The TRIPOS Filing Machine, a front end to a File Server

M. F. Richardson
R. M. Needham

Computer Laboratory, Cambridge University, England

Introduction

This paper discusses an experiment which sets out to improve the performance of a number of single user computers which rely on a general purpose file server for their filing systems. The background is described in detail in reference [1], but for completeness it is necessary to say something about it here.

The Cambridge Distributed Computing System consists, at the time of writing, of between 50 and 60 machines of various types, connected by a digital communications ring. On the ring, there are two file servers [2], [3], which are general purpose (or "universal" [4]) in the sense that they have no commitment to a particular directory or access control structure. This is done in order that they may support several client systems, and so that new systems may be added without difficulty. We speak of a particular directory and access control structure implemented over the file server as "a filing system".

One of the filing systems supported is that used by the TRIPOS operating system [5], [6], which was originally developed for use with single user minicomputers. At first, this system used local discs directly connected to the machine. Subsequently, TRIPOS was used as an operating system for a number of computers which had no local discs at all, and constituted, in the local terminology, the Processor Bank. These machines are allocated to users as and when they require them, and are accessed through terminals which are themselves connected to the ring. When a processor bank machine is allocated to a user it is as much his as if it were a personal machine in his office.

The initial means of providing filing facilities to the processor bank machines was to arrange that, in each of them, there was a copy of the code necessary to implement the TRIPOS filing system as an abstraction on the mechanism provided by the file server. All instances of TRIPOS shared the same filing system. Since the TRIPOS filing system as a whole is an essentially hierarchical structure originating at a single root directory, and leading to users' directories and files, is was necessary to use the file server's interlocking facilities to ensure orderly access. This arrangement worked fairly well, but had certain drawbacks.

Perhaps the main drawback was that, in order to avoid the need for excessive buffering in the client machine, the data transfer requests made upon the file server were usually for rather small amounts of data, typically not exceeding 512 bytes. While the file server is well capable of handling such requests, it performs more efficiently if somewhat larger requests can be made, in the order of 2K bytes and upwards. A second drawback was that a large amount of file server traffic was generated by transferring to the client machine copies of the considerable number of directories involved in interpreting full path names. Since there is no means whereby the occurrence of changes to a file server object can be ascertained, other than by reading that object, directories could not be cached in the client machine. Because of the large number of piecemeal transactions required, the file server became a severe bottleneck when several machines were running TRIPOS, and the response to users became unsatisfactorarily slow. Finally, the TRIPOS filing system code, and other fixed material, took up a significant amount of space in the memory of the processor bank machines.

The TRIPOS Filing Machine attempts to remove some of these drawbacks. Its general function may be described quite briefly. It is itself one of the processor bank computers, but contains the only full copy of the TRIPOS filing system code. The TRIPOS system which is run in processor bank machines allocated to users has the filing system code

replaced by stubs which make remote procedure calls [7] on the filing machine. The filing machine acts upon these requests, and makes use of the file server as necessary. It is able to cache file server material, to read ahead material which appears to be being accessed in a sequential manner, and to maintain a cached abstract of relevant parts of the directory structure to permit rapid interpretation of path names. It is able to manage interlocks in a manner specifically suited to the TRIPOS filing system, rather than by making use of the general facilities provided by the file server. It can also provide protection and accounting facilities which could not otherwise be implemented securely.

It is our belief that the TRIPOS filing machine is very successful in the reliable execution of its tasks, and in improving the performance experienced by the user. It currently supports up to around fifteen concurrent users (the maximum number of processor bank machines available to run TRIPOS). The remainder of this paper discusses the design considerations of the filing machine itself, and the performance measurements which back up the good impression of its working.

General Design Considerations and Possibilities

The movement of the administration of the TRIPOS filing system from the client machines to a centralised server gives rise to two possible developments which are not possible in the (unprotected) client machines. First, the availability of the entire memory of the Filing Machine enables large amounts of data and other information to be cached in memory for quick access. Second, the client and filing machines can be considered as separate protection domains, allowing the provision of protection and accounting schemes which could not otherwise be enforced. Most effort in the development of the filing machine has been directed towards the caching; recently, developments in the latter direction have been made.

