THE

UBC DISTRIBUTED CONTINUOUS MEDIA FILE SYSTEM: Internal Design of Server[1]

Dwight J. Makaroff, Norman C. Hutchinson, and Gerald W. Neufeld
Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1Z4, Canada

July 19, 1995

Abstract

This report describes the internal design of the UBC Distributed Continuous Media File Server as of April 1995. The most significant unique characteristic of this system is its approach to admission control which utilizes the time-varying requirements of the variable bit-rate data streams currently admitted into the system to properly allocate disk resources. The structure of the processes which implement the file server are described in detail as well as the communication between client processes and server processes. Each major client interface interaction is covered, as well as the detailed operation of the server in response to client requests. Buffer management considerations are introduced as they affect the admission control and disk operations. We conclude with the status of the implementation and plans for completion of the design and implementation. ***This document provides a snapshot of our design which has not yet been fully implemented and we expect to see significant evolution of the design as the implementation proceeds.***

---

1. This work was supported by grants from the Natural Sciences and Engineering Research Council of Canada and the Canadian Institute for Telecommunications Research.

1.0 Introduction

This document describes the algorithms and the communication between the various internal components of the CMFS. Additionally, the design issues of stream admission control and buffer management are discussed which permit the server to function effectively in an environment constrained by limited bandwidth disk devices and limited memory space for buffering.

2.0 Overall Communication Structure

For an initial architecture, we envision one Administrator node and 2 worker nodes. The number of worker nodes is theoretically limited by the cumulative transfer rate from the disk storage device. When this matches the network bandwidth at the various connections, no additional performance improvement is possible. These are implemented as UNIX processes which have their own IP Address, port number pair for communication. Nodes can be mapped 1-1 onto machines or address spaces within machines. In the prototype, a node is equivalent to a UNIX process.

The system is built on top of RT Threads[1], a user-level threads package built at UBC to support real-time, parallel, and/or distributed applications. The transport level communication facility is provided by a user-level implementation of XTP [3][4] (the Xpress Transport Protocol), which is also built using RT Threads.

In this section, the activity generated at the server in response to calls by a client application using the CMFS Interface is fully described. The CMFS Interface itself is described in [2]. The data structures passed in messages or provided as parameters to RttCreate calls are described in Section 2.8.

2.1 Initialization

Figure 1 shows the communication that takes place on system initialization. Each worker node creates a Disk Manager thread for each disk on that node. This thread performs admission control, reads the disk schedule and performs the actual disk I/O operations. It will be described further in following sections. The worker node sends a message to the administrator signifying that it is ready for client requests (1). The administrator node replies back to the worker nodes (2) to allow them to continue.

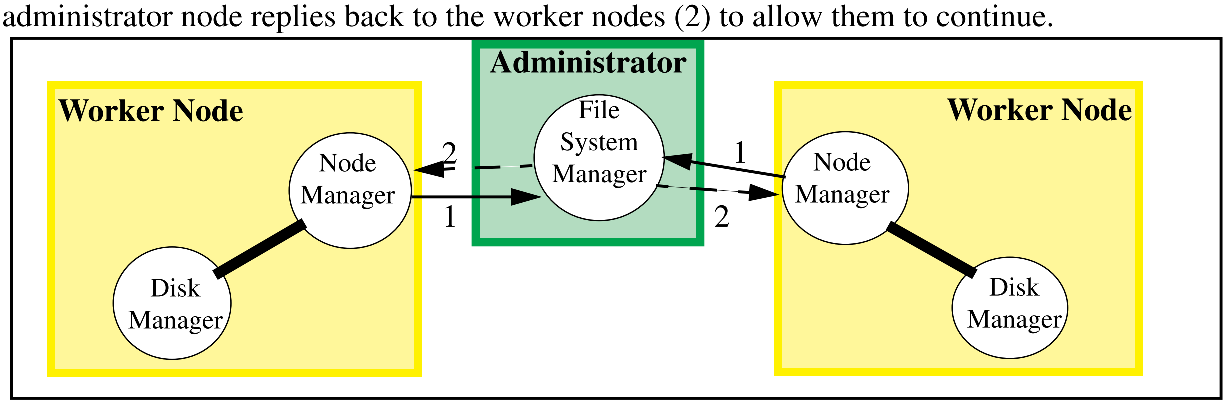

FIGURE 1. Structure and Initialization

The administrator node is started using the following UNIX command:

```
% cmfsadm IPAddress
```

The nodes are started via the following UNIX command:

```
% cmfsnode localIPAddress serverIPAddress
```

A specific port number is assigned to the administrator node in the configuration header file `config.h`. The other processes may allow the system to choose a port number. A client application could use the machine name of the administrator node if that machine has only one IP Address for communication to the server.

If there is no existing database for meta information, the administrator node creates an empty one. Additionally, if there is no disk space allocated for the CMFS, each worker node allocates the necessary storage.

2.2 Opening a Connection

Figure 2 shows the communication that takes place on an open request.

