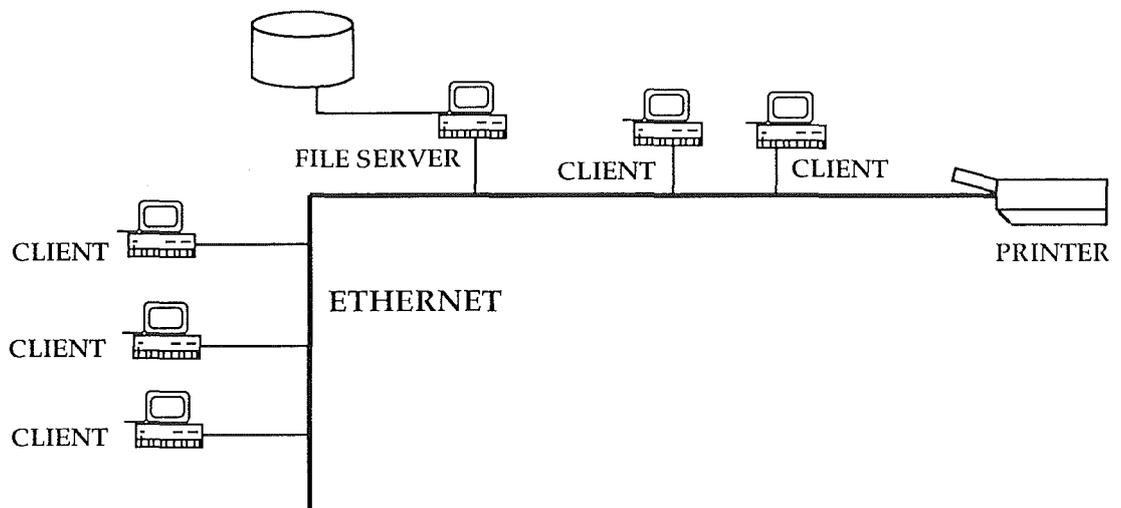


HONOURS PROJECT REPORT
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF CANTERBURY

PERFORMANCE MODELLING OF LAN RUNNING NFS



LAN RUNNING NFS

WAH TUCK FUI
SUPERVISOR : RAY HUNT

Contents

1 Introduction	1
1.1 Background	1
1.2 NFS	1
1.3 Objective	2
2 Response Time	3
3 Model Discussion	4
3.1 RPC	5
3.2 Cache	6
3.3 NFSD	8
3.4 Disk Drive	8
3.5 Network Traffic	9
3.6 Client	9
3.7 Server	12
3.8 Input parameter for the model	14
3.9 Output of the model	15
3.10 Limitation of the model	17
4 Model Validation	18
5 Simulation result of interest	20
5.1 Effect of server cache size on response time	20
5.2 Effect of the number of disk drives on response time	21
5.3 Effect of the number of NFSD process on response time	22
5.4 Effect of the network load on response time	24
5.5 Effect of faster disk drives on response time	25
6 Case study	26
7 Conclusion	27
Appendices	
1 References	
2 Program listing	

1. INTRODUCTION

1.1 BACKGROUND

A distributed computer system is based on a set of separate computers which are linked by a computer network, capable of autonomous operations. This design enables the individual computers, of which they are composed to utilise shared resources in the network, providing computing facilities that could be superior to the conventional, centralised or "mainframe" computers in flexibility and application. Users of a distributed system are given the impression of using a single integrated computing facility, although the facility is actually provided by more than one computer and the computers may be in different locations.

One of the most popular distributed system architectures is the **workstation/server model** or the **client/server model**. In this model, the client provides the computing power to execute application programs for a single user or even several users in a multi-user environment such as UNIX¹. The clients are integrated by the use of communication software supplying everyone the access to the same set of servers. The servers provide access to shared devices-printer, disk storage, files and other network resources.

1.2 NFS

Network File System (NFS) is a file system protocol developed by SUN Microsystems for distributed systems.

It is an extension of the UNIX operating systems that provide a distributed file service based on networked UNIX systems. However, it is also available in MS-DOS on IBM PC's and other non-UNIX operating systems.

NFS provides access to conventional UNIX file stores. A client program need not be aware that the file that it is using is distributed. Programs in a client can access remote files and directories using the usual UNIX primitives. Programs written to operate on local file can be used with remote file without modification.

The NFS service is implemented in terms of remote procedure calls (RPC) between kernels. When an application program uses the UNIX file primitives to access a local file, the local kernel behaves as it would in a conventional UNIX system. If the file is in a remote directory the kernel makes a service call to the appropriate remote kernel.

NFS adheres to a stateless-server model. This means that the server retains no information about its client between RPC requests. The local kernel holds all information on open files for each program using v-node (which consist of i-node² number, current position of read/write pointer for each file). This simplifies server implementation.

An NFS client periodically checks with the server to see if a file in its cache has been modified; if so, the client invalidates its cache for that file.

¹ UNIX is a registered trademark of AT&T Bell Laboratories

² i-node contains the following information about a file : type of file(ordinary,directory, special), owners' ID, size, time of last access, last modification and pointer to disk block containing the file's contents.

The interval between checks is a compromise between performance (frequent checking loads the server and delays the client) and consistency (insufficient checking may mean that a client uses stale data from its cache).

