Arca: A Local Network File Server

S. MUIR, D. HUTCHISON AND D. SHEPHERD
Department of Computing, University of Lancaster, Bailrigg, Lancaster LA1 4YR

Distributed systems can be built to a client-server model in which one of the important servers is the file server, consisting of at least one processor and disk drive, attached to the network with the sole purpose of storing information on behalf of its clients. In the Department of Computing at Lancaster University we have designed and built a file server which is connected to both of our networks: an Ethernet-type network (called Strathnet, of local design) and a Cambridge Ring type network. The Arca File Server is based on work started at Cambridge University. This paper describes the background of file servers for local networks and the design and implementation of our system.

1. INTRODUCTION

One of the fastest growing subjects for research in computing today is that of local area networks (LANs) and their applications. These differ from wide area networks in that they are of a limited size, commonly a maximum span of 1km, and they have both a high data transmission rate and an inherently low error rate. They may be connected to other LANs or to wide area networks by gateways, which are machines devoted to passing messages between networks.

There are two main types of commercial LAN in existence: Ethernets[1] and Cambridge Rings[2,3]. The original experimental Ethernet[4] is still in active use at Xerox PARC[5]. The Cambridge Ring is named after Cambridge University where it was developed and where there is now a fully operational distributed system connecting many machines together[6].

The advent of LANs has given rise to an increased interest in loosely-coupled distributed systems. In the past, each computer had its own printer, disks, terminals and so on. Terminals were connected to the computer by direct cables, which meant that users were literally hardwired into one particular computer. Neighbouring computers normally had no connection between them and consequently no way of transferring files from one to another. In a LAN-based system, multi-user computers can be connected together using the LAN to provide file transfer capabilities. User terminals can be clustered through terminal concentrators attached to the LAN so that each terminal can potentially access any computer. Furthermore, the resources hitherto centralised on one machine can now be spread over a number of machines interconnected by the LAN. Some of these machines may provide a service of some specific kind (servers), and others use these services (clients). Examples of servers are printer servers (which spool files and print them), name servers (for converting the name of a service to its address on the network), compiler servers (for compiling programs written in a particular language) and file servers (to hold files for clients in non-volatile storage). There may also be several processor servers, which accommodate the user workload of the system between them, and terminal concentrators as previously mentioned.

At Lancaster University we are running two different networks in parallel[7]: a Cambridge Ring, and an Ethernet-type network called Strathnet. The Arca file server was designed to provide a centralised file store for any machine on either network. It is based on the Cambridge File Server (CFS)[8] but, unlike that system, it is not intended to be used for swapping programs in or out of a host's main store; therefore, it does not implement the CFS fast read/write mechanism.

The Arca file server allows the client to impose any kind of file structure he desires and supports simple commands to manipulate files or directories. Unlike the Cambridge File Server, which runs on an operating system kernel called TRIPOS, our implementation is written as a monolithic program running on a bare machine.

2. LANCASTER ENVIRONMENT

In our department we have a VAX-11/750, a PDP-11/24, an LSI-11/23, four LSI-11/02's, two M68000 systems and a few microcomputers, as illustrated in figure 1. Both Strathnet and the Ring have a similar set of hosts attached and, in some cases, hosts have interfaces to both networks. Strathnet was entirely designed and built by two of the authors[9]. Each device is interfaced to Strathnet by means of a microprocessor-based network access unit (AU). The VAX-11/750 is used for research and the PDP-11/24 is for undergraduate work. The LSI-11/23 is for the file server itself and has a Winchester disc attached. Three of the LSI-11/02's are used as stand-alone work stations, the other is used as a print server and the two M68000 machines are for gateway development work. The file server can thus be used to store files for these machines and any others which may be added at a later date.