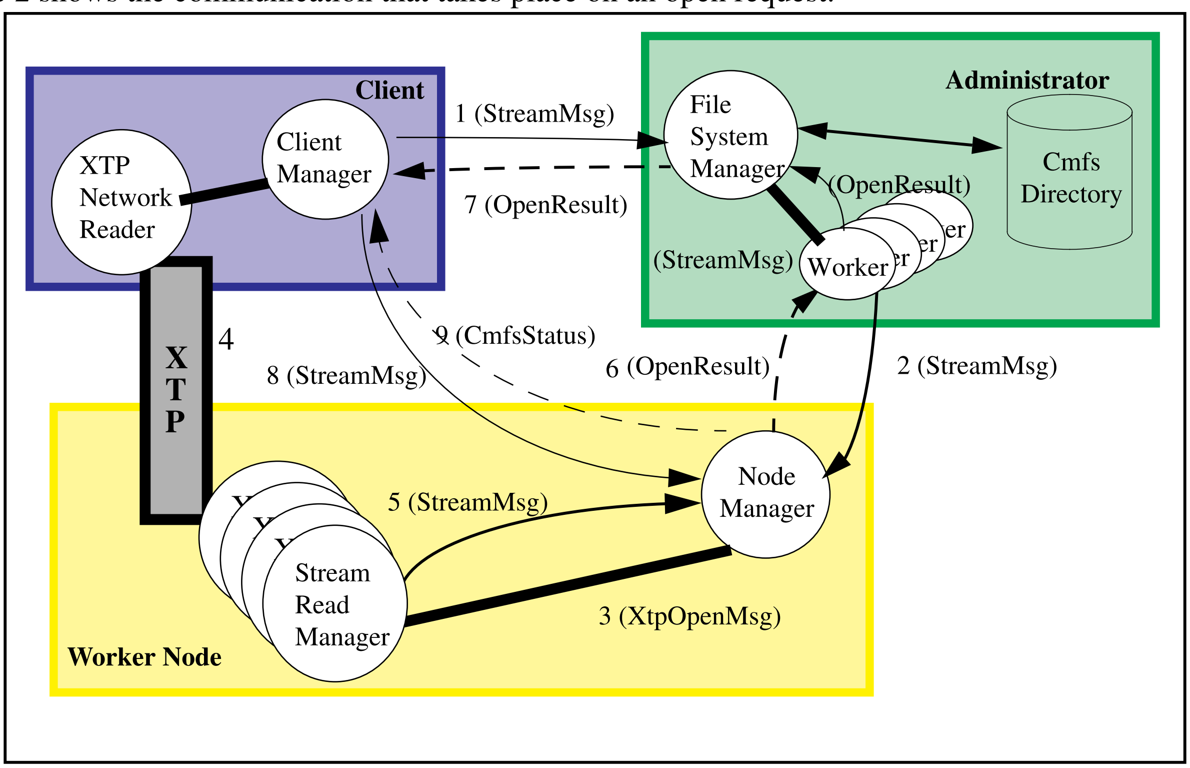

FIGURE 2. Structure and Communication on CmfsOpen/CmfsClose

When the client makes a request for an object to the administrator node (1), the client constructs the pid (Thread Identifier) for the File System Manager from the UOI (this is done inside the CmfsOpen routine). Initially, there is only one administrator node in the entire system. The File System Manager thread creates a worker thread which sends the request to the node manager that has the object (2). This is determined by looking up the UOI in the CMFS directory in some fashion. If the object does not exist in the directory, a corresponding failure status is returned immediately. The Node Manager creates an XTP Stream Manager Thread for the object (3), passing it an XtpOpenMsg structure. This thread actually establishes the XTP connection to the XTP port on the client machine (4). If the initial negotiation regarding the connection is successful, then a reply message gets sent back to the Node Manager (5). The node manager sends back a confirmation (6) to the Administrator worker node, and finally, the client call to CmfsOpen returns as the File System Manager sends back its confirmation (7). Included in the confirmation is the PID of the Node Manager to be used by the client for further stream control requests.

If the connection is not acceptable, or any other failure occurs during processing at the server, then indication of failure is propagated back to the client.

When a connection is closed, the communication is made directly to the Node Manager, and the Stream Read Manager is killed once its buffers have been flushed.

2.3 Preparing An Object

The communication on a prepare request is shown in Figure 3.