NFS follows a write-through policy, i.e. an NFS server is required to write data to stable storage before returning from remote procedure call, the cached information that is vulnerable to loss during a crash is minimised.

The write-through policy has a distinct performance disadvantages that is its write-through limits the performance benefits of client-side caching, since a server disk access is required for every write.

1.3 OBJECTIVE

Users on a LAN running NFS often complain about the slow response time they are experiencing. Since most clients on NFS are diskless, they will have to compete against each other to access the shared resources of the network, namely the file server and LAN, thereby increasing the response time.

The **objective** of this project is to locate the bottleneck in the LAN which causes the slow response time and determine ways in which the response time can be reduced.

NFS was chosen as the distributed system to be investigated for the following reasons

- i) it is available in the Computer Science Department of the University of Canterbury
- ii) the technical details on the running of NFS is available from the SUN Network Programming Manual.

A discrete-event simulation model of NFS was developed and used to study the file systems performance under various workload condition to locate the bottleneck and the causes of slow response in a network. An analytical model is not used as NFS is too complex to lead to a simple solution.

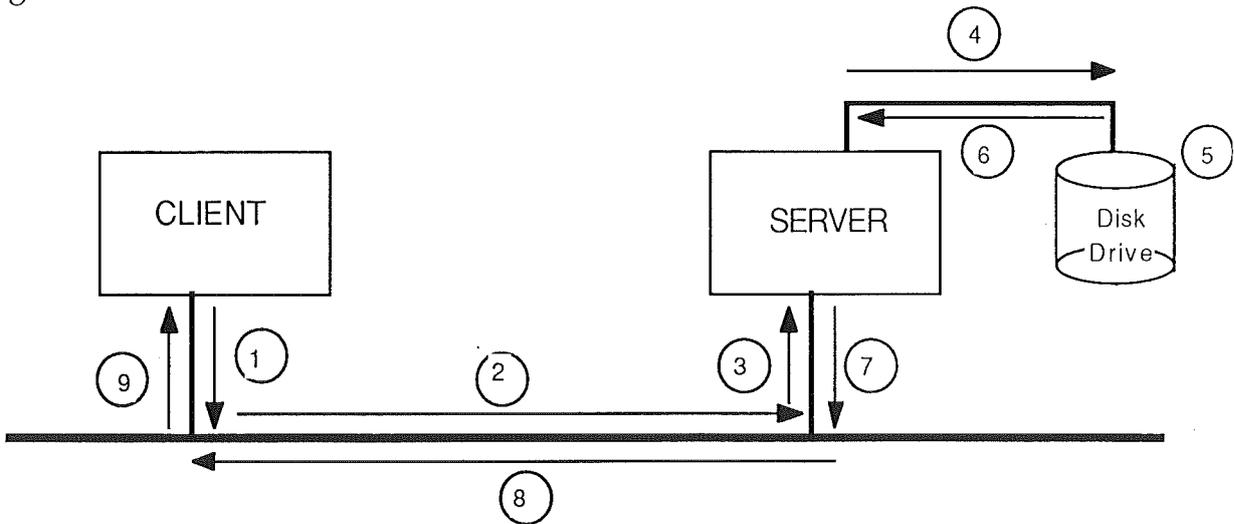
The model that was developed is able to provide the system administrator a tool to modify network configuration and conduct experiments in order to improve the response time without making changes to the actual network.

2 RESPONSE TIME

The location of the bottleneck is of primary interest in this investigation. In order to obtain this location, it has been decided to use response time.

Response time in this report is defined as being the *time taken from when the RPC software of the client first transmits the RPC call request to the server, to the time the client receives the reply comes back from the server.*

Response time is then broken into nine components, as illustrated in the diagram below. When a client transmit a RPC request to the server, the packet generated will travel from the client to the server and back, collecting the time spent at every component. Analysing the components of the response time for the location where the packet spent the longest time will give us the location of the bottleneck in the LAN.

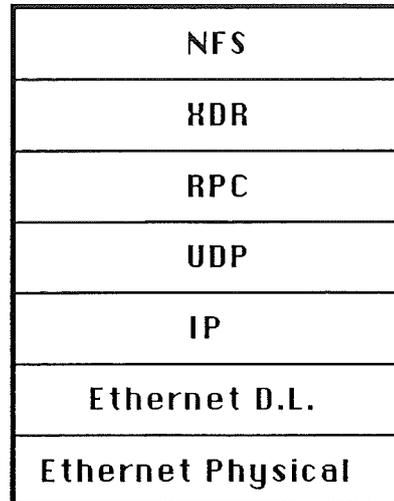


Components of the response time are :

- 1) Queue time in client before transmission
- 2) Transmission time
- 3) Queue time before server receives packet
- 4) Queue time to disk drive
- 5) Disk access time
- 6) Queue time out of disk
- 7) Queue time in server before transmission
- 8) Transmission time
- 9) Queue time before client receives packet

3 MODEL DISCUSSION

In order to perform its distributed file system service, NFS uses the following protocols as a base [Dupre-Blussseau 1989]:



(The figure above shows the components of the OSI 7 layer architecture in a LAN running NFS)

- **XDR** (eXternal Data Representation) : performs a task equivalent to that of the Presentation layer of the OSI reference model. The purpose of XDR protocol is simply to reorganize data and make it comprehensible for the various machines communicating with each other. This task has no effect on network load, since all work is done locally. This is **not** modelled in this project as response time is defined to be from the RPC protocol and below.