The Cambridge Ring, called Polynet, was supplied by Logica VTS Ltd.[10], but the interfaces to the 6809 and 68000 machines were designed and built by us. The file server is connected to, and can communicate with, both networks at once. Requests can, therefore, be sent to it on either network and the reply will be returned on the same network from which the request came. The file server's Strathnet interface is connected to a DRV11-B, a general-purpose DMA interface. The original intention was that, once a block had been transferred into the node from Strathnet, it would be transferred to the 11/23 using burst mode DMA. This implementation is not complete but meantime we are using program-controlled single cycle DMA (controlled by the microprocessor in the Strathnet AU).

There are two performance monitors, one for each network. These are being used in a programme of comparison experiments between Cambridge Ring and

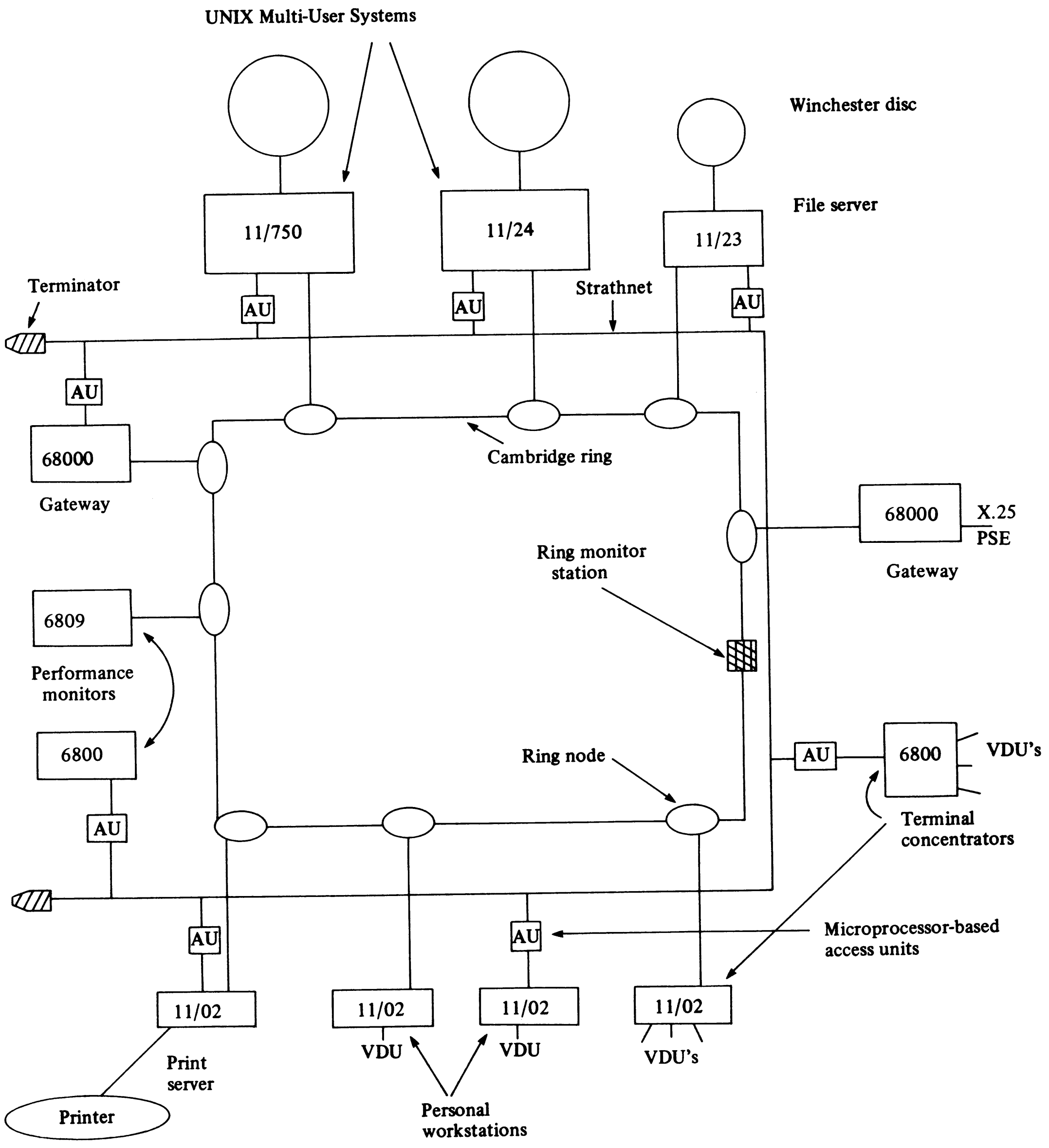

Fig 1. Dual network configuration

Ethernet-type local networks[7], to capture network traffic and produce statistical performance information.

3. PREVIOUS WORK ON FILE SERVERS

A review of the current literature showed that the only two places with file servers that had been running successfully for any length of time were Xerox PARC in the USA and the Cambridge University Computer Laboratory. At Xerox PARC there were two file servers in existence, WFS[11] and XDFS[12], and at Cambridge there was CFS[8]. Other work on file servers was in progress, particularly at MIT on the SWALLOW[13] system, but none of the work was sufficiently advanced to form a basis for our file server implementation. We, therefore, decided to examine WFS, XDFS and CFS in some detail.

We were soon able to eliminate WFS from our considerations. It is a simple system which does not support atomic transactions on files, a feature we considered essential in a distributed environment. Furthermore, we felt that its structure does not lend itself to further development such as the addition of garbage collection and distribution across more than one machine.

XDFS and CFS were found to contain most of the facilities we required (see[14] for a good comparison of these two file servers). For instance, they both support concurrent random access to files and an atomic transaction mechanism covering modification of files. However, they had been designed with different goals: the XDFS was intended to support database research whereas an important constraint on CFS was that it should be suitable for use as the CAP computer's backing store[15]. Thus XDFS and CFS have slightly different characteristics. CFS is biased towards the requirements of operating systems: rapid access to files and high transfer rates. It has a minimal set of file operations and concentrates its full bandwidth on one particular client for the duration of one read or write. The XDFS, by contrast, can cover updates to a number of files so that the atomicity of a higher level database operation can be maintained. It provides a directory for file names, and access control.

