A PEER-TO-PEER COLLABORATIVE ENVIRONMENT

Mujtaba Khambatti, Mahesh Kamath and Thanigaivelu Elangovan

Computer Science and Engineering Department
Arizona State University
Tempe, AZ 85287-5406

Technical Report
May 2002

ABSTRACT

Peer-to-peer systems function without any centralized control. We have designed and implemented a peer-to-peer system that allows files to be shared amongst distributed users in a synchronized and consistent manner. We enable participating peer nodes to initialize a peer-to-peer network, and exchange initial information about the names of files that they own and are willing to share with the other members. Thus we form a collaborative group that enables member peer nodes to view and use files that are owned by any participating peer node. In order to participate in a collaborative group we require a new peer node to know the identity of only one existing member peer node.

Peer nodes automatically keep track of the location of the latest version of a shared file. This is accomplished by assigning the responsibility to the peer node that owns the file, i.e. the peer node that introduced the file into the group. Peer nodes allow their users to read, write and commit changes made to any shared file. The user is also given a choice of valid replica locations from where to download a file from so that the location that is closest can be selected.

We optimize on the number of file transfers that occur between peer nodes by only performing a transfer after comparing the local replica of a file with the original file. Additionally, we increase concurrency by granting read requests for a file during a write operation on the same file. The requesting peer node is however warned that the file was being written to. Finally, the use of local synchronization objects ensures sequential operations when needed.

This report describes the design of the algorithms and our implementation of them along with the required data structures in order to achieve these functionalities. At the end of the report we demonstrate the correctness of our algorithm by discussing the tests we carried out.

1. INTRODUCTION

Peer-to-peer systems have come to focus of late with the immense popularity of Internet based file sharing applications like Gnutella, Freenet and Napster. These systems have a lot of interesting technical aspects such as decentralized control, self-organization, adaptation and scalability. Peer-to-peer systems can be characterized as distributed systems in which all nodes have identical capabilities and responsibilities and all communication is symmetric.

These features make the concept of peers sharing files among themselves without relying on a server fascinating. The problem with the client server model is that it presents us with a single point of failure. A failure in the client server model could have catastrophic consequences if redundancy is not available. On the other hand, on a peer-to-peer system, the entire portion of the data is not lost. Only the data that has been stored on the peer that crashed gets lost. The rest of the system works as before.

We are trying to build a collaborative group where all the member peer nodes dedicate a folder on their hard disk to store shared files or replicas that can be accessed by other member peer nodes. We assume that the peer nodes in the group trust each other and there is no need for any authentication or encryption. Further, we assume that the files present in the dedicated folder can be viewed or modified by all member peer nodes except in the case that the files are not valid replicas or if a write operation is requested by a peer node for a file that is being written to by another peer node.

When a peer node modifies a file, all previous replicas of the file lose their status of being a replica and become old cached copies of the file. Subsequent requests to use these old cached copies of the file will cause our application to download the latest copy of the file so that the user always works on a valid replica. If replicas of files exist, they serve the purpose of increasing concurrent reads. Additionally, when peer nodes need to download the latest copy of the file, they can do so from any of the locations that hold a valid replica. The user makes this decision most likely based on the distance to each of the replica locations. We improve concurrency by allowing a read operation on a file that is being written to. In this case the user warned that the copy being read might not be valid for too long.

2. OVERVIEW OF THE SYSTEM

2.1 Data Structures

The member peer nodes of the collaborative group incur the overhead of maintaining information about all the member peer nodes and the files that they own. Additionally, the members also maintain information about the various locations where replicas of a particular file might exist. This information is critical to enable the algorithms to work efficiently. Since multiple threads might simultaneously access these data structures, we ensure mutual exclusion by using critical section objects for each data structure. We maintain the following data structures:

- A Replica Table associated with each file
- The Files Directory

2.1.1 Replica Table:

The Replica Table is the most important data structure of the collaborative group. This structure contains the list of all peer nodes that have a valid replica of a particular file. A separate replica table is maintained for each file present in the system. The only peer node whose replica table is guaranteed to be valid at all times is that of the owner of a file.