Since the TRIPOS file structure is held in data files which are stored on the file server, the caching which the filing machine provides need be no more complicated than imaging parts of these files. In practice, this is the major concern of the filing machine. About 750K bytes of memory are available for the cache, and are divided between those file server files which represent TRIPOS files, and those which represent TRIPOS directories. Data is cached in fixed size blocks, the sizes of which are chosen to be a reasonable compromise between efficiency of data transfer to and from the file server, and effective memory utilisation. Advantage is also taken of knowledge of the way in which the file server stores data on disc, so that transfers can be aligned with file server disc blocks.

The availability of cache space makes practicable substantial amounts of advance reading of data in anticipation of requests from client machines. This has proved generally very effective. If material has been read in advance and turns out not to be required, it can always be discarded. In the case of write requests from the client, a policy decision is needed as to when the material should actually be written to the file server. It will be evident that immediate write-through, synchronous with the client's request, would be highly inefficient for the client, and necessarily worse in performance than the direct writing to the file server that occurred in the system as it was before the filing machine came into use. The choice made was to reply to the client's write request as soon as the material is known to be in the filing machine's cache, and to transfer it to the file server asynchronously and in economically reasonable pieces. This approach gives rapid response to the client and minimises protocol overhead. It is evidently necessary to give the client some comprehensible guarantee about the fate of his material. This is done by synchronising at the time the file is closed: the filing machine does not respond to the client's close call until all buffered material has been written to and acknowledged by the file server. This approach is practical and sensible for two reasons. First, the filing machine does not fail very often. Second, TRIPOS is a system in which the notion of a file is important and in which failure to write a file is likely to be followed by regeneration of the (entire) file. It would not be a practical approach in some other types of system, in particular where the file server was supporting a database rather than a conventional filing system. By taking advantage of the known properties of TRIPOS, synchronisation overheads can safely be very much reduced.

It was mentioned earlier that the operation of the TRIPOS filing system involves a substantial amount of directory searching, both in the course of interpreting long path names and in accessing some of the more commonly used system directories. The effective amount of cached information is increased by holding an abstract of the directory structure, which occupies less space than would be needed to retain the same information in the data cache. This is essentially a map of directories and their entries.

The current implementation of the filing system stub in the client machines provides a small amount of local buffering. It was hoped that this would not be needed, but it was found that without it, the inherent delays in any request to the filing machine caused some degradation of performance. If it is desired to minimise the complexity and memory requirements of the client stub, then this local cache can be removed, without impairing the functionality of the system.

The presence of a protection boundary, in the form of the ring, between the client and the filing machine makes it possible to have an effective protection scheme in a way that was not possible with the earlier ring-based TRIPOS system in which each machine contained full file-system code. The method adopted requires calls on the filing machine to be validated, where appropriate, by an accompanying token which is drawn (originally by the filing machine) from a sufficiently large sparsely occupied space that the token is effectively unforgeable and may be used as a capability (it is assumed that the communication network is secure so that third parties cannot illegaly obtain the tokens). The validity of these tokens can be rigorously checked by the filing machine, which is thus able to control the client's access to objects. Specific, restricted access rights can be associated with any token; this is not possible with the file server, since the possession of its equivalent token implies the right to read or write to that object.

The protection mechanism is based on that used by the CAP computer [8], and uses capabilities rather than access control lists. In practice, users have full access to their own private files, but only restricted access to other objects. By default, all files and directories may be read by any user of the system, but this can be restricted if required.

The protection of the filing machine from the client also permits a simple accounting scheme to be implemented. In this, an accounting "demon" is started running in a processor bank machine once a day (usually during the night when the load on the system is smallest). This demon scans the entire filing system, totalling the space used by each user, and adjusting their file space accounts. The filing machine inspects this account before allowing a user to create any new objects, and prevents him from doing so if he has exceeded his disc space quota. The accounting demon also checks the file structure as held on the file server for integrity and consistency, so that any corruption can be detected at an early stage, and hopefully be corrected before it has a chance to spread.

Apart from caching and protection, there are two other relevant points which should be made about the TRIPOS filing machine system. First, since the filing machine assumes that it is the only user of the TRIPOS files on the file server, all interlocking against conflicting client access may be performed within the filing machine. Interlocking within the filing macine is at least an order of magnitude faster than obtaining an interlock in the fileserver, since no network communication is required. This results in a substantial improvement in performance, particularly when searching trees of directories and files for a particular object. In the old system, there were two ring requests at each directory level, as file server interlocks were obtained and released; in the filing machine system, there is just a single client request to locate the object.