There is also a considerable difference in the size of machine required to run the two systems. XDFS runs on an Alto Computer using 164K 16-bit words, 64K of which is data and 100K code. CFS runs on a Computer Automation LSI 4/30 with 64K 16-bit words of memory, 50K words of code and 14K for data and disk buffering.

We decided that the CFS approach was more suitable for our needs for two reasons:

(1) We required a simple file server with high transaction speeds to support distributed operating systems research.
(2) Our system had to fit into a 32K 16-bit word machine and we felt that a system along the lines of CFS could fit into the machine.

4. THE ARCA FILE SERVER

The Arca file server is a stand alone system which provides a file store to be shared by other systems, henceforth called clients, on the network. It allows clients to create, destroy, read and write files from a distance. It is important when designing a file server to strike the correct balance between what the file server is expected to do for the clients and what they must do themselves. In the case of the Arca file server we had three main aims:

(1) To do as much for the client as possible without either imposing any kind of restriction on its clients regarding file structure or taking on time consuming operations that will hold up other clients.
(2) To allow each client to impose his structure on files and to let several different file structures co-exist on the one filing system.
(3) The file server should not lose data when a crash occurs and should be able to recover automatically.

In order to satisfy the third requirement the file server has to ensure that certain types of file will be updated atomically, that is, if either the hardware or software fails during a transaction, the file server will return the file to either its final or original state depending on whether the file was closed or not[16] (see section 7). Because of the overheads involved in the atomic update sequence, it is adopted for a file only if the client explicitly requests it when the file is created. This type of file is called 'special'. Although the client could be allowed to change the 'special' attribute of an object, we decided it was not a useful thing to do because objects which are easily re-creatable (ie compiler listings) will always be re-creatable and vice-versa for non re-creatable objects (ie program sources).