We have implemented the Replica Table as a doubly linked list. The first node in the list usually contains the identity of the owner of the file. The exception to the rule is if the owner no longer has the most valid replica. In this case, the identity of the peer node with the valid copy (i.e. the peer node who modified the file last) is present in the first node.

The purpose of the replica table is to keep track of the valid copies of the file at all times. This information is useful for the following reasons:

- A peer node can at any given time know where to look for the most valid replica.
- A peer node can take advantage of the location by choosing the nearest peer node to fetch the file from.
- Conflicts can be resolved by having the owner of a file as its coordinator.

The following is the structure of the Replica Table:

```
struct ReplicaTable
{
     char *fileName;                  // name of the file
     char *peerName;                  // name of peer node
     struct ReplicaTable *next;       // forward pointer
     struct ReplicaTable *prev;       // backward pointer
};
```

2.1.2 Files Directory:

The Files directory is maintained as two separate structures that contain a list of local and remote files. The remote files directory contains the list of all the remote files and their owners. This information is useful whenever a read or write permission for a remote file is required. A look up is performed on this data structure to determine the peer node that should receive the permission request. The local file directory has the list of the local files that the peer node owns. It additionally contains a flag for each file listed that indicates whether a write is being performed on the file. This data structure is implemented as a linked list.

The structure is given as follows:

```
struct FILES
{
     char *fName;                // name of file
     char *pName;                // name of peer node
     int WIP;                    // Write in progress flag
     struct FILES *next;         // forward pointer
};
```

2.2 Synchronization

Synchronization between threads to ensure sequential operation in certain situations is implemented using operating system defined Event Objects. Critical section objects, as earlier stated, control data access.

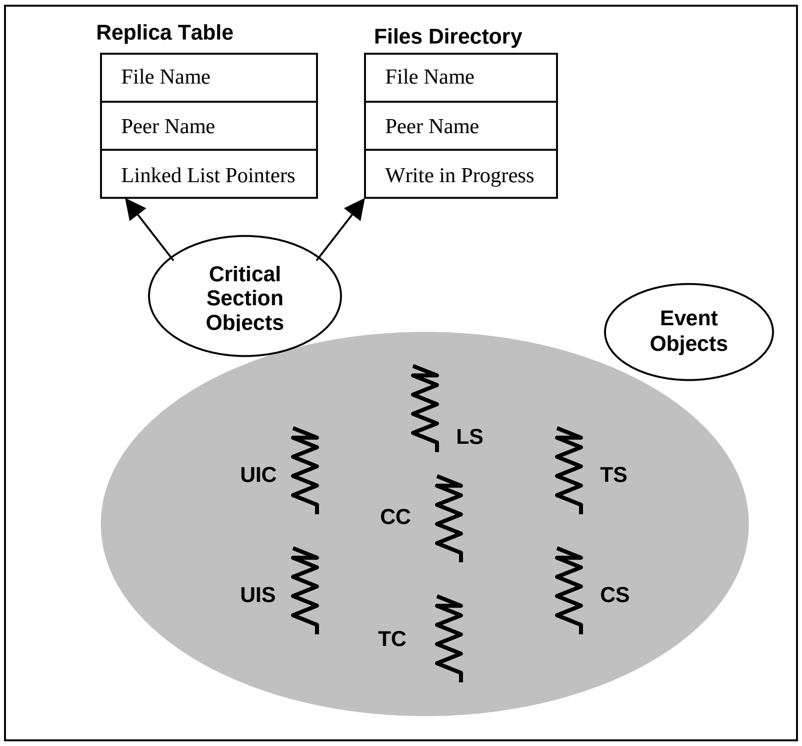

Figure 1: Logical View of the System Architecture

2.3 Graphical User Interface

The GUI for the application was developed in Microsoft Visual Basic. However, since the algorithms were programmed in Microsoft Visual C++, we had to converted the application into a DLL with some exportable functions that could be called from the Visual Basic front-end. The exportable functions are:

- `startApp`
- `quitApp`
- `ClientSideProcessingGetPeerName`
- `ClientSideProcessingConnect`
- `ClientSideProcessingListPeers`
- `ClientSideProcessingGetPeerListSize`
- `ClientSideProcessingGetConnectionStatus`
- `ClientSideProcessingGetFilesList`
- `ClientSideProcessingAddFile`
- `ClientSideProcessingGetFilesListSize`
- `ClientSideProcessingGetMyFilesList`
- `ClientSideProcessingGetMyFilesListSize`
- `ClientSideProcessingReadFile`
- `ClientSideProcessingWriteFile`
- `ClientSideProcessingFlushRT`
- `ClientSideProcessingWriteDone`
- `ClientSideProcessingGetWriteAllowed`

3. NETWORKING INFRASTRUCTURE

Each peer node that is a member of the collaborative environment is empowered as both, a server of files and also as a client that can request files from other peer node members. This arrangement allows peer nodes to function as either a server or a client depending on the role required at the time.

Our system is built to run on Windows 2000 based machines that are connected via Ethernet. We employ the TCP protocol provided by the operating system to carry out any communication. Consequently, all our communication is achieved through numerous *message passing* by the Windows Socket API. We additionally assume that no message is lost over the transmission link and that all communication is reliable.

The networking infrastructure has been implemented using a multiple thread model to achieve concurrent communication and processing. The first thread created on a peer node is the *Listener Server Thread* (LS). It is charged with listening for connection requests and taking the appropriate action. Following the creation of the LST, a *User Interface Client Thread* (UIC) is created. It sends a connection request to the LS and also identifies itself. On receipt of the connection request, the LS will create a *User Interface Server* Thread (UIS) to correspond to the identity of the requesting thread.

The responsibility of the UIC is to display the user interface and get any input from the user. Depending on the input, the UIC might perform some client-side processing and then send all the information obtained to the UIS.

The UIS receives the information and initiates some action based on the input from the user. If the user had requested to connect to another peer node, the UIS will create a *Temporary Client Thread* (TC) that sends a connection request to the LS on the remote peer node. Based on the identity of the requesting thread, the LS creates a *Temporary Server Thread* (TS) within its process space to communicate with the TC on the requesting peer node's machine. The TC and the TS execute the Join Algorithm (described in Section 4.1).

Although in our implementation of the system, we have kept the connection between the TC and the TS alive, this is not necessary. When a peer node requests to be linked to another peer node, the connection is used to start the execution of the Join Algorithm. After this, the connection is no longer needed and the two peer nodes can remain disconnected from each other.

Collaboration amongst peer nodes takes the form of:

1. A peer node needing to serve a local file or replica to a remote peer node.
2. A peer node needing to request a copy of a remote file or replica.

In both these cases, a temporary connection is established between the two peer nodes by means of a *Collaborative Client Thread* (CC) and a *Collaborative Server Thread* (CS). The connection is kept alive for the duration of the file transfer and then broken down. Therefore, the networking infrastructure can provide a good peer-to-peer solution in wireless networks.

Any peer node that wants to join an existing collaborative group must set the port number of its Listener Server equal to the port number of Listener Servers belonging to the existing member peer nodes of the collaborative group. By virtue of this design, the infrastructure allows a peer node to participate in more than one collaborative group. This can happen if a peer node runs more than one instance of the application and specifies distinct port numbers for each Listener Server.

4. ALGORITHMS

4.1 Join Algorithm

If a peer node wants to join an existing collaborative group, it needs to know the machine name of exactly one peer node that is already a member of the group. The peer nodes will connect to each other and execute the Join Algorithm. At the end of the algorithm, all the members of the collaborative group will know about the new member and the files that it has to share, and the new member will know about all the existing members and which files they have to share.

This algorithm is only meant as a protocol for peer nodes that are not a member of a collaborative group to join it by connecting with exactly one peer node that is already a member of the group. It is not designed to allow member peer nodes to connect to non-member peer nodes in order to bring them into the collaborative group.

On receipt of the user's request to link to a remote peer node, the *User Interface Server Thread* (UIS) creates a *Temporary Client Thread* (TC) that sends a connection request to the *Listener Server Thread* (LS) on the remote peer node. A *Temporary Server Thread* (TS) is formed as a result of that connection request.