Second, the protocols used between the client and filing machines are designed specifically for the TRIPOS system. Since messages can be lost in the communications network, they are made sufficiently reliable by a combination of serial identification and idempotent definition. The protocols also permit a client machine to request several filing machine operations concurrently.

Implementation: Protocols

The protocols used between the client and the filing machine follow the remote procedure call paradigm. This may simply be described as an alternating series of data exchanges between two participants. Each participant knows how much data to expect from the other on each occasion, so there is no notion of flow control, and each participant knows that it will not receive another call until the other has received the results of the last one. (There are more elaborate ways of defining RPC, but this will suffice for present purposes.) Rather than using a general RPC implementation, we have chosen to take advantage of the known properties of the interaction between the filing machine and its client in order to achieve simplicity. A client request is acknowledged by the receipt of the results of that request. When the recipient of a general remote procedure call delivers its results, these are acknowledged by the receipt of the next call. Until the next call the results need to be retained, in case they were lost in transmission, causing the client eventually to repeat the call that generated them. If the results are bulky it may be inconvenient to hold on to them until the next call (the delay until which is unknown) and some complication may thus be needed in order to obtain earlier acknowledgement in this case. In our implementation it has been possible to define the calls either so that they give results so small that there is no embarrassment in holding on to them, or so that the call is strictly repeatable with identical results in which case there is no need to retain them.

A client machine when first booted contacts the filing machine and requests the initiation of a session [9] and of a number of series within that session. The session corresponds to the interval between a client machine being bootloaded and its crashing or being rebooted. A series corresponds to a single train of procedure calls, so that the basic RPC discipline in which the next call acknowledges the last reply applies to the next call belonging to a particular series in a particular session. A

specific call must accordingly contain the session identifier, the series identifier, and the serial number of the call within its series. The filing machine checks the validity of session and series and acts on the basis of the serial number in the obvious ways.

There is one case in which the TRIPOS calls fit uneasily into the RPC model - the client writing data to a file. The badness of fit comes from the unpredictable amount of material to be written. Rather than set an arbitrary (and rather small) maximum, we have arranged that the initial call is a 'request permission to send x bytes' and its reply is a 'go ahead to send x bytes'. This is the method used, for the same reason, by the underlying file server. The filing machine retains knowledge that the data has arrived until there is a new call in the series, so if the acknowledgement of data receipt is lost and the whole transaction retried later then the data itself need not be retransmitted (the client receives a reply "data already received").

Sessions (not series) which are otherwise inactive are kept alive by a periodic idle handshake (initiated by the client). The filing machine uses the reply to such handshakes in order to return some miscellaneous status information, such as a current 'message of the day'. In the event of a session timing out, all unwritten data is flushed from the cache, and any outstanding interlocks for that session are broken. A session is also cancelled if a client machine which already has a session requests a new one, on the assumption that the client has crashed and has been rebooted. An improved arrangement might be to have the Resource Manager [1] (the machine which manages the allocation and deallocation of processor bank machines) control the creation and cancellation of sessions. When an instance of TRIPOS is started, the resource manager would inform the filing machine; when the client machine was returned to the pool of free machines, or rebooted, the filing machine would again be informed. The only shortcoming would occur with client machines outside the control of the Resource Manager.

Implementation: Caching Strategies

The operations provided by the filing machine are tailored to those required by client TRIPOS systems. A single client-filing machine request may correspond to several filing machine-file server requests. However, many of the latter requests are eliminated, or can be performed asynchronously, by use of the filing machine's caches, which are now described.

The data cache is implemented as a number of fixed-size blocks, which are always aligned with respect to the file server file on a boundary which is a multiple of their own size. Since directories are observed to be fairly small, and therefore suggest a small block size, while for files a large block size is desirable in order to reduce file server protocol overheads, it was decided to use two sizes of cache block: a small one for directories, and a larger one for files. The exact sizes chosen are based on the known implementation of the file server and low-level ring protocols. Directory cache blocks are 512 bytes long; these can be read or written to the file server in a single request-reply operation, and few directories will require more than two such blocks. For files, the size is 2048 bytes; this is the main file server disc block size (hence cache blocks align exactly with disc blocks), and is also the maximum block size possible with the basic ring level protocol. The filing machine would still work with a ring or file server with different characteristics, but might require some adjustment in order to achieve best performance.