As indicated in section 3 we decided to base our file server on CFS, which has the following desirable properties:

- high speed transfers to random access word-addressed files.
- the ability to perform atomic updates to files.
- a capability-like access control mechanism.
- automatic reclamation of unused storage.
- attention to the integrity of stored data.

In the next section the underlying file structure of our implementation, based strongly on that of CFS, is explained in detail.

5. FILE SYSTEM STRUCTURE

Each object is uniquely identified by a PUID (Permanent Unique IDentifier) which is created with the object and is never re-used or destroyed. A UID has the format shown in figure 2.

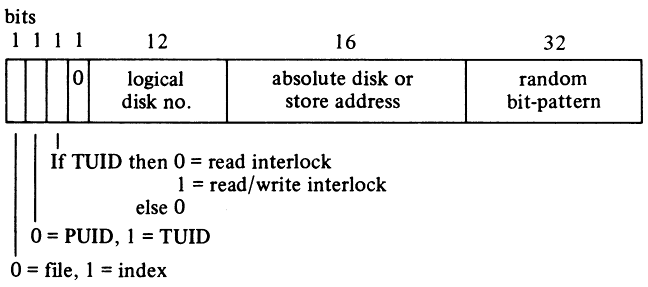

Fig 2. Format of a uid

When an object is opened, (ie disk held information about the file is entered into a table), the file server returns a TUID (Temporary Unique IDentifier) synonymous with its PUID. This TUID is valid only until the object is closed and is never re-used.

The random bit pattern is the only kind of protection the file server uses. It makes use of the fact that 32 random bits will be very difficult to guess and will take too long to try out all combinations.

Two types of object exist on the File Server: files and indices. A *file* is an array of bytes, whereas an *index* is an array of PUIDs, each representing a file or index. Some (or all) entries in an index may be the null PUID (all zeroes). Note that the client is not allowed to write explicitly to an index. Any index may contain a PUID for any object provided only that the object is in existence. Thus, the structure is that of an undirected graph (figure 3). One index, however, is distinguished from the rest – that of the *root-index*. This index is important because any object not reachable by any chain of indices starting from the root is eligible for deletion. No further structure is imposed on the file store other than that mentioned above. Thus, although creation of objects is controlled by the clients, their deletion is controlled by the file server.

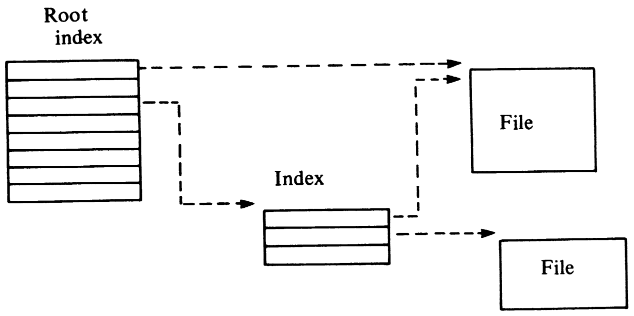

Fig 3. File store graph structure

So far we have described the client's view of the filing system. Its internal structure is now explained. Each object (file or index) has certain attributes, namely whether it is a file or index, special or non-special (indices are always special), its PUID, its 'uninitialised' value and its logical size. The 'uninitialised' value is the value returned for any byte which has never been written, and is always zero for indices. The logical size determines *only* the highest word address which may be written and has little bearing on the actual physical size. Thus, increasing an object's logical size will not change its physical size (except if a change in the object's 'depth' is caused – see later), but a reduction of its logical size may cause some physical blocks to be removed from that object. Reduction of the logical size of an index may also cause references to PUIDs in that index to be lost. Each object also has a 'high water mark' indicating the highest address ever written. The logical file size and the high water mark are the only dynamic attributes of an object.