- **RPC** (Remote Procedure Call) : operates at a level equivalent to the Session layer of the OSI reference model. Only the procedure call that is transmitted via the network constitutes as a load to the server and network are modelled.

- **UDP** (User Datagram Protocol) : operates at a level equivalent to the Transport layer of the OSI reference model. The processing performed by the UDP protocol is very limited since there is no check at this level. This associated service time is negligible and the protocol is **not** modelled. **UDP** is used instead of **TCP** in NFS because **TCP** is a reliable protocol which requires the overheads of call setup and shutdown which will cause transmission delays.

- **IP** (Internet Protocol) : operates at a level equivalent to the Network layer of the OSI reference model. The processing performed by the IP protocol basically consists of determining if the datagram received has to be fragmented before transmission to the subordinate protocol, Ethernet, which has a maximum data frame of 1518 bytes. When the IP receives a datagram larger than this, it has to break it down into several smaller elements. When the IP receives frames from the Ethernet protocol, it must rebuild the original datagram (if it has been fragmented) and then send it to the higher protocol. The times for fragmenting and re-assembly of a datagram are each 4.5 msec each, giving 9 msec in all. When a datagram contains no more than one Ethernet frame, the IP protocol processing simply

corresponds to the data length test, i.e. 1.5 msec in and out, or 3 msec in all [Dupre-Blusseau 1989]. This protocol is **not** modelled as the service time taken is fixed for all the packets generated. A fixed time is therefore assigned to simulate the IP.

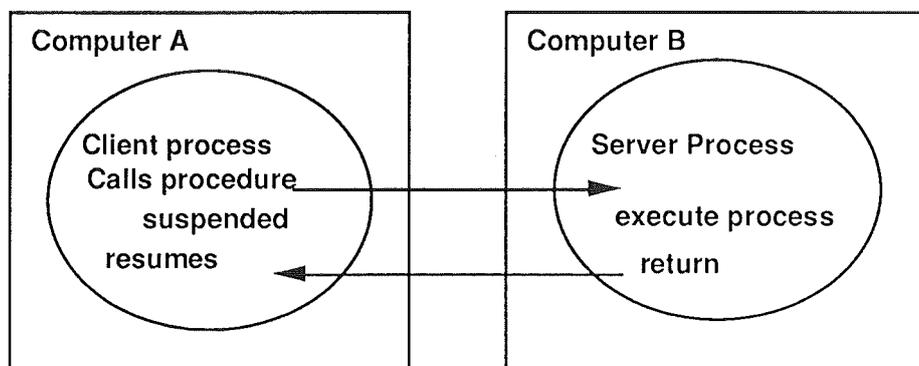
- **Ethernet** : operates at a level of the Physical and Link layers of the OSI reference model. NETMOS [Marriot 1986], a LAN Simulation Context is used to provide the Ethernet routines used in this model. Routines of clients and servers are written in DEMOS[Birtwistle] and it uses the Ethernet routines of NETMOS.

The major components of NFS that are identified for this project and therefore modelled are :

- i) RPC
- ii) cache (see Section 3.2)
- iii) nfsd (see Section 3.3)
- iv) disk drive
- v) network traffic
- vi) client
- vii) server

3.1 RPC

Remote Procedure Call (RPC) is the means of communication between computers in a distributed system. RPC calls transforms a network of individual computers into a single large computer to the eye of the users. It is modelled on the local procedure call, but the called procedure is executed by a different process and usually on different computer from the caller.



The diagram above shows a RPC request from computer A to computer B. The client process in computer A is suspended until it receives the reply from computer B which acts as a server. When the server process finishes processing the request a response is generated to the caller. Upon receiving the reply, the client process resumes processing. If the suspended time is long, the response time is said to be long by the user.

Application programs can make use of distributed services by calls to remote procedure by name without knowing their locations thus allowing complete configurability of a distributed system in terms of the location of its component.

Request is the term given to RPC generated by the client to the server. The client generates a RPC request when it requires a service from the server. **Response** is the reply by the server to the client.

To model RPC in NFS, the types of RPC requests and responses generated are needed. `nfsstat` - Network File System statistics, displays statistical information about the NFS and RPC interfaces to the kernel, is used to obtain the information required. The output of `nfsstat` are as follows:

Client rpc:

```
calls    badcalls  retrans   badxid   timeout  wait     newcred
257080   1         37        14       38       0        0
```

Client nfs:

```
calls    badcalls  nclget    nclsleep
257041   0         257071    0
```

```
null      getattr   setattr   root      lookup    readlink  read
0 0% 110933 43% 2724 1% 0 0% 51623 20% 33205 12% 6842 2%
```

```
wrcache   write     create     remove    rename    link      symlink
0 0% 22867 8% 19385 7% 1601 0% 544 0% 119 0% 0 0%
```

```
mkdir     rmdir     readdir    fsstat
8 0% 10 0% 7153 2% 27 0%
```

The ten popular RPC requests modelled and their purpose are listed in the table below :

RPC request	Purpose
getattr	get attributes of a file
*setattr	set attributes of a file
lookup	look up file name
readlink	read from symbolic link
read	read from file
*write	write to file
*create	create a new file
*remove	remove a file
readdir	read from a directory
fsstat	get filesystem attribute

The RPC request which has an "*" beside it is a write request. This means that these requests are written directly to the disk (NFS's write through policy). The rest of the requests are read requests which is first checked against the server cache. If it is found in the cache, an instant response is experienced by the client else a disk read access is required.

