad or incomplete messages, or when it is unable to transmit a periodic check message to that site. In the case where the failure due to a transmit time-out, this indicates that the message which was to be sent to all stations was not received by at least one of them. If the transport level system detected the failure, this indicates other sites may have received a message that the detecting station did not. In both cases a site has failed and it is necessary to insure that all sites have processed all operations.

The simplest way to insure that all sites are synchronized is for the detecting site to broadcast a message freezing user inputs at all healthy stations, then download each station with a copy of the data. This is feasible for small groups or for a small amount of data, but not for larger groups working with a large data set (e.g. a 500-page document). For these applications it is necessary to perform what I will call an *on-line recovery*. An on-line recovery is accomplished as follows. When an application is notified of a failure (**failure_detect** event) it broadcasts a **recovery_complete** message to all other stations and enters State 8 (Failure Recovery). The recovery message contains a vector of operations received by other stations and the site id of the failed station. This vector is similar to the state vector used by dOPT, but it also includes operations that have been received but not processed. When a **recovery_complete** message is received while in State 4(Ready) the station first compares the operations received vector from the message with its own. If it has received operations that the sender has not, then it transmits those operations (either from its request queue or operation log) to the sender. The site then broadcasts its own **recovery_complete** message to all sites and enters State 8 (Failure Recovery). If the station is already in State 8 when a **recovery_complete** message is received, it compares the operations received vector from the message with its own. If it has received operations that the sender has not, it transmits those operations to the sender. While in State 8 applications will queue incoming operations, but not process them. Duplicate operations are discarded. Once a station has received a **recovery_complete** message from all other healthy stations, it may return to State 4 and process its request queue.

If the on-line recovery process completes, then two things are known. First, every station has been able to exchange messages with every other station, and second, all stations have received all operations. This algorithm will work when multiple sites detect a failure simultaneously since the detecting station initiates, rather than arbitrates, the recovery process. If a second failure occurs while the recovery is in progress, then the best course of action is for each site to check point its data and exit. Multiple failures indicate network problems or conditions that cannot be recovered from on-line.

The final event that can be received in State 4 is the exit of the application by the user. If the user is not the last one to exit the session and a data save is desired, then the system will have to go through quiescence before saving and exiting. This process is identical to the one described previously. If the user is the last one to exit, then the data may be (must be) saved and quiescence is implicit.

Analysis

Table 1 presents a summary of performance measures of this algorithm. The strength of approach is its ability to process locally generated requests without incurring transmission delays to a central node. Data locking or sequencing of operations is not required. The total number of transmissions per operation is roughly equivalent to that required by a central sequencer, N-1 versus N (N= number of concurrent users). Network transmissions are both a plus and a minus for the dOPT algorithm. On the plus side, each site is responsible for transmitting an operation to all the other sites, thus eliminating the bottleneck of a central sequencer and spreading the processor overhead required for transmissions evenly over all sites. On the minus side, each active site must incur the processor time to transmit to all other sites. Under the sequencing model, each site need only transmit its operation to the central node. If processing capacity is a problem and a dedicated sequencer is available, then the sequencing model could improve overall performance.

Transaction efficiency is calculated as the size (in bytes) of the operation divided by the total size of the request. A request was defined as $< id, S_i, O_i, P_i>$, the state vector and priority are O(N) in size, and the size of the operation is application dependent. Using a simple text editor as an example, the operation would require 1 byte for operation type, 1 for the character, and 4 for the position (6 bytes total). If the site id, priority, and states are all 2 bytes long, then efficiency is 6/(4N+8). If 4 users are active, then efficiency is 25%; for 10 the efficiency drops to 12.5%. Therefore efficiency is inversely proportional to the number of active sites.

Request creation and processing times can become a concern if there are a large number of users or if the quiescence interval is long. To process a request it is necessary to first compare state vectors O(N), and possibly transform the operation. Transforming an operation requires comparing the priority of the operation to that of operations in the request log and performing a simple manipulation of the operation. The length of the request log is a function of the quiescence interval (Q_t) and the number of users (N) . The time required to compare priorities is O(N). Thus operations that do not need to be transformed can be performed in O(N) time, operations that require transformation in $O(N^2Q_t)$ time. Similarly, request creation requires scanning the request log and comparison of priorities in order to calculate the priority of the operation. This does not compare favorably with the central sequencing model, which can create requests and process incoming requests in constant time.

In short, the dOPT algorithm when implemented as described will provide immediate response to local operations at the expense of processor overhead required in message transmission and operation transformation. Each operation for a given application must have its own transformation algorithm and it is likely that algorithms may not be possible for all operations that would be expected to be supported by the application. This makes this algorithm less generalizeable than the others.