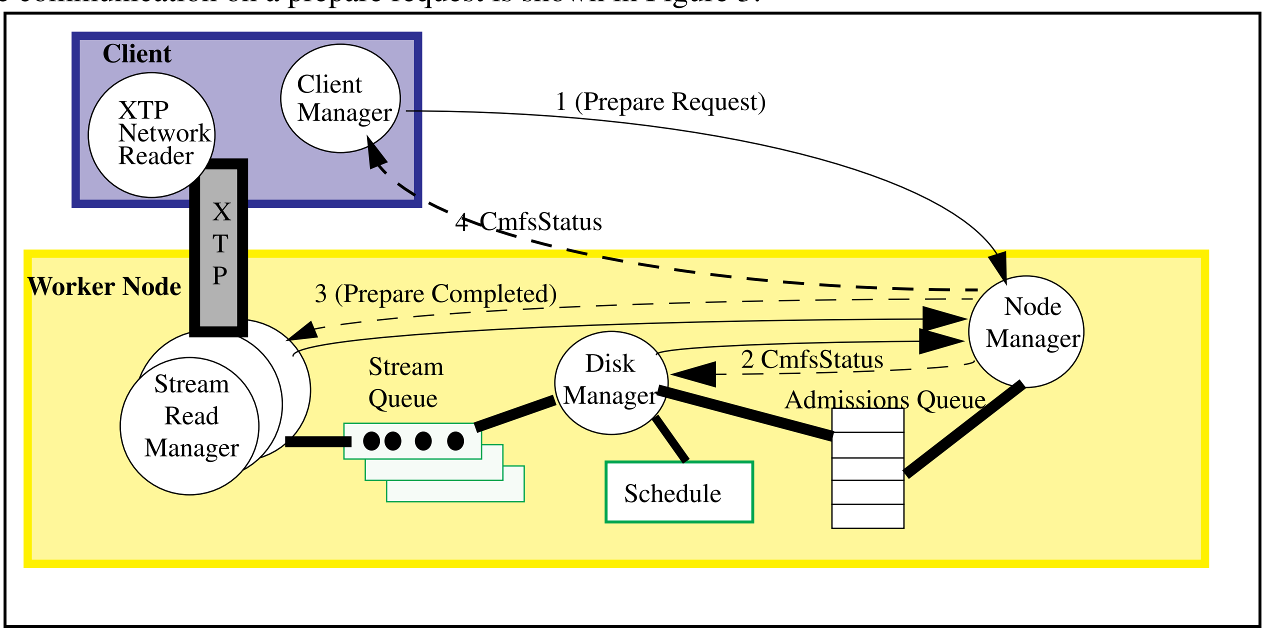

FIGURE 3. Structure and Communication on CmfsPrepare

The client sends its request directly to the node manager (1). The request is put on the Admissions Queue. The disk manager examines the admissions queue after every read and enqueue operation that is performed. The disk manager attempts to schedule the requested sequences of the object in accordance with the parameters to CmfsPrepare. If the object can be scheduled, then a positive response is sent to the node manager (2) and the schedule is updated. The disk manager continues to dequeue prepare requests until the admissions queue is empty before going on to perform the reads for the next slot.

Actual transfer of data begins once the disk schedule is not empty. The mechanics of this data transfer depend on how the parameters were specified in the prepare request and the size of the initial buffer requested by the details of the XTP communication. If there are disk requests in the schedule, the disk manager performs those requests as in Section 4.4. The disk manager begins filling up buffers on the server in anticipation of CmfsRead requests from the client. The Stream Read Manager for that stream dequeues the buffers and transmits them on the appropriate XTP connection as described in Section 4.5. If the transport level window is not opened sufficiently, the requests will back up at the Stream Read Manager and the queue that the disk manager uses for this stream will fill up. If this happens, a solution may be to destroy the connection, or evaluate more equitable methods of sharing buffer space among the streams.

Once the node manager is informed that the data requested as the first two slots of the initial buffer has been transmitted (3), CmfsPrepare completes by sending back the response to the client (4).

2.4 Stopping Delivery of an Object

Figure 4 shows the communication on the stopping of stream delivery.

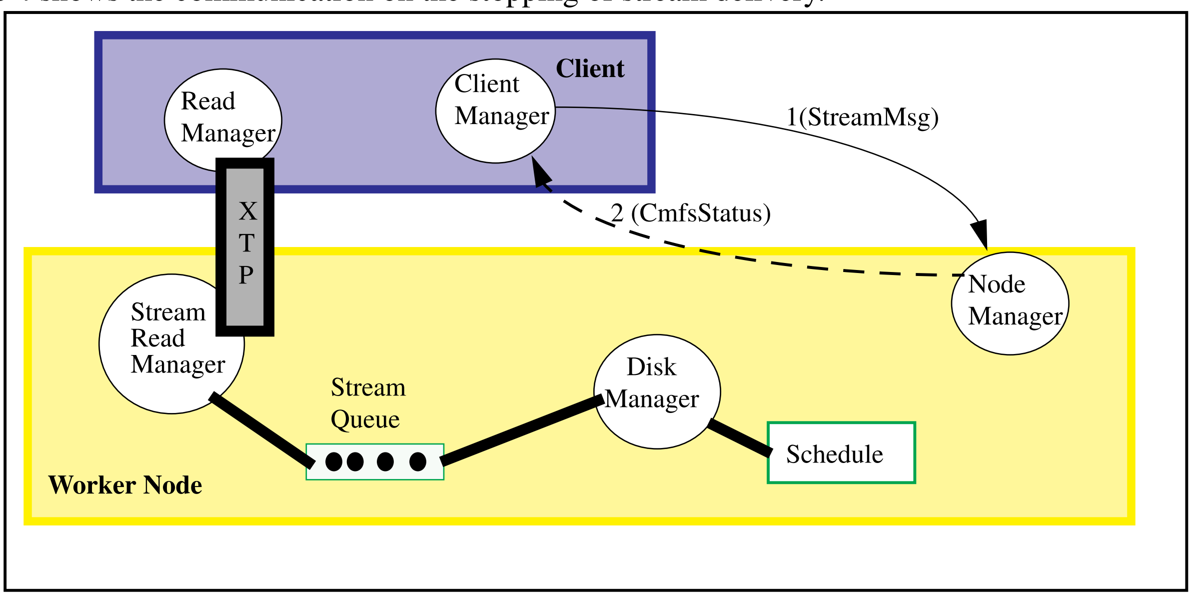

FIGURE 4.Structure and Communication on CmfsStop

The stop message (1) is sent directly to the Node Manager, which sets the state of the stream to stopping, and thereby causes the disk manager to remove the object from its active list. The XTP Stream Manager notices this change in state and the buffers are flushed without sending them across the network connection. Finally, the status is returned to the client (2). The code at the client which implements CmfsStop also throws away buffers that have been sent by the network layer, but not yet consumed at the client via CmfsRead before returning control to the application.