Transparent to the client is the object's 'depth' which can be one, two or three depending on its logical size. The following gives the object's depth, D, in terms of the logical file size (in bytes), L:

$$L \leqslant 504 \Rightarrow D = 1$$
$$504 < L \leqslant 252*512 \Rightarrow D = 2$$
$$252*512 < L \leqslant 252*256*512 \Rightarrow D = 3$$

Note that, for an index, L is always a multiple of eight (because a PUID is eight bytes), but the client sees its logical size only in terms of the number of entries it holds.

There are 512 bytes in each disk block. All objects consist of at least one block, called the 'first block', which contains some of the object's attributes in the first eight bytes. The meaning of the remaining 504 bytes depends on the object's depth. For a depth of one, these bytes contain the data itself; for a depth of two they contain 252 block addresses, each of which contains 512 bytes of data; for a depth of three they give the addresses of indirect blocks, each of which contains 256 block addresses, each of which in turn contains 512 data bytes.

When an object is created, it always occupies one physical block regardless of its logical size; if its depth is greater than one, the remaining 504 bytes of its first block are null pointers. Further blocks become allocated as necessary; until then, they are read as the appropriate number of 'uninitialised' bytes.

The last block of each cylinder of the disk is used by the file server as a 'cylinder map'. This map contains four words of status for each block on that cylinder. The information contained in the cylinder maps and the tree structure of each object are mutually redundant: thus, each may be rebuilt from the other. This is the basis of the file server's ability to perform atomic updates (see section 7). If the cylinder map becomes bad, a new disk will need to be obtained. The information held in the cylinder map for each block is as follows:

word 1:

1. allocation state (1 = allocated, 0 = de-allocated) – 1 bit
2. intention state (1 ⇒ intending to change allocation state) – 1 bit
3. first (1 ⇒ first block of an object) – 1 bit
4. index (1 = index, 0 = file) – 1 bit
5. commit (1 = commit, 0 = don't commit) – 1 bit
6. level1 (1 ⇒ this block contains pointers to data blocks) – 1 bit
7. level2 (1 ⇒ this block contains pointers to level1 blocks) – 1 bit
8. (9 bits unused)

word 2:

sequence number (0..255) of block within its parent (the block that contains its address) or zero if none

words 3 and 4:

if this is the first block of an object, these words contain the random part of its PUID, otherwise they contain the address of its parent block and the address of the object's first block.

Finding a free block to extend an existing object is performed by looking first on the cylinder containing its 'first block', then examining cylinders in both directions from this point. It is hoped that this algorithm will minimise the head movement when accessing an object.

6. COMMUNICATION WITH THE FILE SERVER

As stated in the introduction, the file server is connected to two different types of network. The file server access protocols used on the networks are built above the CSMA/CD (carrier sense multiple-access with collision detection) level for Strathnet and mini-packet protocols for the Ring. At the network control level, the Basic Block protocol[17] is used on both networks so that higher protocol levels need not know which network is currently being used. Two access protocols are used by the file server: the Single-Shot Protocol[18] and the Remote Procedure Call mechanism[19].

With the Single-Shot Protocol (SSP), a request is sent (function + parameters), then the reply is received (status + results) when the operation is complete. Examples of functions are: CREATE_FILE, DELETE, OPEN, CLOSE, READ and WRITE. A complete list of the Arca file server functions is given in the Appendix to this paper. The format of an SSP request or reply is as follows:

1. Basic Block header – 16 bits
2. Basic Block destination port (0 is SSP request port) – 16 bits
3. SSP request tag – 16 bits
4. reply port (only meaningful if request) – 16 bits
5. function (request) or status (reply) – 16 bits
6. tag – 16 bits
7. parameters (request) or results (reply) – array of 16-bit quantities
8. Basic Block checksum – 16 bits

A port is a known address, represented by a 16-bit integer, to which requests and replies can be sent. Note that the two 'tag' words are ignored by the file server, but are copied from the request block to the reply block. They may be used, for example, to distinguish between different replies to the client.