The TC begins the Join Algorithm by sending its computer name to the TS. In return, the TC receives a list of the computer names of the existing member peer nodes. Next, the TC creates separate threads called *Join Client Thread* (JC) that establishes a connection with each of the peer nodes in the list. A *Join Server Thread* (JS) is created at each member peer node to correspond with the JCs on the non-member peer node.

A JC will access a global data structure that maintains a list of all the files that are available locally at the non-member peer node. This list of local files is sent to the corresponding JS. The JC then receives a list of files that are available at that remote peer node. All the lists of remote files that are received by all the JCs are collectively stored in another global data structure. Global data structure access is mutually exclusive and is controlled by operating system critical section objects. Connections are broken when the Join Threads have completed their execution.

4.1.1 Replica Tables and their use in the Join Algorithm:

The Replica Table is not directly used to exchange information in the join algorithm. However, before a peer node is ready to join the network, all the local files must be registered by creating replica tables for each file. A structure called FILES maintains a data structure containing the identification of all member peer nodes and their files. It is actually a similar to the replica table. The use of this data structure is only to identify the owner of a file at any given time. During join, the JCs will exchange their local copies of FILES. This helps the algorithms described in the following sections to share files with other peer nodes that are members of the collaborative group. The FILES data structure is updated each time a new peer node joins the group.

4.2 Node that owns a file or a replica

This algorithm is implemented inside the *Collaborative Server Thread* (CS) of a peer node. As stated earlier in Section 3, the CS is created when a remote peer node requests a file or a replica that is stored locally on the current peer node. On creation, the CS receives the machine name of the remote peer node, the type of request and the name of the file being requested. In some cases, the algorithm needs to compare two files. Instead of using a hash function, our implementation makes use of the file size. Albeit, comparing the sizes of two files is not a good indicator of their similarity, it is a simplification that does not change the demonstration of the algorithm's correctness.

The following table serves to tabulate the actions performed by the CS depending on the type of request:

Type of Request	Action Taken
New Read (requesting peer node does not have a replica of the file)	Check the replica table of the file to see if any other peer node has a valid replica of the file. If so, send the replica table associated with the file to the remote peer node. This allows the user on the remote peer node to select a location from where to download the file. On receiving the name of the selected peer node, the CS will compare it with its own computer name. If they are the same, the CS will send the file to the remote peer node. If the file was being written to, an additional message will be sent to the peer node indicating that the file recently sent is not the latest copy. However, if no valid replica copies of the file exist on other peer nodes, the CS will send the file to the remote peer node. If the file was being written to, an additional message will be sent to the peer node indicating that the file recently sent is not the latest copy. Finally, a new entry is made in the local copy of the replica table associated with the file to indicate the new replica location.
Quick Read	The CS sends the file and its associated replica table to the remote peer node. Additionally, a new entry is made in the local copy of the replica table associated with the file to indicate the new replica location.
Old Read (requesting peer node has a copy of the file but it might not be a valid replica)	The CS receives a number indicating the size of the file in bytes and then checks the replica table associated with the file to see the recorded size of its latest valid replica. If the file sizes are the same, it indicates that the remote peer node has a valid replica of the file. The remote peer node is sent a message indicating that a local read operation on the replica may continue. If the file was being written to, an additional message will be sent to the peer node indicating that the file is being modified at another peer node. However, if the file sizes are not the same, the CS sends the replica table associated with the file to the remote peer node. This allows the user on the remote peer node to select a location from where to download the file. On receiving the name of the selected peer node, the CS will compare it with its own computer name. If they are the same, the CS will send the file to the remote peer node. If the file was being written to, an additional message will be sent to the peer node indicating that the file recently sent is not the latest copy. Finally, a new entry is made in the local copy of the replica table associated with the file to indicate the new replica location.