2.5 Reading/Changing Attributes of An Object (CmfsGetAttr/CmfsPutAttr)

The File System Manager thread at the administrator node handles calls to CmfsGetAttr and CmfsPutAttr. The administrator node holds the CMFS directory. Directory entries will have the following format:

<oid$attrkey, non-basicattr>

The Administrator Node makes the tdbm calls to retrieve or update directory information. Client applications, client stub code, and worker nodes may make use of the attribute facility.

2.6 Creating an Object (CmfsCreate/CmfsWrite/CmfsComplete)

When a client application wishes to create an object, the interaction is similar to what happens on CmfsOpen (See Figure 2) . The File System manager receives a message from the Client Manager, which sets up the initial attributes and determines on which node the object will be stored. The File System Manager sends a message to the Node Manager, which creates a per stream **Write Manager thread** for that new object and returns control back to the Administrator and then to the client. The PID of the Write Manager thread is returned in the reply message to the client and then the client writer thread and the server Write Manager thread engage in S/R/R to store the data for the object.

The write manager waits to receive messages from the client. The write manager writes disk blocks as it receives them across the network, using the DSM interface (see section 4.3). It is the responsibility of the Write Manager to store the list of sequences as attributes. As well, the specific unit sizes provided in each CmfsWrite message must be stored in the database.

When CmfsComplete is called, the client sends a message to the write manager thread. The write manager thread stores the proper sequence information in the attributes and returns control when the attributes have been stored (or an error occurs).

2.7 Removing an Object

To remove an object, the client application must call CmfsRemove. This will send a message to the administrator, which will determine the worker node responsible for that object. A worker thread at the administrator will foward a message to the worker node. The worker node will determine which blocks of the disk device contain that object and free them so that the space is available for other streams to be written to the disk device. It may be possible to have the server perform its own archiving policy by removing streams that have not been accessed in a long while. This is possible if an off-line archival storage system is setup for permanent backing store.

2.8 Structures for Messages/Thread Creation

This section describes which message types are sent from thread to thread and/or from address space to address space.

The C language structures in the CMFS are defined as follows (the list is incomplete):

```
typedef struct {
      u_int       length;     /* number of characters in the key */
      u_char      *key;       /* actual value of the key */
} datum;

typedef struct
{
      RttTimeValue time;
      u_long      units;
      u_long      startBlock;
      u_long      startOffset;
      u_long      endOffset;
      u_long      length;
} SequenceEntry;

typedef struct
{
      ipAddr location;        /* client address */
      u_long port;            /* UDP port for client */
      u_long xap;             /* XTP access port */
} StreamReq;


typedef struct StreamDescriptor
{
      u_long init;            /* initial buffer reservation request */
      u_long windowSize:      /* size of Transport Level buffer */
      u_long avgBitRate;
      u_long avgPeriod;       /* time over which avg bit rate calculated */
      u_long maxBitRate;
} SD;

typedef struct
{
      CmfsStatus        resultCode;
      u_long            cid;
      RttTimeValue      prepareBound;
      RttTimeValue      maxReadDelay;
      PID               nodePid;
} OpenResult;

typedef struct
{
      UOI         uoi;
      StreamReq   stream_req;
} OpenRequest;

typedef struct
{
      int         numUnits;
      u_long length;
      UOI         uoi;
```

```
} CreateRequest;

typedef struct
{
      CmfsStatus  resultCode;
      UOI         uoi;
      int         numUnits;
      u_long      cid;
      u_long      length;
      PID         nodePid;
} CreateRes;

typedef struct
{
      CmfsStatus  status;
      u_long      cid;
} Other;

typedef struct
{
      UOI   uoi;
      int   keyLength;
      char  keyValue[MAXKEYLENGTH];
      int   valueLength;
} Attr;

typedef struct
{
      UOI   uoi;
      int   keyLength;
      char  keyValue[MAXKEYLENGTH];
      int   attrLength;
      char  attrValue[MAXATTRLENGTH];
 } DoubleAttr;

typedef struct
{     u_long      cid;
      pos         startPos;
      pos         stopPos;
      int         speed;
      int         skip;
      RttTimeValue delayTime;
} Prepare;

typedef struct
{
      StreamReq   sp;
      UOI         uoi;
      u_long      connId;
} XtpOpenMsg;

typedef struct
{
      int         type;
```

```
      u_long      length;
      char        buffer[MAXWRITESEQ];
} WriteMsg;

typedef struct
{
      int         type;
      int         units;
      u_long      cid;
      u_long      length;
      u_long      uSizes[MAXUNITS];
} WriteCtlMsg;

typedef struct
{
 u_long type; /* request type */
 union
 {
      OpenRequest open;
      CreateRequestcreatereq;
      Other       other;
      Attr        getattr;
      DoubleAttr  putattr;
      Prepare     prepare;
      OpenResult  ores;
      CreateRes   cres;
      PID         nodePid;
 } parms;
} StreamMsg;
```

2.8.1 Node Registering

From Node Manager to Administrator: StreamMsg (nodePid).