The Remote Procedure Call (RPC) is similar, but guarantees successful communication of at most one request-reply pair for each one sent, even in the event of crashes of the host or file server. It relies on the file server storing a history of the most recent requests and replies in stable storage[20] and, if a request matches one of the stored requests, the file server will *only* send the same reply again without taking any other action.

The format of an RPC request or reply is as follows:

1. Basic Block header – 16 bits
2. Basic Block destination port (1 is RPC request port) – 16 bits
3. identifier – 16 bits
4. node number – 16 bits
5. time – 32 bits
6. serial number – 16 bits
7. node number for reply (request only) – 8 bits
8. port number for reply (request only) – 8 bits
9. function (request) or status (reply) – 16 bits
10. number of parameters (request) or results (reply) – 16 bits
11. parameters (request) or results (reply) – array of 16 bit quantities
12. Basic Block checksum – 16 bits

Note that the fifth and sixth fields together must form a monotonically increasing function to enable any server to distinguish between old and new requests. The 'time' field is the time as seen by each client and the server remembers the last value of each of these. The 'serial number' field is used to distinguish between requests sent in the same time slot and is incremented (by one) for each request. It is assumed that it is not possible for this serial number to return to its starting point before the client's clock increments.

7. ATOMIC UPDATES

The File Server has a mechanism by which an object can be updated atomically in that, if an update fails, the object will be restored to either its original or final state. A file may be given the property 'special' at create time which causes all updates to the file to be performed atomically, otherwise this mechanism will be by-passed. The latter case is useful when a file can be easily re-created and one wishes to avoid the overheads of atomic updates. Note that indices are *always* treated as 'special'.

As stated in the previous section, the information pertaining to the consistency of the file store (excluding the contents of data blocks) is recorded twice: once in the graph structure of the indices and the tree structure of each object, and once in the cylinder maps (so called because there is one per cylinder).

An atomic update to a 'special' object of depth two is as follows:

When an object is opened for writing, a copy is made of its first block and the new block is marked as a 'new first block'. As the object is written, intending to allocate/de-allocate block pairs are generated. These intending to allocate/de-allocate states are used by the automatic crash recovery program to enable it to ascertain which blocks should be freed and which blocks should become part of the object. When the object is closed, the following steps occur:

1. Set the 'commit' bit associated with the object.
2. Copy the 'new first block' into the 'old first block' and de-allocate the 'new first block'.
3. Change all 'intending to allocate' blocks to 'allocated'. Change all 'intending to de-allocate' blocks to 'de-allocated'.
4. Reset the object's commit bit.

If the file server crashes and restarts, the 'restore' program will examine the disk and will either undo or complete any unfinished atomic transactions. If any cylinder map is found to be unreadable, all cylinder maps will be rebuilt by traversing the undirected graph from the root-index. Otherwise, the commit bit of each object is examined and the allocation state of each block involved is changed as follows:

commit = 0:
  intending to allocate → de-allocated
  intending to de-allocate → allocated

commit = 1:
  intending to allocate → allocated
  intending to de-allocate → de-allocated

Note that the 'restore' program undoes/completes atomic transactions in the same way as the file server so that a crash in this program is also recoverable. Note also that this method assumes that a block which was half written will be left detectably bad.

Having set the commit bit of the object, no further cylinder maps are written until the 'new first block' has been copied onto the 'old first block'. Thus, if the cylinder maps need to be rebuilt, the view of the object from its first block will be correct according to what the commit bit would have been. Therefore, the only ways the file store could be damaged are due to software errors or physical damage to the disk.