3.3 Independent-Immutable Objects Model

In the context of collaborative applications, independent objects can be defined as objects whose physical and temporal ordering have no effect on the document or group view. That is, all operations on the document are additive and commutative, and all operations on the objects themselves are autonomous. These properties allow a group document to be maintained without the use of a central sequencer or transformation of operations. Examples of independent objects are: telepointers[7], electronic votes, surveys, and unordered lists (brainstorming ideas, nominations, etc.)[14]. If differences in the presentation of overlapping graphics can be tolerated, then pen strokes and graphical shapes can be also be considered independent objects. A good example of an application written using immutable objects is the Tivoli shared whiteboard [12]. The algorithm described in this section is modeled around the one employed in Tivoli.

Consistency can be maintained for concurrent editing operations on independent objects if the objects are immutable [12]. Changes to immutable objects are accomplished by deleting the old instance, then adding a new one. If two users change the same object at the same time, then the old instance is deleted and two new ones appear. Users are then required to resolve the changes [12]. An alternative to the immutable object approach is to allow only atomic level changes to the objects [9], and resolve conflicts based on a user priority system. For example, if one user changes the color of an object while another user moves the object then both changes can be applied without conflict. If both users were to move the same object at the same time, then the operation initiated by the user with lower priority (determined by user id) would be rejected. The user whose operation was accepted would see the object moved to the desired location, the other user would see the object moved first to the location designated by him/her, then to that of the higher priority user. Use of this method requires the previous state of the changed attribute to be transmitted with the operation and the user id of the person who initiated the last change maintained as part of the object. Comparing the previous state in the transaction to the current state of an object allows collisions to be detected. Comparing the user id of the last change with that of the transaction allows collisions to be resolved. Either method allows editing of independent objects without sequencing or locking.

Implementation

Figure 3 represents the state diagram for implementing an application using the Independent Object Model in a multicast environment. When an application is executed it is initially in State 1 (Idle). The application session is not known by the network, the data object has not been loaded. As before, the application must open a multicast connection. If this is the first instance of the application, i.e. the first user to enter a session, then session data is loaded from an existing object on the server disk or a new object is created. After which the application goes to State 3 (Ready). If this is not the first instance, then the application must initiate a download sequence.

Session data can be downloaded from any active station by first sending an **download_req**, then going to State 2, Download In Progress. The selected station then responds with **download_data** messages and a **download_complete** message. Once the **download_complete** message is received, the station goes to State 3 (Ready). While in State 2, all messages are queued except the **download_data** and **download_complete**. Once the down load has completed, then the queued messages may be processed.

While in State 3 (Ready), sites receive and process operation request messages (State 4), close requests, and failure messages. All operations are executed as soon as they arrive. The most recent operation from each site must be saved for failure recovery operations. If a node fails while it was in the process of broadcasting an operation to other sites, then it is likely that some of the sites received the last operation and some did not. Therefore, the recovery process must verify that all sites have received the same number of operations.

Failure recovery under the independent object model follows the same general process as that of the dOPT model, but is less intricate. When an application is notified of a failure (**failure_detect** event) it broadcasts a **recovery_complete** message to all other stations and enters State 5 (Failure Recovery). The recovery message contains the id of the failed site and the last operation received from the site. When a **recovery_complete** message is received while in State 3(Ready) the station broadcasts a **recovery_complete** message that contains the last operation of the failed site, then enters State 5 (Failure Recovery). If the station was already in State 5 when a **recovery_complete** message is received, it compares the last operation of the failed site from the message with its own. If it the operation in the message is newer, then the operation is processed, else it is discarded. While in State 4 applications will queue incoming operations, but not process them. Once a station has received a **recovery_complete** message from all other healthy stations, it may return to State 3 and process its request queue. A software timer is used to prevent stations from remaining in State 5 indefinitely.