2.8.2 Open Connection

From client to administrator: StreamMsg (OpenRequest).
From Administrator to Admin Worker Thread: StreamMsg (OpenRequest)
From Admin Worker thread to Node Manager:
StreamMsg (OpenRequest)
Entire Sequence List and Unit Schedule as attribute
From Node Manager to XTP Manager: XtpOpenMsg
From XTP Manager to Node Manager (on Open Completed): StreamMsg (other).
From Node Manager to Admin Worker Thread: StreamMsg (OpenResult)
From Administrator to Client: StreamMsg (OpenResult)

2.8.3 Prepare Stream

From Client to Node Manager: StreamMsg (Prepare)
From Node Manager to Schedule Manager: StreamMsg (Prepare)

From Schedule Manager to Node Manager: StreamMsg (other)
From XTP Stream Manager to Node Manager: StreamMsg (other)
From Node Manager to XTP Stream Manager: CmfsStatus
From Node Manager to Client Manager: CmfsStatus

2.8.4 Stop Stream

From Client to Node Manager: StreamMsg (other)
From Node Manager to Client: (CmfsStatus)
From Node Manager to Schedule Manager: StreamMsg (other)
From Schedule Manager to Node Manager: StreamMsg (other)

2.8.5 CmfsRead

No messages are transmitted across the network at the application level. There are low level network reservation requests transmitted if the transmit window size falls below the proper level.

2.8.6 Attributes

CmfsPutAttr:
From Client to Administrator: Stream Msg (putattr).
From Administrator to Client: StreamMsg (other).

CmfsGetAttr;
From Client to Administrator: StreamMsg (getattr).
From Administrator to Client: attribute value.

2.8.7 Positioning

CmfsGetPosition, CmfsTime2Pos, CmfsPos2Time
    Straight RPC.

2.8.8 Write Interface

CmfsCreate:
From client to Administrator: StreamMsg (CreateRequest).
From Administrator to Node Manager: StreamMsg (CreateRequest).
From Node Manager to Write Manager: StreamMsg (CreateRequest).
From Node Manager to Administrator: StreamMsg (CreateResult).

CmfsComplete:
From Client to Write Manager: StreamMsg (other)
From Write Manager to client: status code
From Write Manager to Administrator: server defined attributes
From Administrator to Write Manager: status code

CmfsWrite:
From Client to Write Manager: WriteCtlMsg and WriteMsg;
From Write Manager to Client: CmfsStatus.

3.0 Administrator Node

```
Basic Logic:
1) Loop forever
  2) Receive message
  switch message type
  3) Case Register:
        update structure of nodes which are accessible
  4) Case Deregister:
        update structure of nodes which are accessible
  4) Case Open:
        Create a worker thread passing it an OpenMsg structure
  5) Case Create:
        create a worker thread with a CreateReq structure
  6) Case GetAttr:
        Make tdbm retrieval call.
  7) Case PutAttr:
        Make tdbm update call
  8) Case OperationCompleted:
        Reply to worker thread sending original message
        Reply back to the client with appropriate message contents
end loop
```

Admin Worker Thread:

```
Basic Logic:
1) Send message to Node Manager.
2) Send message to Administrator Manager.
3) Exit.
```

The creation of worker threads for each request is optional. A pool of worker threads (optionally of different types) could be statically created which would constantly be waiting for work to do. If a static pool of worker threads is implemented, each thread would execute an infinite loop of steps 1 and 2.

4.0 Worker Node

4.1 Admission Control

Each node has autonomous control over the admission of streams into the schedule for objects which are stored at that node. This admission is based on two major factors: the maximum disk bandwidth at each disk at the node and maximum number of server buffers available on the node at any given time throughout the lifetime of the stream. This admission is complicated by the fact that the number of client buffers at each client affects the number of server buffers utilized by the current mix of streams.

Additionally, the network resources available provide a constraint on admissability. Network bandwidth is reserved at stream open time, so this constraint applies only indirectly. It is assumed that a request for CmfsOpen will be charged to the user such that the bill is for the life of the connection even if no data is transferred for long portions of time. If this is the case, it makes sense to reserve the network bandwidth. Otherwise, another method of creating a stream connection must be used, which will greatly increase the latency in beginning to present the stream.

CmfsPrepare schedules the disk reads for the stream being considered for presentation. The time interval of the presentation is divided into fixed time intervals called slots. Disk reads are performed on a slot basis, and each disk can guarantee a fixed number of reads per slot specified as *MINIO*. This number is used to determine if the data requirements of a stream can be accommodated by the node. Part of the data for the stream is a list of the number of buffers required per slot per disk over which the file is stored. This is calculated at prepare time as a 2-dimensional array as in Table 1.

Slots **Disk ID**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	3	3	3	2	4	5	6	2	5	1	0	0	1	5	4
1	1	2	5	4	2	1	3	0	0	2	1	2	0	2	1
2	2	5	4	2	1	3	2	2	2	1	3	2	0	2	0

Table 1: Individual Stream Requirements

4.1.1 Original Algorithm (Conservative)

Scheduling a stream (S) consists of the following activity:

```
 for each slot
      for each disk
            If the amount currently committed to read for this disk in this slot
                  plus the amount for S is greater than the minimum guaranteed
      then return (reject stream)
end for
```

This is a very conservative admission control scenario, because any disk idle time is not used for reading ahead, and therefore, any temporary peaks in the stream bandwidth requirements above the per slot minimum will result in a rejection, when many of the previous intervals would not have used the entire time to transfer data. If the data could have been read ahead of time, this fluctuation would not have violated the disk bandwidth capacity of the server. Thus, reading ahead during under-utilized slots can smooth out the reading requirements and allow more streams to be accommodated.