3.2 CACHE

Cache is an area of main memory which buffers frequently used/most recently used instructions and data in the high speed memories close to the computer. If a program issues a read request that is already in the cache, it can be satisfied without a disk access.

NFS uses caching for both the clients and servers. Client caches the results of read operation from remote files while the server caches the pages of files that they access just as they would for local files.

It is used in many operating systems to improve file system performance. A request by the computer that is **not** found in the cache requires disk access. This slows down the execution of the program.

Advantages of caching are:

- 1) it reduces delay, requests found in cache can be returned to a suspended process much faster than one that must be fetched from disk.
- 2) it reduces contention for disk arm, which is an advantage when several processes are attempting to access files on the same disk.

The effectiveness of a cache is measured as the **hit rate** or hit ratio, which is the fraction of I/O's that are captured in the cache, i.e. it does not require access to the disk drive.

Modelling of the client cache will be discussed in Section 3.5 . Modelling a server cache is extremely complicated as there are many considerations that must be taken into account. The few more important ones are:

- 1) size of cache,
- 2) load on the server (in terms of the number of clients it is supporting),
- 3) type of paging algorithm used,
- 4) block size of cache.

Literature on cache performance of NFS [Lyon, Sandberg 1989] have reported server cache hit rate of between 80 - 98% but there is no mention of any of the four factors above. As there is no literature on the NFS server cache performance on the four factors above, the cache cannot be modelled properly.

It has been decided that the server cache hit-rate should be determined by the user. The hit rates can be fine tuned by experimenting with **nhfsstone** (see model validation) and the simulation output until it reaches a satisfactory result.

It has been observed (from the output of **nhfsstone**) that hit rate for certain RPC requests are much higher than others. The rank for the hit rate are as follows

- i) *getattr* & *fsstat*
- ii) *lookup*, *readdir*, *readlink*
- iii) *read*

This is understandable as NFS follows a stateless protocol, and all clients will need to check the consistency of the files in its own (the clients') cache. RPC request like *getattr* will have higher hit-rates as it is accessed regularly. Read request has the lowest hit-rate as the space a page occupies in the server cache is likely to be swapped out quickly by a more recent read request, this is especially true when the load on the server is heavy. A miss will then occur and a disk access is needed.

Experimental results have shown that the hit-rate of the read request is about 40% of the *lookup*, *readdir* & *readlink* request which is between 80% to 98% mentioned above. The hit-rate of *getattr* and *fsstat* is about 95 % from experiments conducted.

The ideal model for the server cache would be as follows. The user inputs the size of server cache and the load on the server in terms of the number of RPC requests it receives a second. A table lookup is then used to approximate the cache hit-rate for all the requests. A more sophisticated model will also include the modelling of server cpu, as this will affect the

time a client RPC request needs to wait at the server queue before it is attended by the server. However this is not implemented in this project as there is no study on the server cache hit-rate based on the factors mentioned above.

The current model is merely a draw distribution which is either true or false at every draw to simulate the cache. The cache hit-rate for getattr is fixed at 97%, fsstat is fixed at 95%. The user would have to input the cache hit-rate of lookup, readdir and readlink, which is approximately 80% - 90%. The model then calculates the hit-rate of read which is 40% of the hit-rate of lookup. This can be easily modified by future users when a better model of the cache is discovered.

3.3 NFSD

Network File Systems Daemon (NFSD) are processes that handle client file system requests. The number of nfsd processes limits the number of requests that can be serviced by the file server at any one time.

Modelling the NFSD processes is easy by declaring these processes as **resources** in DEMOS [Birtwistle]. All requests to the server would need to acquire these resources in order to be served. After the request has been served, the NFSD process is freed to be used by another request.

3.4 DISK DRIVE

A disk drive is a permanent storage device for files and programs in a computer. It is not volatile; a power failure would not cause the loss of data.

The access time for a disk drive is very much longer than to high speed memory. The disk drive is a mechanical device with the disk spinning and the read/write head arm moving to the correct track.

The time taken to complete a read/write request varies greatly as there are three factors to be taken into account:

- 1) average disk seek time,
- 2) disk rotational delay,
- 3) the number of disk access needed to complete the request,
eg: write to disk -requires 3 disk access:
 - a) read the disk for the location to write to,
 - b) write the data to disk,
 - c) write to i-node.read from disk -requires at least 1 disk access:
 - a) read from disk.

The exact number of disk access for a read request is not known as a read request from the client include the i-node for the file required. However this i-node may point to another i-node and another until one which points to the location of the file in the disk.

In view of this difficulty of obtaining the number of disk access for a read request, a decision was made to assign **fix times** for the read and write request. The user of the model would have to give the read time and write time (in milliseconds) which is reasonable for the disk drive.

A good approximation for the read and write time are

i) read time = 2 * disk seek time + 1/2 rotational delay

ii) write time = 3 * disk seek time + 1/2 rotational delay

this observation is made by comparing the output of the model with the output of nhfsstone (see model validation).

3.5 Network Traffic

Traffic mentioned here are packets that are transmitted by clients and server(s) which are not RPC requests, e.g. electronic mail messages or file transfer packets. These packets may not have even been generated by any of the clients or server(s) in a network as it may be packets from other networks somewhere in the world.