Cache blocks are held in both a hash table for efficient access, and in a simple linear queue for allocation and deallocation. Allocation and deallocation are administered using a least-recently used (LRU) mechanism, with blocks being moved to the back of the queue whenever they are touched, and being deallocated from the front. It is unlikely that this scheme is optimal. Some trials were made with an alternative which ordered cache blocks on the basis of time-weighted use, but no convincing improvement in cache hit rate was observed. Unfortunately, the time involved in collecting significant statistics is too great to permit frequent changes to the algorithms. Simulation using data logged from real operation would probably yield better information.

In day-to-day use, the data cache achieves around a 90% hit rate for reading. By this we mean that in only about one in ten cases is a required block of a file server file not present in the cache, so that a request is held up awaiting the completion of a file server transaction. Since a particular client transaction may require several blocks from a file, the hit rate as seen by client requests may be smaller. However, it is probable that misses are grouped on some client transactions, rather than being evenly distributed. There does not appear to be a great change in this figure with load [figure 1], up to the peak recorded load of about 50000 cache requests per hour. With increasing load (corresponding to increased numbers of clients), the greater spread of objects accessed might be expected to reduce the hit rate. However, a greater proportion of all accesses would be to shared system objects, increasing the hit rate to some extent.

The current system runs on a Motorola 68000 based machine, with 1M byte of memory. Of this, about 660K bytes are allocated to the data cache. Some simple simulation and logging of access was done, mainly in

order to investigate the possible gains to be made by attaching a local disc, to act as a much extended cache. However, using a very simple LRU mechanism (applied to entire files rather than individual cache buffers), it was found that the hit rate improved little once the cache size reached around a megabyte. This suggests that a local disc need not be large, but should be fast. The conclusion was drawn that it would be better to use more main memory, and take advantage of its short access time.

The second part of the cache is the abstracted directory structure. Although all the information that is retained in the directory map could be held within the data cache described above, memory space can be more efficiently used if the relevant information is extracted. This arises since the map contains copies of individual directory entries, while the data cache would have to hold the entire enclosing block (or two blocks, if the entry straddles a block boundary). The map consists of a number of (directory,entryname) to (object) mappings, and is used whenever a directory is searched for an entry. Only if the entry is not found is the data cache, and ultimately the data stored on disc, used. In order to keep the map simple, it is not used when a directory is to be updated, and the data cache must then be used. Since the vast majority of directory operations are searches, this does not seriously degrade performance.

Map entries are maintained using an LRU algorithm as for the data cache. Since the lifetime of a map entry is believed to be much greater than that of a cache block (indeed, entries for commonly used system commands are likely to be permanant), there seems little need to try more sophisticated allocation mechanisms. In the current implemention there is sufficient space to retain entries for 150 directories with a total of 500 entries. Directory searching operations succeed in the directory map about 65% of the time; of the remaining 35%, about half are for entries which do not in any case exist. These figures are observed with about 660K of memory being used for the data cache and 64K for the directory map.

Some consideration was given to the inclusion in the map of 'this entry does not exist' mappings. However, this was rejected on the following grounds. When TRIPOS is presented with a command, it first searches the user's own directory, before trying the system command directory. Thus, a large proportion of the map failures are attempts to find system commands in user directories. In the present scheme, the map will contain at most a single entry for a system command in the command directory. If 'does not exist' mappings were introduced, then there would be a large probability of an entry corresponding to each current user, and the effective amount of information contained in the map would be much reduced.

For both the data cache and the directory map, the filing machine is designed on the assumption that it is the only machine which changes the contents of the filing system. We do not regard this as being a serious limitation; it would no doubt be possible to design a distributed implementation of the service the filing machine is designed to give.

Implementation: Interlocks

In TRIPOS, interlocks perform two functions. First, they provide a mechanism by which conflicting access requests can be resolved. Secondly, the representation of an interlock forms a token which can be used to identify an object. For example, if a client successfully requests that a file be opened for output, an interlock on that file is established, and the client is sent a copy of the associated token. In order to have a subsequent write operation performed, the token must be presented. The general locking scheme is multiple-read, single-write.

In certain cases, an interlock may be used solely as a token, and does not confer any particular access rights, other than to guarantee that the object cannot be deleted by another client. Such an interlock is referred to as a **void** lock. This situation arises in TRIPOS where a user has an interlock on a directory which he may wish to search and update at intervals (for example, his 'home' directory). If he were given a write interlock, then no other client could even search the directory (as this would require reading), while a read interlock would prevent update of the directory by any client, including himself. These are equivalent to **nolock reads** implemented in LOCUS [10] at about the same time as in the filing machine.