4.1.2 Read Ahead Buffering at Server (Concepts and Questions)

This section describes in general tems the constraints and concerns of our approach to admission control. The concrete realization of these goals is formalized in the “Detailed Algorithm” on page 17.

The server wishes to fully utilize the disk in the present so that future admissions will have a higher likelihood of acceptance. Thus, the server reads ahead for each stream and stores those disk blocks in buffers at the server. If there was infinite buffer capacity, then the entire stream could be read into primary storage as quickly as the disk could read and transferred to the client as needed. Since there are only a fixed number of buffers available, read-ahead is limited by the total buffer space.

When scheduling a stream, it is possible to alter the delivery requirements based on the following factors:

(1) Client buffer space

(2) Server buffer space

(3) Time between CmfsPrepare and calls to CmfsRead

(4) Client presentation bit rate.

The analysis in this section assumes no buffering in the client. A goal of this design, however, is to maximize the use of client buffer space and minimize the corresponding use of server buffer space. This will be explored in the next section.

When total buffer space at the server is exhausted, the rate at which the disk can read is limited by how many buffers are released during each interval. Ultimately, in the long run, the disk can only read as fast as the cumulative presentation rate of the clients.

Consider three 2-dimensional arrays of disk block requests being compared when scheduling a stream: new, committed, and diskmin. Table 1 contains an example of the request for a new stream. Table 2 and Table 3 show sample logical contents of committed and diskmin arrays, respectively. The diskmin array initially contains all maximum values, but changes dynamically if buffer space becomes used up. It returns to the maximum as buffer space is released by presentation at the client. Thus, a counter of available buffers must also be kept to know when the buffer space limits the disk reading. This buffer space is per NODE, so the disks on the NODE must share this buffer capacity.

slot **Disk Id**	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	12	15	10	8	4	15	18	12	10	6	5	8	4	9
1	10	10	10	11	12	14	16	12	9	8	9	9	10	5
2	5	15	12	10	18	6	8	4	9	15	10	11	11	1

Table 2: Current Disk Commitments

Slot **Disk ID**	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	20	20	20	20	20	20	20	20	20	20	16	18	17	20

Table 3: Effective Minimum Guaranteed Disk Bandwidth

Slot Disk ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	20	20	20	20	20	20	20	20	20	20	15	17	18	20
2	16	16	16	16	16	16	16	16	16	16	12	13	15	16

Table 3: Effective Minimum Guaranteed Disk Bandwidth

As scheduling of read-ahead progresses, the system keeps track of how many server buffers are utilized.

When the number of free buffers reaches a value smaller than the minimum number that can be read in a slot, that value is used to compare to the ‘committed+current’ value for admission control as seen in the detailed algorithm. The scheduler also uses the playout rate of the current mix of streams to determine when buffers released by the stream manager (via the network) get added to the pool of free buffers. The client guarantees to be performing CmfsReads within a certain amount of time from the return of CmfsPrepare. Thus, the schedule itself can be used to add back buffers into the pool which are guaranteed to have been presented at the client by the time of the slot being considered (see *noFreedForSlot* in the detailed algorithm). This will serve to increase the amount of data that can be read in an interval back to the level indicated by the disk bandwidth constraints.

To adequately address fairness in buffer usage, each stream should have a running count of how many buffers or the amount of time it has read ahead. This count should be kept within a range (lower bound - upper bound). When the upper bound is reached, the scheduler can place a value of 0 for the read requirements of an interval until a lower bound is reached, at which point the read-ahead can continue. This helps to ensure some fairness in the server buffer allocation among streams. The admission control algorithm must ensure that the read-ahead value can never get below 0, since that implies client starvation.

4.1.3 Read Ahead (Buffering at the Client)

Based on network transmission speeds and client display rate, a time **t1** can be calculated which will indicate when all the buffers at the client will be filled. In the meantime, the server continues to schedule the read-ahead at its pace for the rest of the schedule.

Client buffer space affects scheduling decisions. The client must communicate how much buffer space it has committed for this stream. After the time that the network takes to fill the client buffers, some buffers can be slowly released at the server. The server knows that the client is ahead by that committed amount and can reduce the high-water mark for that client appropriately. This is equivalent to releasing those buffers (in the sense that it does not need to store as many read-ahead buffers at the server). Those buffers must still be transmitted to the client, but the knowledge that the client has the buffer space allows the scheduler to insert 0’s into the schedule for that stream, and dynamically reduce the read-ahead for that particular stream.

Some existing questions remain unresolved. They include issues such as: what is an adequate policy for allocating buffer space at the server among the clients, or should we keep the read-ahead high-water mark the same for every client, regardless of the amount of buffer space existing at the client?

The reason for placing 0's in the data stream is to leave bandwidth and buffers for other streams which have not been admitted yet. Care must be taken so that the stream keeps a certain amount ahead and that the bandwidth requirements in the future are not violated (not too many 0's).

The value of the high-water mark must allow for (or at least consider) the ability to admit new streams in the near future (i.o.w. Don't use up all buffers). There is a trade-off in this because using up more buffers now means that there will be more bandwidth later. This is good, because it allows a stream arriving further into the future that has higher bandwidth requirements to be accepted, but it also may delay admissions in the near future due to a shortage of available buffers.