These packets will certainly cause transmission delay in a network and are therefore modelled. Network traffic is a packet generator which randomly generates packets to any clients and server(s) in the network. The packets generated are called TRAFFIC packets in the model. A client or server which receives these packets ignores them as no processing are required for it.

3.6 CLIENT

A client is a computer which provides the computing power in a network. It uses the resources provided by the server (see Section 3.6).

To model a client, the knowledge of the workings of a client running NFS is required. As this project only models NFS from the RPC layer and below, the reasons why a client makes RPC request is **not** needed (see Section 3.1).

When an application program requires access a page which is not found in the client cache, a page-fault occurs. To satisfies this page-fault, the client makes a RPC read request to the server for the page required. When a user on a client saves a file, a write request is generated.

Factors that needs to be taken into account when modelling the client are :

- 1) memory/cache size of client
 - clients with smaller memory will have more page-faults. This causes a greater number of RPC requests, especially read requests.
- 2) load on the client
 - the greater the number of users on the client, the smaller the allocation of memory for each user. This will cause greater number of page-faults and therefore generates more RPC requests.
- 3) application program being executed
 - large programs which requires more memory tend to cause more page faults, therefore more RPC requests.

When developing the model for the client, robustness of the model must be taken into account. It would be inappropriate to model a client running

- a)x number of file editing process,
- b)y number of database retrieval process,..... etc with
- c)z megabytes of memory/cache size.

each time without being able to modify the three factors mentioned above.

The model developed must be valid for any applications that are being executed in the client, client's memory/cache size and the client's load. The

solution to this problem is to avoid modelling any of the three factors but to model

- (a) number of RPC requests per second and
- (b) the distribution of RPC requests made by the client.

This ensures robustness as the number of RPC requests per second and the distribution of RPC requests can be varied according to the application, memory size and load on the client.

The data for (a) and (b) above can be obtained from NFS software monitors such as `nfsstat` (see Section 3.1 RPC). It displays the number and types of RPC requests made by the client from the time it is rebooted until the completion of the process. Taking two `nfsstat` outputs one hour apart will give a good idea as to the number of RPC requests generated per second and also the distribution of the requests made by a client to the server.

The client model therefore requires (a) and (b) as its input parameters. The distribution of RPC requests is then used to create a table (see example below) to model the client RPC requests. Note that only the 10 regular RPC requests are modelled.

RPC request	Cummulative Probability
getattr	0.13
setattr	0.14
lookup	0.48
readlink	0.56
read	0.78
write	0.93
create	0.95
remove	0.96
readdir	0.99
fsstat	1.0

As mentioned above, this project only models NFS from the RPC layer and below. Therefore a model of RPC software is created for the client. The reasons behind the generation of RPC requests are of no interest to us.

The codes for the client are as follows :

```

loop : if time < next_trans_time then
        hold ( next_trans_time - time)
    else begin
        hold_time = think_time.sample;
        next_trans_time := hold_time + time;
        hold ( hold_time);
    end ;

```

The above codes are needed to ensure that the generation time of requests are consistent for each simulation run. The codes for a normal situation of this type is as follows :

```

loop : hold (think_time.sample);
        if interrupted = 0 then
            generate_packet

```

```
else
    receive_packet;
repeat;
```

When the parameters for the server changes, the arrival time for the response packets also changes (the speed at which a server can serve a request affects the response time) and will cause the client to generate the next request at a different time for each simulation run. The arrival of the response packet will interrupt the client and the client will lose the time it needs to hold for the next generation of RPC request. A new time for the next generation of RPC request will need to be sampled. This will affect the simulation results and is therefore undesirable.

The codes for this model ensure that the generation time of RPC requests is affected only by the client parameters and **not** the server's as the time for the next transmission is kept in the variable *next_trans_time*. A test is conducted at the top of the loop to check if a new transmission time needs be sampled and this guarantees that the transmission time is only affected by the client parameters.

The client will be awakened when

- i) holding time is up i.e. time to generate a request
- ii) the arrival of a response from the server

and it will execute the following lines of codes :

```
if interrupted = 0 then
    generate_packet
else
    receive_packet;
```