The filing machine implementation associates three access bits with each interlock, for read, write, and delete. A void interlock has none of these set, and is converted to a read or read-write interlock whenever searching or update is required (and converted back afterwards). Interlocks within the filing machine are also be divided into two groups, those that have been given to clients, and those which are purely internal. The former group may last for an arbitrarily long time, while the latter are transitory. If a request to the filing machine causes a conflict with an external lock, the request is rejected; if the conflict is with an internal lock, then the request is queued until the lock is removed, or changed to an external lock. As all access to directories can be performed using internal interlocks, conflicts over directories are never seen by clients.

In the protection mechanism, interlocks also carry protection information. This is particularly important in the case of void interlocks on directories. Here the protection information forms the basis on which the maximum access permitted to any object reached from the directory is calculated. As an object is located by following its path name from some directory, transient internal interlocks are obtained on the directories encountered along the path. At each stage, the access to a directory is determined from the access available to the previous directory, and the access control bits (called the access matrix) associated with the entry which referred to it. Since all the protection information is present within the interlocks and the directory entries which are processed during the path search, the protection mechanism involves only a very small overhead compared to an unprotected system, and no extra data traffic results.

Filing machine interlocks are thus tailored to the requirements of TRIPOS; they are both necessary and sufficient, and efficient.

Implementation: The Client Stub

There is no restriction on who may use the filing machine, provided that they conform to the specified protocols. At the time of writing there is, however, only a single implementation of a client stub, which is used on all instances of TRIPOS. The number of user program requests which the stub should handle simultaneously is specified when the stub task is started, and is limited to eight only by the filing machine protocols. In practice, this is set to two, as it has been observed in an equivalent TRIPOS system that this leads to the queueing of requests (because two are already outstanding) in less than 5% of cases. The stub can also handle an asynchronous data transfer to or from its local cache, and a periodic idle handshake, simultaneously with these two requests. There are thus three series within each session.

It is unfortunate that the local cache is necessary. Without it, the amount of code and storage needed by the stub is less than half of that required by the filing system code for a system that communicates directly with the fileserver (and can be reduced still further by specifying that only a single request be performed at any one time). However, the performance improvements derived from the local cache are sufficient to justify its existence except in machines with limited memory. The stub performs a limited amount of read-ahead, and asynchronous writing, provided that there are local cache buffers available. For reading, this yields a hit rate (measured in the same way as for the filing machine) of about 70%. Of the remaining 30%, about half are already under way to the local cache when the user program makes a read request.

Observed Results

Accompanying this description of the TRIPOS filing machine are some measurements made during normal use. Diagram 1 shows the filing machine cache hit rate for reading as a function of the load as measured by the number of requests. Hit rates were logged at hourly intervals; the diagram indicates the limits beyond which very few occurances were recorded. Tables 1 and 2 present average response times for requests from the filing machine to the file server, and from the client stub to the filing machine. In both cases typical TRIPOS operations which cause these operations are listed; it should be remembered that most TRIPOS operations are equivalent to several file server operations.

Extensions

The filing machine project was largely complete when an opportunity arose to experiment with the same system in a rather different context. The Universe project [11] interconnects local networks (in fact all Cambridge Rings) at seven sites by means of a geostationary satellite. As part of a programme of distributed computing experiments using this network, it was decided to explore the practicality of separating the filing machine from the file server by the satellite link. There is little intrinsic difficulty in doing this, particularly if one concentrates on performance, rather than interlock issues. Machines were inserted in the ring at the Rutherford Laboratory in Oxfordshire, and arrangements made to cause them to use the Cambridge bootstrap service rather than the local service. By suitable settings in the boot server's tables, it can be arranged that it is only necessary for the machines to be switched on and for two simple commands to be issued at an ordinary terminal in Cambridge in order to make Tripos, using the Cambridge filing system, available at the Rutherford. Some performance measurements are given in table 2. These are about as one might expect.

Any action which requires synchronous operations to be performed (eg., creation of a brand new file) has a substantially greater turnround because of the delay in the satellite link. Once a file has started coming, the performance perceived remotely differs much less compared to the performance perceived locally, due to read-ahead being performed by the filing machine. Similarly, writing is almost as fast, though the final close operation is slower as the cache and file server must be synchronised. The figures in table 3 were obtained over a period of about two hours, with a single client system. It is expected that the average times will improve over a

longer sampling period, as the effects of caching become more pronounced.