4.1.4 Read Ahead Algorithm with Buffering at both Client and Server)

Assumptions

1. To consider the effect on buffer utilization, we assume the disk reader consumes at most MaxIO buffers per slot, where MaxIO is the maximum number of IOs per slot for the server. This value is a calibrated constant depending only on disk performance parameters. The server either reads MaxIO blocks or 0 blocks per slot. In reality, fewer blocks than this might be read, due to different locations of the disk blocks, but this provides a worst case for buffer utilization.
2. The disk reader can guarantee MinIO reads per slot. This is the worst case value for the ability to read-ahead. The combinations of MaxIO and MinIO gives a range by which we can bound the amount of read-ahead from below and the buffer utilization from above.
3. The network releases (at least) the scheduled I/Os per slot (call this S). This amount (S) varies per slot according to the playout schedule. We assume that even though more than S buffers may be released, we cannot take advantage of that possibility in allocating buffer space.
4. The buffer pool size diminishes by MaxIO - S buffers per slot.
5. Slot[] is the currently allocated playout vector for the set of active streams. The vector is composed of two pieces: the read ahead portion (at the beginning) which is always 0, plus the unread portion. Assume that this vector extends arbitrarily long (bounded by SLOT_VEC_SIZE).
6. BuffAlloc[] is a vector containing the number of buffers actually allocated to the current read ahead portion (again for the set of active streams) plus the number of buffers expected to be allocated (this later part is equal to unread portion of Slot). As the network transmitter sends the blocks and frees the buffers it decrements the values in the vector.
7. The variable "ShouldBe" points to the slot position in Slot and BuffAlloc which should be currently read to maintain continuous delivery of the data across the network. The disk schedule must be at least this far ahead and typically has read significantly more data into server buffers (at any given point).

Example

0	0	0	0	0	0	8	9	5	7	9	7	8	9	10	9	8	7	5	9

TABLE 4. Slot (i.e., this is the currently allocated schedule

6	9	7	8	9	6

TABLE 5. These are the slots that were read ahead

And here is the BuffAlloc vector. Notice that in the first three slots the network was able to transmit a bit

4	7	6	8	9	6

TABLE 6. BuffAlloc

faster than the normal expected playout rate. As a result we were able to free 5 extra buffers and add them to the total pool. Actually, BuffAlloc extends past these six slots but after that it is identical to the Slot values.

Detailed Algorithm

```
typedef enum{ Accept, Abort } Result;

Result AdmissionsTest( stream, slotCount )
{
/*    stream[] is the playout vector for the new stream */
/*    slotCount is the size of the vector in number of time slots */


/*    F = no of free buffers in the Server, a copy of a global variable */
      totalReadAhead = 0;

      for( i = 0; i < slotCount; ++i ) {
            Slot[ShouldBe+i]  += stream[i];
            BuffAlloc[ShouldBe+i]+= stream[i];
      }

      /* We assume that the slot count will not cause wrap-around */
      /* in the allocation vectors */
      for( i = ShouldBe; i < SLOT_VEC_SIZE; ++i ) {

            ioCountForSlot    = Slot[i];
            noFreedForSlot    = BuffAlloc[i - 1];

            /* This read-ahead could be negative in MANY cases */
            readAhead = MinIO - ioCountForSlot;

            if( F - MaxIO >= 0 ) {
                  F = F - MaxIO;/* read MaxIO blocks * /
                  totalReadAhead += readAhead;
            }
            else {
            /* We have run out of buffers, so the server stops
            reading. However, we can continue to accept
            this stream as long as totalReadAhead is not
```

```
            negative. Since we do not read ahead, nor
            read at all, we actually fall behind */
                  totalReadAhead -= ioCountForSlot
            }

            F = F + noFreedForSlot;/* transmit slot blocks */

            /* Correct the schedule because of the rejection */
            if ( totalReadAhead < 0 ){
                  /* De-allocate the number of buffers needed */
                  for( i = 0; i < slotCount; ++i ) {
                        Slot[ShouldBe+i] -= stream[i];
                        BuffAlloc[ShouldBe+i]-= stream[i];
                  }
                  return Abort;
      }
      return Accept;
}
```

Notes

1. It is necessary to check the allocations in slots past the end of the new stream. The reason for this is that the new stream will alter the buffer requirements of the entire node allocation. Hence, a stream which was previously accepted may not have the resources available for presentation now even though the critical point (in terms of either bandwidth or buffers) in that stream is past the end of the new stream.
2. The algorithm assumes that each stream will only send to a client at its play out rate (i.e., buffers will be freed based on the schedule). This is a reasonable assumption since the client can not consume the data faster than the play out rate (averaged over the entire stream).
3. The algorithm takes into account a stream which has read faster than its play out rate via BuffAlloc. This is also reflected in the value of F.
4. This algorithm is based on the notion that the file server can not read faster (ahead) than it has buffers available.
5. We should still have dynamic rescheduling of streams if one stream is hogging too many buffers.
6. This algorithm only deals with one disk per stream. 2 dimensional allocations would be needed to deal with a stream that is striped across several disks.

4.2 Node Manager

This process controls the connections for clients which are presenting data from streams stored on this machine.