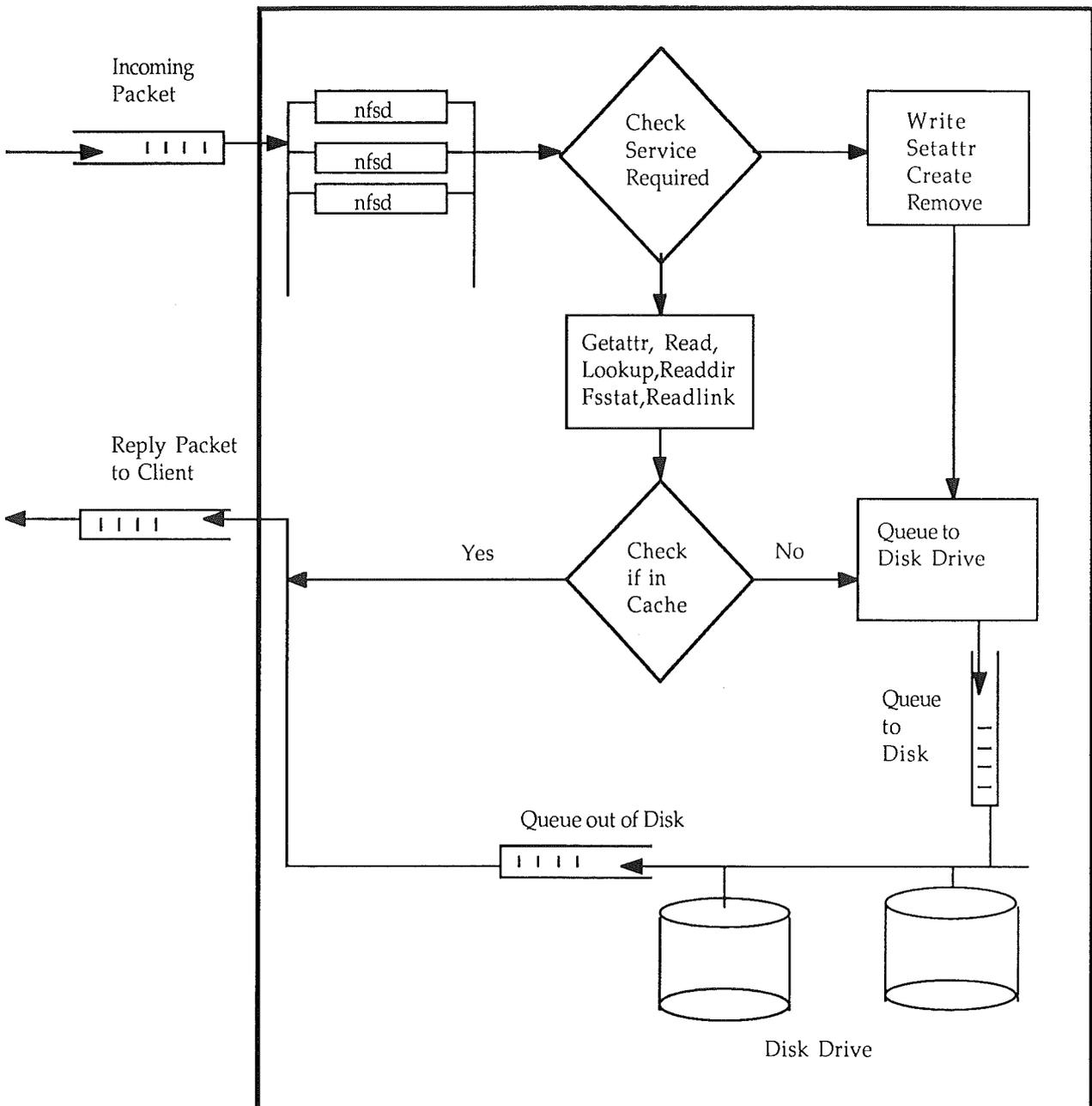
The above codes will decide if the client is to transmit a request or receive a response packet from the server.

If the client is to generate a request packet, random number generators are used to determine the types of RPC requests to be generated and also the inter-arrival time of the next requests for the client. The inter-arrival time is assumed to be **poisson** with the number of RPC requests per second used as its parameter. The types of RPC requests generated is assumed to be **uniformly distributed** based on the table of distribution created (see table above).

The arrival of a response packet will cause the client to collect the statistics of the components of the response time (see section 2). These components are then used to determine the bottleneck of the LAN.

3.7 SERVER

A file server in NFS is a computer that provides access to permanent files on a disk and other shared resources to all other workstations (clients). In the diskless client environment, the server is the only computer in the network that has a disk drive and therefore stores all the files for all the applications provided in the network. It performs functions related to the sharing of the information and hardware resources that would be performed within the operating system of a centralized computer system. It also provides an emulation of the UNIX filing system functions with remote access to the UNIX filestore mounted on different computers.



The diagram at the previous page shows workings of the server that is modelled in this project. As this project only models NFS from the RPC layer

and below, we are only interested in the actions of the server when it receives a RPC request from the client.

This project assumes a **dedicated server**, that is the server is **not** involved in any other processing work other than servicing RPC requests. Therefore there is no delay for the kernel to attend to a client request whenever there is a free NFSD process (see Section 3.3) available.

The model of file server works as follows:

1) when a packet is received by the server, it is placed in a queue into the server. The packet remains in the queue until a free nfsd process is available to attend to it.

2) the kernel will then examine the packet for the type of service required and attend to it appropriately. The actions taken by the file server for the various rpc calls are as follows:

getattr, read, readlink, readdir, lookup, fsstat -
these requests require a read from memory. It is first checked against the cache. If it is satisfied, it will be replied to immediately, if not, a disk read is required. This request is then queued to the disk drive. After the disk access, a response is then generated back to the client.

write, create, setattr, remove -
NFS uses a write-through policy. This means all write requests are not cached, but written directly to the disk drive. This request is queued to the disk drive. When the write is completed, a response packet is sent to the client.

The codes for the server are as follows :

```
procedure check_request (in_pkt); ref(packet) in_pkt;
begin
  if ( request = WRITE, SETATTR, CREATE, REMOVE) then
    queue_to_disk(in_pkt)
  else begin
    if ( request in the CACHE) then
      transmit_reply(in_pkt)
    else
      queue_to_disk(in_pkt);
    end
  end;
end;
```

The above codes checks the service requested by the client. The write requests are queued to the disk drive. The rest of the requests checks if the request received is found in the cache or if it needs to be queued for a disk access.