This system was used for a limited amount of work, and proved quite tolerable. With a single client there is ample room in the cache for the common system commands to be permanently resident. From the user's point of view, performance is much as expected; simple commands take a relatively longer time, but more complicated ones (compilations, for example) are not observably much slower. It is anticipated that this system will be used more in the future.

Complications arise when interlocking and the consistency of cached data is considered in situations where there are multiple filing machines. In the context of the Universe project, the following scheme has been considered, although no implementation work has yet been done.

A filing machine at a site attached to a particular satellite ground station would 'own' all the file servers on that site, and would have primary control over access to objects on those file servers. Should a filing machine at a remote site R wish to access objects physically stored at local site C then it would first obtain permission from the filing machine at C. That would take out an interlock on the object in the file server at C, and return the identifier representing the interlock to the filing machine at R. It would also provide a time stamp giving the time at which the object was last updated, so that the remote machine can ascertain the validity of any data which it may have cached. The filing machine at R would then bypass the filing machine at C and communicate directly with the fileserver. Only when it has completed its transaction will it further communicate with the local filing machine at C (other than perhaps to maintain an idle handshake).

The need to obtain an interlock in the fileserver causes a little inefficiency. However, if such an interlock were not invoked, then all communication would have to go through the both filing machines. This would be a source of further overheads to both sites. If the filing machine at R crashes, then the idle handshake will time out, and the filing machine at C can release the interlock in 'its' file server. If the local filing machine crashes and is rebooted, then the object will still be safely locked in the file server. Should both filing machines crash, then the file server interlock will time out of its own accord. Finally, should the file server crash, the interlock will become invalid, and the filing machines will become aware of this when the file server is restarted.

The provision of the time stamp removes the need for one site to inform all others when it updates an objects. In the scheme above, filing machines may be introduced, and subsequently removed, from the system, without the need to inform all other sites. Time does not have to be accurately synchronised between sites, since all time stamps needed for an object will be provided by the site which 'owns' the object. It is thus only neccessary that the time at each site be monotonically increasing, and can safely be derived from some local service.

Conclusions

It is probable that a more efficient system could be provided by a single machine which combined the functions of the filing machine and the file server. This could be done either by implementing a single dedicated TRIPOS file server, or by running the file server and the filing machine as separate virtual machines on the same physical hardware. These would eliminate some communications overhead, and allow optimisation of the disc data structures. It would be the obvious solution if the problem was simply that of providing a centralised filing system for TRIPOS. There are however two reasons for not doing this.

First, in the Cambridge environment, there are several filing system clients. The file servers provides the basic facilities which are needed by all clients, notably garbage collection and maintainance of disc integrity, and saves their duplication and reimplementation every time a new system is created. It would be comparatively easy to provide further services analogous to the TRIPOS filing machine, for example a data-base server. The centralisation on a single file server allows, in principal, links between the filing systems of different clients. Also, it has proved quite easy to modify the filing machine, and to add new facilities to the client interface, without major upheavals and inconvenience to users. Second, we do not have the necessary hardware available. It is unfortunate that the hardware of the file servers does not lend itself to experiment, as the machines are not large enough. We cannot therefore give comparisoms between the performance of the system we have and one with a file server machine dedicated to the needs of TRIPOS, even on an experimantal basis.

We believe however, that the practical success of the TRIPOS filing machine demonstrates several useful results. First, there is great advantage to be gained from having a front end to a file server that takes advantage of knowledge of the patterns of use expected of a particular subset of the file server's clients. Secondly, this advantage outweighs the cost of transmitting data more than once over a local network. Thirdly, the remote procedure call paradigm is powerful and appropriate in this case. Finally,

the effects of caching in memory the results of large-scale file server transactions in order to retail the results to clients in smaller amounts are sufficient to mask to a considerable and useful degree even the delays caused by the presence of a satellite segment in the network.