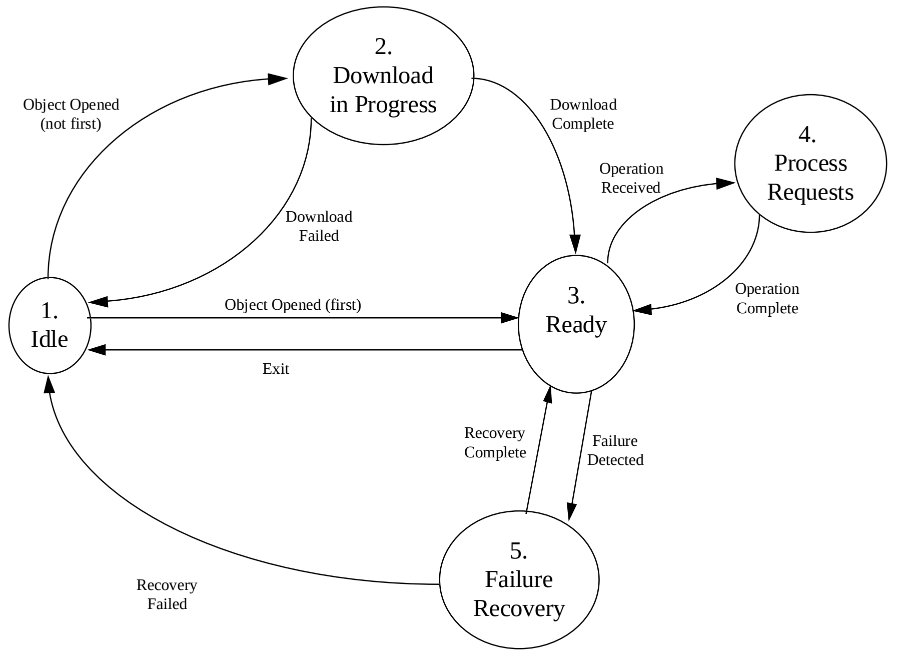

Figure 3 - Implementing Independent Objects

Once the recovery process completes, then two things are known. First, every station has been able to exchange messages with other station, and second, all stations have received all operations. This algorithm will work when multiple sites detect a failure simultaneously since the detecting station initiates, rather than arbitrates, the recovery process. If a second failure occurs while the recovery is in progress, then the best course of action is for each site to check point its data and exit. Multiple failures may indicate network problems or conditions that cannot be recovered from on-line. Quiescencing the system is not necessary for checking pointing of data, allowing stations to enter or leave the session, or recovery from failures.

Analysis

Table 1 presents a summary of performance measures of the independent object model. The strength of approach is its ability to process local and remote requests in constant time and the fact that it doesn't require quiescence for check pointing, and failure recovery. The total number of transmissions per operation is equivalent to that of the dOPT algorithm, N-1 (N= number of concurrent users).

Network efficiency for this technique is dependent on the type of operation. Editing operations will have an efficiency of about 50%, while add/delete operations are nearer to 80%. This is due to the requirement of transmitting previous state information when performing edit operations on existing objects. An edit request is defined as $< id, O_i, P>$, where the site id is 2 bytes long , and the sizes of the operation O_i and previous state P are application dependent. Using the movement of a shared graphics object as an example, the operation would require 1 byte for operation code, 2 bytes for an object id, and 4 bytes for the x,y location. The previous state (x,y location) would require 4 bytes. Therefore, efficiency is 7/13 or 53%.

Of the three approaches, independent object is the simplest to implement. Applications are not required to maintain and search request logs and queues, operations do not have to be transformed, and failure recovery operations are far less intricate. This is evident by the fact that only 5 states are required to implement sequencing as compared with 7 for sequencing and 9 for dOPT. The difficulty in implementing this model is in deciding where to draw the boundary between objects. This is readily apparent in the example of a shared document. Are the objects individual characters, paragraphs, sections, or chapters? Reducing the granularity of the objects enables more concurrent work to be done, but increases the likelihood of conflicts between works. Depending on implementation, the resolution of conflicts involves either discarding work done by one user, or requiring users to manually resolve collisions. This makes the overall approach somewhat less generalizeable than the central sequencing approach.

4 Summary and Implications for Groupware Developers

A summary of the evaluation criteria is presented in below in table 1. The Independent Objects algorithm is superior in terms of response time, intricacy, and is second to Central Sequencing in the other categories. Central Sequencing is the most efficient in terms of network overhead and is the most generalizeable. Distributed Operations fair poorly in network efficiency, intricacy, and generalizeablity. This is largely due to the need for transmitting, processing, and maintaining transaction state information and transaction logs. In Distributed Operation algorithm message size is a function of the number of users, this limits its scalability.

Criteria	**Central Sequencing**	**Distributed Operation**	**Independent Objects**
$M_{recovery}$	3(N-1)	$(N-1)^2$	$(N-1)^2$
$M_{initialize}$	3+	3(N-1) +3	3+
$M_{transaction}$	N	N-1	N-1
$M_{efficiency}$	$O_i/(O_i+2)$	$O_i/(O_i+4N+2)$	$O_i/(O_i+2+P)$
M_{size}	O_i+2	O_i+4N+2	O_i+2+P
T_{local}	O(T)	$O(N^2Q_t)$	O(c)
$T_{process}$	O(c)	$O(N^2Q_t)$	O(c)
States	7	9	5
Generalizeability	High	Fair-poor	Fair

Table 1 - Algorithm Comparison[2]

Based on this analysis, groupware developers should look first at structuring applications to work under the Independent Objects method. For some applications this will mean relaxing WYSIWIS constraints or devising a mechanism for ordering objects which does not rely on the sequence they were generated in or upon any physical ordering. This would include functions such as telepointers, shared bitmap editors, brainstorming, electronic voting, and broadcast audio/video. Central Sequencing is suggested for those applications that cannot be implemented as Independent Objects due to the necessity for physical or temporal ordering of data items. Examples of applications in this category would include shared text editors, and object-based graphics or CAD editors. Distributed Operation algorithms should only be used in cases where local response time is critical and physical or temporal ordering is required. These applications might include real-time control systems, networked games, and shared text/graphics editors operated between remote locations.