The main loop for the server is as follows :

```
loop :
  server_free := TRUE;
```

```

    passivate;
    server_free := FALSE;
    if ( out_of_disk_full ) then
        get_pkt_from_disk;
    if ( buffin.full and nfsd.avail > 0 ) then
        receive_request_pkt;
    repeat;

```

The server will be awakened when

- i) a disk request was successful and a response is generated to the client
- or
- ii) there is an incoming request from a client.

The codes at the previous page will decide if the server will be generating a response packet or will be receiving an incoming request packet.

The modelling of the server cache is complicated and is discussed in Section 3.2. Modelling of the disk drive is discussed in section 3.4 while modelling of NFSD processes is discussed in Section 3.3.

3.8 Input Parameters for the Model

The input for the model are shown below. The timings are all in milliseconds. For each client in the network, the distribution of the RPC requests are needed. Finally, the rate of network traffic, i.e. traffic in the network which is not generated by the clients are also needed. This can cause delays in a real network and are therefore modelled.

```

Enter simulation length : 300000
Enter number of server : 1
Enter number of nfsd for server : 8
Enter number of disk drives : 4
Enter disk read time : 30
Enter disk write time : 50
Enter cache hit rate : 0.9
Enter number of clients : 1
Enter number of transactions per second for the client 1 : 20
Enter probability for GETATTR : 8
Enter probability for SETATTR : 1
Enter probability for LOOKUP : 34
Enter probability for READLINK : 8
Enter probability for READ : 22
Enter probability for WRITE : 15
Enter probability for CREATE : 2
Enter probability for REMOVE : 1
Enter probability for REaddir : 3
Enter probability for FSSTAT : 1
Enter amount of network traffic per second : 30

```

3.9 Output of Model

HISTOGRAMS *****

SUMMARY

TITLE / (RE)SET/ OBS/ AVERAGE/EST.ST.DV/ MINIMUM/ MAXIMUM
 ResponseTime 5000.000 12029 56.094 62.488 0.125 507.516

CELL/LOWER LIM/ N/ FREQ/ CUM %

CELL/LOWER LIM/	N/	FREQ/	CUM %
0 -INFINITY	0	0.00	0.00
1 0.000	4154	0.35	34.53
2 20.000	1771	0.15	49.26
3 40.000	1795	0.15	64.18
4 60.000	1171	0.10	73.91
5 80.000	932	0.08	81.66
6 100.000	619	0.05	86.81
7 120.000	482	0.04	90.81
8 140.000	346	0.03	93.69
9 160.000	224	0.02	95.55
10 180.000	169	0.01	96.96
11 200.000	111	0.01	97.88
12 220.000	54	0.00	98.33
13 240.000	51	0.00	98.75
14 260.000	23	0.00	98.94
15 280.000	28	0.00	99.18
16 300.000	15	0.00	99.30
17 320.000	17	0.00	99.44
18 340.000	14	0.00	99.56
19 360.000	10	0.00	99.64
20 380.000	11	0.00	99.73
21 400.000	6	0.00	99.78
22 420.000	5	0.00	99.83
23 440.000	7	0.00	99.88
24 460.000	5	0.00	99.93
25 480.000	8	0.00	99.99
26 500.000	1	0.00	100.00

REST OF TABLE EMPTY

|-----

The histogram above shows the response time that is experienced by all the clients in this hypothetical network.

The three tables below shows the response time that is experienced by a **single** client. Every client in this network will have three tables of output each. The first table shows the response time of every RPC request made by that particular client. This output has the same format as nhfsstone (see model validation).

The second table shows the components of response time for client requests that hit the cache, i.e. no disk access is required.

The third table shows the components of response time of client requests that requires disk access.

PACKET STATISTICS

RPC CALLS BY THE CLIENT

Operation	Percentage	Calls	sec	msec/call	time %
getattr	12.880%	386	15.101	39.121	9.056
setattr	0.767%	23	1.904	82.775	1.142
lookup	35.302%	1058	41.810	39.518	25.074
readlink	7.207%	216	9.722	45.007	5.830
read	20.554%	616	38.774	62.945	23.253
write	15.349%	460	44.862	97.526	26.904
create	2.002%	60	5.780	96.334	3.466
remove	0.968%	29	3.266	112.607	1.958
readdir	3.604%	108	3.687	34.135	2.211
fsstat	1.368%	41	1.843	44.945	1.105

150.00 secs 2997 calls 19.980 calls/sec 55.638 msec/call

STATISTICS OF REQUESTS IN CACHE

Number of request in cache is 1931

Title	Average	Min	Max
Response Time	37.515	0.125	480.445
Server Transmission QTime	0.621	0.000	7.684
Server Receiving QTime	36.473	0.000	477.000
Client Transmission QTime	0.036	0.000	3.805
Client receiving QTime	0.000	0.000	0.000
Transmission Time	0.403	0.125	2.299

STATISTICS OF REQUESTS THAT ACCESS THE DISK

Number of disk access is 1066

Title	Average	Min	Max
Response Time	88.467	30.125	485.516
Server Transmission QTime	0.055	0.000	2.227
Server Receiving QTime	34.541	0.000	415.000
Client Transmission QTime	0.045	0.000	4.141
Client receiving QTime	0.000	0.000	0.000
Time Queuing into Disk	12.061	0.000	50.000
Time Queuing out of Disk	0.000	0.000	0.000
Transmission Time	1.054	0.125	2.688
Disk Access Time	40.732	30.000	50.000

The table below is the last table to be created by the model. It shows the components of response time for the all the requests generated by all the clients. This data is used to plot the graphs that is discussed in Section 5.