Table 1 : Filing Machine to File Server request times in MilliSeconds

	Local	Sat	
Create Index	364.3		creating new directory
Retain	554.9	1243	used during rename
Delete	348.8	465	delete or during rename
Create File	456.7	1077	create new file
Open	127.3	841	(
Ensure	255.2		updating a directory
Close	273.8		(
Read	316.0	1010	read >=2048 bytes
Write	369.7	1350	write >=2048 bytes
SSP Read	152.6	995	read <= 512 bytes
SSP Write	188.6	980	write <= 512 bytes

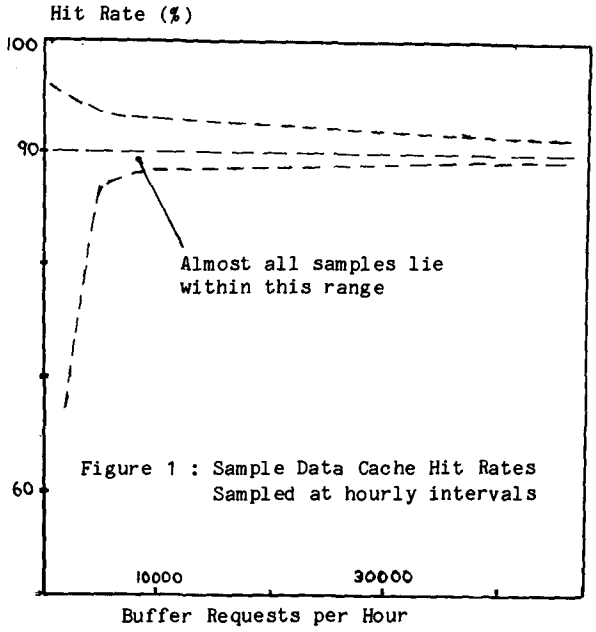

Figure 1 : Sample Data Cache Hit Rates Sampled at hourly intervals

Table 2 : Client Stub to Filing Machine request times in MilliSeconds

	light load <4 clients	heavy load >8 clients	over satellite	
Locate	88.2	118.6	1489	(find a directory)
Free	35.6	36.7	15	(release void interlock)
Copy	16.0	67.4	12	(copy a void interlock)
Delete	180.0		4570	(delete a file)
Rename	1590.0	3201.3	5326	(rename a file)
ExamineObj		202.5	877	(info about an object)
ExamineNext		243.2	519	(examining a directory)
SetAccess	20.0	20.0		(change access to object)
FindInput	113.2	233.9	549	(open a file for input)
FindOutput	1280.0	1666.2	6590	(open a file for output)
Close	45.4	111.9	306	(close a file)
Read	72.8	260.7	468	(read data operation)
Write		204.7	233	(write data operation)
SSPRead	23.7	50.2	59	(read into local cache)
SSPWrite	20.3	53.2	119	(write from local cache)
Refresh	3.2	11.9		(idle handshake)

References

[1] R.M.Needham and A.J.Herbert : The Cambridge Distributed Computing System -
International Computer Science Series, Addison-Wesley, 1982

[2] J.Dion : Reliable Storage on a Local Area Network -
Ph.D. Thesis, Cambridge Computer Laboratory, 1980
: The Cambridge Fileserver -
ACM Operating Systems Review 14(4), (pp26-35), Oct 1980

[3] J.G.Mitchell and J.Dion : A Comparisom of Two Network-Based File Servers -
Communications of the ACM, New York, 25, 2, 233-45

[4] A.D.Birrel and R.M.Needham : A Universal File Server -
IEEE Transactions on Software Engineering - Sep 1980, 450-53

[5] M.Richards et al. : TRIPOS - A Portable Operating System for MiniComputers -
Software - Practice and Experience, Jun 1979

[6] B.J.Knight : Portable System Software for Personal Computers on a Network -
Ph.D. Thesis, Cambridge Computer Laboratory, 1982

[7] B.Nelson : Remote Procedure Calls -
Ph.D. Thesis, Carnegie-Mellon University, 1981

[8] M.V.Wilkes and R.M.Needham : The Cambridge CAP Computer -
Operating and Programming Systems Series (No 6), North Holland 1979

[9] C.N.R.Dellar : A Fileserver for a Network of Low Cost Personal MiniComputers -
Software - Practice and Experience, Nov 1982 (section 7.4)

[10] C.Popek et al. : LOCUS: A Network Transparent, High Reliability Distributed System -
Proceedings of the 8th Symposium on Operating Systems Principals, 1981

[11] Kirstein et al. : The UNIVERSE Project -
Proceedings of the 6th Conference on Computer Communications, 1982