References

[1] Chang, J.M., Maxemchuk, N.F., "Reliable Broadcast Protocols", *ACM Transactions on Computer Systems*, Vol 2, No. 3, pp. 251-273.

[2] Ellis, C.A., Gibbs, S.J., and Rein, G.L, "Groupware: Some Issues and Experiences", *Communications of the ACM*, Vol 34, No. 1, pp. 39.

[3] Ellis, C.A., Gibbs, S.J., and Rein, G.L. "Concurrency Control in Groupware Systems." *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, 1989.

[2] N = Number of concurrent users, T = transmission time, Q_t =Quiescence time interval, O_i = Operation size, P = previous state size.

[4]Greenberg, S., Roseman, M., Webster, D., & Bohnet, R. (1992). “Issues and experiences designing and implementing two group drawing tools”. Paper presented at the 25th Hawaii International Conference on Systems Sciences, Honolulu, HI.

[5] Greif, I., Sarin, S., "Data Sharing in Group Work", *ACM Transactions on Office Information Systems*, Vol. 5, No. 2., April 1987, pp. 187-211.

[6] Gutekunst, T., Bauer, D. Caronni, G., Hasan, Plattner, B., “A Distributed and Policy-Free General Purpose Shared Window System”, *IEEE/ACM Transactions on Networking*, Vol. 3, No. 1, Feb. 1995.

[7]Hayne, S., & Pendergast, M., Greenberg, S. “Implementing gesturing with cursors in group support systems.” *Journal of Management Information Systems*, *vol.10, no.3*(Winter), p43-61.

[8]Ishii, H., & Miyake, N. (1991). “Toward an open shared workspace: computer and video fusion approach of TeamWorkStation”. *Communications of the ACM, v34, n12*(Dec), p36(15).

[9] Knister, M.J., Prakash, A., “DistEdit, A Distributed Toolkit for Supporting Multiple Group Editors”, CSCW’90 Proceedings.

[10] Knuth, D.E., *Fundamental Algorithms*, Addison-Wesley, 1975.

[11] Kompella, V.P., Pasquale, J.C, and Polyzos, G.C., “Multicast Routing for Multimedia Communications”, *IEEE/ACM Transactions on Networking*, Vol 1. No 3., June 1993.

[12] Moran, T, McCall, K., Melle, C., Pedersen, E., Halasz, F., Xerox Parc, “Design Principles for Sharing in Tivoli, a Whiteboard Meeting-Support Tool”, *Groupware for Real-Time Drawing, A Designers Guide*, edited by Greenberg, Hayne, and Rada, McGraw-Hill, 1995

[13] Melliar-Smith, P.M., Moser, L.E., Agrawala, V., "Broadcast Protocols for Distributed Systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, January 1990, pp 17-25.

[14] Nunamaker, J. F., Jr., Dennis, A. R., Valacich, J. S., Vogel, D. R., & George, J. F. (1991). Electronic meeting systems to support group work. *Communications of the ACM, 34*(7), 40-61.

[15] Pendergast, M.O. "Multicast Channels for Collaborative Applications: Design and Performance Evaluation ," *ACM Computer Communications Review*, Vol 23, No. 2, April 1993.

[16]Roseman, M, Greenberg, S., “Building Real-Time Groupware with GroupKit, A Groupware ToolKit”, *ACM Transactions on Computer-Human Interaction,* Vol 3. , No. 1, March 1996.

[17] Stefik, M., Foster, G., Bobrow, D. G., Hahn, K., Lanning, S., and Suchman, L., "Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings,*" Communications of the ACM*. Vol. 30, No. 1, 1987. pp. 32-47.

[18] Sun, C. , Jia, X., Zhang, Y., Yang, Y., “Generic Operation Transformation Scheme for Consistency Maintenance in Real-time Cooperative Editing Systems”, *Proceedings of Group’97,* Phoenix Arizona.

[19]Tang, J. C., & Minneman, S. (1990). VideoDraw: a video interface for collaborative drawing. *ACM Transactions on Information Systems*, Vol. 9, No. 2, April 1991.