8. GARBAGE COLLECTION

Garbage collection is performed asynchronously with the normal operation of the File Server. The garbage collection algorithm runs at priority zero and is

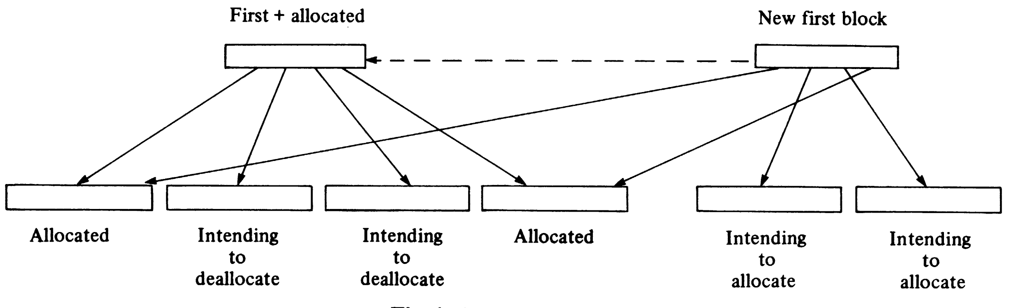

Fig 4. Atomic update

interrupted by the receipt of any network request. This algorithm is as follows:

1. Mark all objects as 'not found'.
2. Search graph marking objects seen as 'found'.
3. For each object marked as 'not found' do the following:
   (a) mark its first block as 'intending to delete'
   (b) mark the rest of its blocks as 'intending to delete'
   (c) mark its first block as 'deleting'
   (d) free all its blocks in depth-wise order.
4. Reset the 'intending to delete' bits of every block.

During step 2, a note is made of any first block marked as 'deleting'; if so, the process of deleting that object will be continued immediately before step 3 begins. Thus, there will only be one partially deleted object at any time.

No object will be 'retained' in an index if its first block has the 'deleting' bit set. Whenever any object is 'retained' in an index, the 'found' bit is set for that object. This is done to prevent the garbage collector from deleting it. When an index is retained, the garbage-collector is restarted.

9. CONCLUDING REMARKS AND FURTHER WORK

At present the Arca File Server is being used to support comparison experiments between Cambridge Ring and Ethernet-type (ie Strathnet) local networks.

The Arca performance has been investigated along lines previously adopted for other file servers (see XDFS and CFS details in[14]), and has been found to be comparable given the normalisation of the following factors: disc average access times, disc file block sizes and communications overheads.

Using the Remote Procedure Call interface described earlier (section 6), it is our intention to integrate the file server into a network operating system being developed in the department[21].

We are also considering using stable storage instead of cylinder maps so that there will be no need, in the event of crashes, to rebuild these maps by traversing the undirected graph. Atomic updates will thereby be speeded up and the need for intending to allocate/de-allocate block pairs may, in addition, be circumvented.

One final area of interest is the idea of distributing the file server over two or more machines. This will allow for maintenance and expandability, and will improve file server reliability.

Acknowledgements

The work reported in this paper is being supported by the UK Science and Engineering Research Council. Thanks are due to Jeremy Dion at Cambridge University Computer Laboratory for his kind help throughout the early stages of this project.