Write	If the file is currently being written to, the CS sends a message to the remote peer node indicating that it should try again later. Else, the CS receives a number indicating the size of the file in bytes and then checks the replica table associated with the file to see the recorded size of its latest valid replica. If the file sizes are the same, it indicates that the remote peer node has a valid replica of the file. A message is sent to the remote peer node indicating that it can proceed with the write operation. The replica table associated with the file is modified to indicate that the file is being written to. However, if the file sizes are not the same, the CS checks to see whether a valid replica of the file exists locally. If so, it sends the file to the remote peer node and a new entry is made in the local copy of the replica table associated with the file to indicate the new replica location and also that the file is being written to. Else, if a valid replica of the file does not exist locally, the CS gets the machine name of the peer node that has a valid replica of the file and sends that to the remote peer node. A new entry is also made in the local copy of the replica table associated with the file to indicate the new replica location and also that the file is being written to.
Unlock	The CS receives a number indicating the size of the file in bytes and then checks the replica table associated with the file to see the recorded size of its latest valid replica. If the file sizes are the same, it indicates that the remote peer node did not modify the file. The write flag in the local replica table associated with the file is reset. However, if the file sizes are not the same, the local replica table is flushed except for the entry that indicates the remote peer node. Therefore the remote peer node becomes the only peer node that has a valid replica of the file. Additionally the write flag associated with the file is reset.

4.3 Node that wants to use a file

This algorithm is implemented inside the *Collaborative Client Thread* (CC) of a peer node. As stated earlier in Section 3, the CC is created when a remote peer node requests a file or a replica that is stored locally on the current peer node. On creation, the CC finds the machine name of the remote peer node that owns the file. It sends the remote peer node, the local computer name, the type of request and the name of the file being requested. Once again here, the algorithm needs to compare two files. As mentioned in Section 4.2, instead of using a hash function, our implementation makes use of the file size. (See explanation in Section 4.2)

The following table serves to tabulate the actions performed by the CC depending on the type of request made and the response received:

Request made	**Response Received**	**Action Taken**
New Read (peer node does not have the replica)	**Replica Table**	Display a list of different locations holding valid replicas of the file and ask the user to select one. Send the location name to the owner of the file. If the location selected was the owner peer node, receive the file from the owner. If an additional message was received with the file, display a warning to the user that a write was in progress.

		However, if the location selected was that of another peer node, then the CC would create a new thread that would connect to the peer node selected and send it a *Quick Read* request for the file.
New Read	**File**	Additionally the peer node would receive the replica table from the owner. The peer node can go ahead with the read operation.
New Read	**File + Special**	Additionally the peer node would receive the replica table from the owner. The peer node can go ahead with the read operation. However the user is warned that the file is being written to elsewhere.
Old Read (peer node has a copy of the file)	The peer node would have to also send the size of the file in bytes to the owner peer node.	
Old Read	**OK**	The peer node can go ahead with the read operation.
Old Read	**Special**	The peer node can go ahead with the read operation. However the user is warned that the file is being written to elsewhere.
Old Read	**Replica Table**	Display a list of different locations holding valid replicas of the file and ask the user to select one. Send the location name to the owner of the file. If the location selected was the owner peer node, receive the file from the owner. If an additional message was received with the file, display a warning to the user that a write was in progress. However, if the location selected was that of another peer node, then the CC would create a new thread that would connect to the peer node selected and send it a *Quick Read* request for the file.
Quick Read	**File**	The peer node can go ahead with the read or write operation.
Write (peer node does not have the replica)	The peer node would have to also send the size of the file as zero bytes to the owner peer node.	
Write	**Replica Location**	The CC would create a new thread that would connect to the peer node selected and send it a *Quick Read* request for the file.
Write	**File**	Additionally the peer node would receive the replica table from the owner. The peer node can go ahead with the read operation.
Write	**Special**	The file is being written to by another peer node. The user should try again later.
Write (peer node has a copy of the file)	The peer node would have to also send the size of the file in bytes to the owner peer node.	
Write	**OK**	The peer node can go ahead with the write operation.

Write	**Special**	The file is being written to by another peer node. The user should try again later.
Write	**Replica Location**	The CC would create a new thread that would connect to the peer node selected and send it a *Quick Read* request for the file.
Write	**File**	Additionally the peer node would receive the replica table from the owner. The peer node can go ahead with the read operation.
Unlock (sent after the write operation)	The CC calculates the size of the file in bytes and then checks the replica table associated with the file to see the recorded size of the previous valid replica. If the file sizes are not the same, it indicates that the peer node modified the file. The local replica table is flushed except for the entry that indicates the current peer node. Therefore the current peer node becomes the only peer node that has a valid replica of the file. The write flag in the local replica table associated with the file is reset.	