Basic Logic:

```
1) Create Disk Manager Thread(s) for this node
2) Register with to Administrator

3) Repeat forever
```

```
4) receive message
5) switch on message type

      case CMFS_OPEN_CONNECTION:
            call CmfsOpen and save administrator worker PID
            retrieve schedule and frame sizes for entire object in Connection
            Control Block
      case OPEN_COMPLETED:
            reply to the XTP thread
            Reply to administrator worker
      case CMFS_CREATE_REQ:
            call CmfsCreate and reply to administrator worker
      case CMFS_CLOSE_CONNECTION:
            if connection state is prepared then call CmfsStop
            call CmfsClose (deschedule disk reads and kill XTP stream thread)
            Reply to the client PID
      case CMFS_PREPARE_STREAM:
            call CmfsPrepare which enqueues request on Admissions Queue
      case PREPARE_SCHED_COMPLETED:
            reply to the disk manager and to the XTP stream thread
      case PREPARE_SEND_COMPLETED:
            reply to client and to the XTP stream thread
      case CMFS_STOP_STREAM:
            call CmfsStop (deschedule and flush buffers)
            reply to client thread
      case OTHERS (not yet implemented: CmfsGetPosition, CmfsTime2Pos,
CmfsPos2Time, CmfsCreateCompositeObject?, CmfsRemove)
end switch
```

4.3 Disk Interface

Initially, the disk device will be a virtual disk. The first prototype implementation used 1 UNIX file per sequence. The current prototype implementation uses 1 UNIX file to simulate an entire disk. The low-level disk interface accesses the objects to be stored on that disk via block addresses. Later, this will be modified to use the Raw Disk Interface for better performance. We will split up each physical disk into some numbers of virtual disks. The following calls provide a programming shield for the CMFS from the physical implementation of the disk access.

4.3.1 CmfsFormat

This program formats a virtual disk to contain a requested number of blocks. When a call to DSMInit fails, the server will acquire the disk space.

4.3.2 DSMInit

This procedure ensures that the file space has been previously allocated for the system.

4.3.3 DSMalloc

This procedure reserves a series of contiguous blocks of disk storage. The caller requests a specific number of blocks and the actual number of blocks reserved is returned.

4.3.4 DSMfree

Previously allocated blocks are returned to the system. This call is used in conjunction with CmfsRemove.

4.3.5 DSMreq

The actual disk read and write commands can be grouped into a list of disk activities and be performed as a group by this call. Control will return as soon as the requests have been initiated. DSMreq operates on an array of DSMcmds, which are defined as follows:

```
typedef struct DSMcmd
{
      u_int       reqType;
      u_long      blockNo;
      u_long      count;
      DSMbuf      *buffer;
      u_int       status;
      u_int       reason;
      u_long      userData;
} DSMcmd;
```

4.3.6 DSMwait

This procedure waits until one or more of the DSMcmds specified have completed. Thus, DSMreq and DSMwait can increase the parallelism inherent in the disk device, by scheduling all the reads for a particular slot to occur in the order that the lowest level device driver deems appropriate. SCSI devices will do this in any regard, so this interface is designed to take advantage of this mechanism.

4.4 Disk Manager Thread

The disk manager thread performs admission control and reads data for the streams. It has the following logic:

```
Repeat forever
      Wait on semaphore which indicates that there is work to do
      Admit any streams that are on admissions queue (if appropriate)
      Repeat while number of active streams > 0
            For time slot i
                  read the blocks for time slot i
                  wait until reads complete
                  queue blocks for appropriate XTP stream managers
                  if a read returns end of file decrement # of active streams
```

```
                  Admit any new streams from admission queue (if any)
            end for
      end repeat
end repeat
```

The connection control blocks are available for the Disk Manager thread to keep track of the open XTP connections to determine which queue the block should be placed on. Each disk block that is put on a queue must have some method of identifying it with a particular XTP connection. This is accomplished by the userData field in a DSMbuf.

4.5 XTP Manager Thread

```
Basic Logic:
      Make XTP connection and set up the connection control block, saving the
            initial buffer request size
      send message to Node Manager indicating result of opening connection.
            Control will return to this thread once the client has performed prepare.
      If the connection was unsuccessful, then exit
      Send over initial buffer and send message back to Node Manager
      Repeat forever
            dequeue block for stream
            if block indicates NO_MORE_DATA then set the appropriate XTP bit
      transmit on appropriate XTP connection (XtpWrite)
      end repeat
```

5.0 Implementation Status and Work To Do

Two prototype versions of the CMFS have been implemented which have different mechanisms for the disk transfer and scheduling. The network transmission and administrator node logic has remained essentially the same.

The initial prototype was implemented using separate threads for administering the schedule of active streams and managing the disk. This functionality has been combined into the disk manager in the current version. As well, the previous version did not implement the admission control algorithm, nor use the detailed disk block schedule to organize the reads from the disk and only transmitted complete objects from start to end.

The second prototype system is being tested which uses modifications of the design that are not described in this document. A stream presentation cannot pause or be played at alternate speeds.

Buffer management is an issue that has yet to be addressed in the current implementation and the calibration of the disk has not been done so arbitrary constants are used for those values.

References

[1] David Finkelstein, Norman Hutchinson, Dwight Makaroff, Roland Mechler, and Gerald Neufeld. The UBC Real Time Threads Package: Interface Manual, UBC Technical Report 95-07, March 1995.

[2] Dwight Makaroff, Gerald Neufeld, and Norman Hutchinson: The UBC Distributed Continuous Media File System: Interface Manual, Internal CMFS Report, March 1995.

[3] Roland Mechler and Gerald Neufeld, XTP Application Programming Interface, UBC Technical Report 95-17, June 1995.

[4] W. T. Strayer, B. J. Dempsey, and A. C. Weaver. XTP: The Xpress Transport Protocol, Addison Wesley Publishing, 1992