REFERENCES

1. Ethernet: a local area network – data link layer and physical layer specifications. Digital, Intel and Xerox Corporations, version 1.0 (September 1980)
2. W.P. Sharpe and A.R. Cash, Cambridge Ring 82 – interface specifications. UK Science and Engineering Research Council (September 1982)
3. J.Larmouth, Cambridge Ring 82 – protocol specifications. UK Science and Engineering Research Council (November 1982)
4. R.M. Metcalfe and D. R. Boggs, Ethernet: distributed packet switching for local computer networks. *CACM* **19** No. 7, 395 – 404 (July 1976)
5. A.D. Birrell, R. Levin, R.M. Needham and M.D. Schroeder, Grapevine: An exercise in distributed computing. *CACM* **25** No. 4, 260 – 274 (April 1982)
6. R.M. Needham and A.J. Herbert, The Cambridge Distributed Computing System. Addison-Wesley (1982)
7. D.Hutchison and W D. Shepherd, A Direct Comparison of Ring and Ethernet-like Local Networks. Internal Report, Dept. of Computer Science, University of Strathclyde, Glasgow (1981)
8. J. Dion, The Cambridge file server. *ACM Operating Systems review* **14,** 26 – 36 (1980)
9. D. Hutchison and W.D. Shepherd, Strathnet – a local area network. *Software and Microsystems* **1** No. 1, 21 – 27 (October 1981)
10. Polynet product description. Logica VTS Ltd. (April 1981)
11. D. Swinehart, G. McDaniel and D. Boggs, WFS – A simple shared file system for a distributed environment. *Proc. of the Seventh Symposium on Operating Systems Principles* Asilomar, California, 9 – 17 (December 1979)
12. H.E. Sturgis, J.G. Mitchell and J. Israel, Issues in the design and use of a distributed file system. *ACM Operating Systems Review* **14** No. 3, 55 – 69 (July 1980)
13. D.P. Reed and L. Svobodova, SWALLOW: A distributed data storage system for a local network. *Proc. of the International Workshop on local networks* Zurich, Switzerland (August 1980)
14. J.G. Mitchell and J. Dion, A Comparison of Two Network-based File Servers. *CACM* **25** No. 4, 233 – 245 (April 1982)
15. C.N.R. Dellar, Removing backing store administration from the CAP operating system. *ACM Operating Systems Review* **4**, 41 – 49 (October 1980)
16. B.W. Lampson, Atomic Transactions. *Distributed Systems – Architecture and Implementation* Lecture Notes in Computer Science No. 105, Springer-Verlag, 246 – 264 (1981)
17. R.D.H. Walker, Basic ring transport protocol. Internal report, Computer Laboratory, University of Cambridge (1978)
18. N.J. Ody, A protocol for 'single-shot' ring transactions. Systems Research Group, Computer Laboratory, Cambridge (April 1979)
19. G.S. Blair, J A. Mariani and W.D. Shepherd, A Practical Extension to UNIX for Interprocess Communication. *Software – Practice and Experience* **13**, 45 – 58 (1983)
20. R.M. Needham, A.J. Herbert and J.G. Mitchell, How to connect stable memory to a computer. *ACM Operating Systems Review* **17** No. 1, 16 (January 1983)
21. G.S. Blair, D. Hutchison and W.D. Shepherd, MIMAS – a network operating system for Strathnet. *Proc. 3rd International Conference on Distributed computing systems* Fort Lauderdale, Florida, 212 – 217 (October 1982)

Appendix

A list of the functions provided by Arca

CREATE_INDEX (existing index PUID, offset in this index, size of new index)
returns: PUID of new index

RETRIEVE (existing index PUID, offset in this index)
returns: PUID

RETAIN (PUID of retaining index, PUID to be retained)

DELETE (PUID of retaining index, offset to be deleted)

READ_INDEX_SIZE (PUID of index)
returns: high water mark, maximum allowed size

CHANGE_INDEX_SIZE (PUID of index, new size)

CREATE_FILE (existing index PUID, offset in this index, size of new file, 'uninitialised' value, 'special' flag)
returns: PUID of new file

SSP_READ (UID, start address, number of bytes)
returns: number of bytes, data bytes

SSP_WRITE (UID of file, start address, number of bytes, data bytes)
returns: number of bytes

READ_FILE_SIZE (PUID of file)
returns: high water mark, maximum allowed size

CHANGE_FILE_SIZE (PUID of file, new size)

OPEN (PUID of file, 'writing allowed' flag)
returns: TUID

ENSURE (TUID, 'update' flag)

CLOSE (TUID, 'update' flag)

EXPLAIN (return code)
return: length of string, string explaining return code in English

ZERO_INDEX (PUID of index)

ZERO_FILE (PUID of file)

COMMUNICATIONS_TEST ( )