5. DEVELOPMENT METHOD

Building a collaborative group that enables sharing of files amongst its members required a very well thought out design that was clean and able to capture all the scenarios that might occur. Prior to this, it was imperative that the policies of such a system be placed on the table so that design decisions could be made with common knowledge. The system was entirely developed in C++ using the Win32 API.

Soon after the design was made, we began work on the development of the individual parts of the system. The networking infrastructure formed the base of all higher-level operations. That was hence tackled first and thoroughly tested over distributed machines. The networking infrastructure was then augmented with the Join Algorithm. The resulting application allowed peer nodes to connect to one another and reach a state where all peer nodes knew about each other's existence within the group and also knew about the list of all files available to the group.

Simultaneously, development was being carried out to create a graphical user interface for the application. The Microsoft Visual Basic (VB) environment was the most conducive to GUI programming and was therefore picked to implement this goal. The approach used involved converting the existing application into a dynamically linked library (DLL) that exported certain functions. This DLL was then loaded into the VB applications where calls were made to the exported functions in order to achieve the same behavior as without the GUI code.

Once the Join Algorithm was completed and tested, the other algorithms had to be programmed into the application and tested. A simple command-line application was created that demonstrated the correctness of the algorithms. However, this application did not work with replica tables and further, instead of actually sending and receiving a file, it sent or received the file name.

While the GUI continued to be built, the replica table data structure was added into the existing command-line application and the functionality to actually send or receive the actual file was also programmed.

6. TESTS CARRIED OUT

We tested our collaborative group system on Windows 2000 machines. Due to the limitation of the number of machines available, we could only test our system thoroughly using 3 computers connected via a 100 Mbps Ethernet LAN. On an earlier occasion, we had the opportunity to successfully stress test the Join Algorithm with 5 computers. Since our goal was to create a collaborative group for a small number of peers, we did not see the need to extensively stress test our other algorithms, albeit the only increase we foresee would be the memory requirement for each peer if number of peers were increased.

Our objective in carrying out the testing process was to demonstrate the correctness of the algorithms that we had designed. Therefore, the reader will find below a list of use-cases. Experiments to observe the performance of our system has been left for future work.

The list of Use-cases follows:

1	NEW PEER JOIN	This is the result of the execution of the Join Algorithm of Section 4.1
2	VANILLA READ	Show that a peer can read a file that belongs to another peer.
3	VANILLA WRITE	Show that a peer can write to a file that belongs to another peer.
4	UPGRADE	Show that a peer that has ‘Read’ permission for a file can upgrade to ‘Write’
5	READ FROM REMOTE	Show that a peer can read a remote file by fetching it from one of its replica locations.
6	WRITE FROM REMOTE	Show that a peer can write to a file by fetching it from one of its replica locations.
7	BECOME SOLE OWNER	Show that after a peer writes to a file, it becomes the owner of the only primary copy.
8	FAILED WRITE	Show that a ‘Write’ request by a peer fails if another peer was writing to the same file.
9	READ WITH WARNING	Show that a peer can get ‘Read’ permissions for a file that is currently being written to.
10	CHOICE OF LOCATION	Show that a peer can choose a location from where to fetch the replica of a file.

7. FUTURE WORK

In the current implementation, the peer nodes that join the collaborative group have to know exactly one existing member peer node in order to join the group. The first step in expanding this application would be to enable a peer node to dynamically discover a peer node that is a member of a collaborative group. One can take this further, by enabling the peer node to discover the different collaborative groups that exist, find at least one member in each of these groups so that the peer node can join any of these groups. Solutions to this dynamic discovery problem are trivial if centralized approaches can be used. A peer-to-peer discovery poses a challenge that might be interesting to solve.

Currently, the peer nodes trust each other completely. At a later stage, security issues could be solved in order that peer nodes can form collaborative groups with un-trusted peer nodes.