It is also used to determine the bottleneck of a LAN. The component which has the largest proportion of time spent in it is the bottleneck in the LAN.

OVERALL PACKET STATISTICS

Number of servers are 1
 Number of clients are 4
 Disk READ time is 30.000
 Disk WRITE time is 50.000

Number of collision is 3185
 Number of packets is 16434
 Total request in cache is 7596
 Total write to disk is 2315
 Total reads not in cache is 2117

Title	Average	Min	Max
Response Time	56.094	0.125	507.516
Server Transmission QTime	0.396	0.000	7.945
Server Receiving QTime	36.735	0.000	478.000
Client Transmission QTime	0.039	0.004	5.187
Client receiving QTime	0.000	0.000	0.000
Time Queuing out of Disk	0.000	0.000	0.000
Time Queuing into Disk	11.662	0.000	50.000
Transmission Time	0.242	0.016	3.895
Disk Access Time	40.449	30.000	50.000

3.10 Limitation of the Model

The limitation of the model of NFS developed include

- a) how the increase in cache size improves the hit-rate ?
- b) how the increase in load to the server affects the cahche hit-rate ?
- c) better approximation of disk read/write time ?

When the above information is known, a better model of NFS can be produced.

4 Model Validation

This project attempts to model a LAN running NFS. How realistic is this model compared to the real NFS situation ? To validate and to determine the accuracy of this model, **nhfsstone** is used.

nhfsstone - Network File System benchmark test program developed by Legato Systems, USA. It is used on an NFS client to generate artificial load with a particular mix of NFS operations (see Section 3.1 for example of traffic mix from **nfssat** output). The output of nhfsstone reports on the average response time in milliseconds per call and the load in call per second. It assumes that all NFS calls generated on the client are going to a single server, and that all of the NFS load on the server is due to this client.

The output of the model created for this project is :

RPC REQUESTS BY THE CLIENT

Operation	Percentage	Calls	sec	msec/call	time %
getattr	13.106%	649	4.147	6.390	1.799
setattr	0.949%	47	4.581	97.477	1.988
lookup	34.431%	1705	42.890	25.155	18.608
readlink	7.674%	380	10.073	26.508	4.370
read	20.981%	1039	61.903	59.580	26.856
write	15.327%	759	85.825	113.076	37.235
create	1.898%	94	10.186	108.359	4.419
remove	0.868%	43	5.061	117.689	2.196
readdir	3.554%	176	5.127	29.129	2.224
fsstat	1.212%	60	0.705	11.750	0.306

165.00 secs 4952 calls 30.012 calls/sec 46.546 msec/call

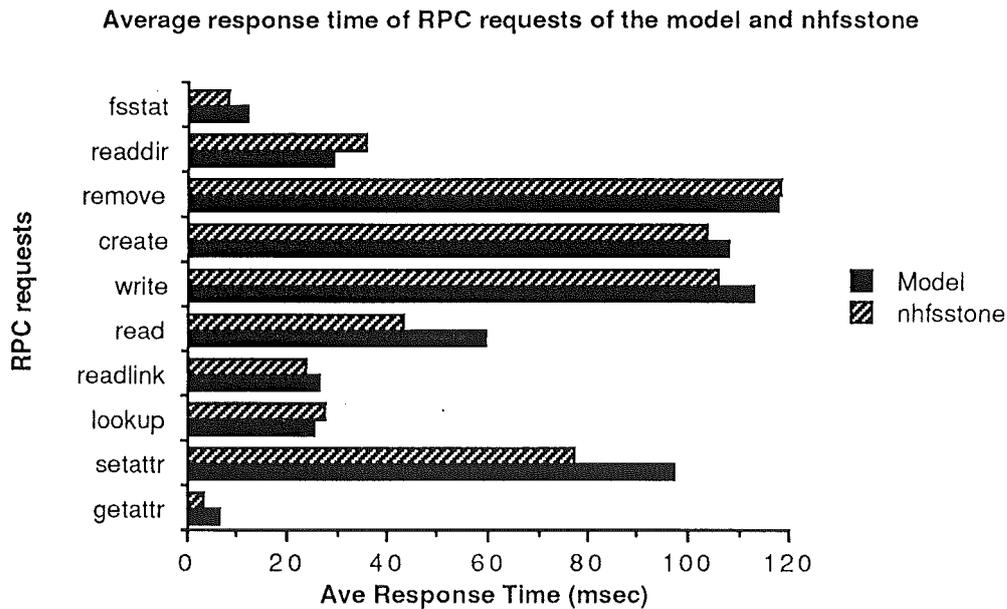
The output of nhfsstone is as follows :

```
./nhfsstone -v ./huiadir*
op          want      got      calls    secs    msec/call  time %
null        0%        0.00%    0         0.00    0.00       0.00%
getattr     13%       13.95%   716       2.29    3.20       1.04%
setattr     1%        1.05%    54         4.18    77.50      1.90%
root        0%        0.00%    0         0.00    0.00       0.00%
lookup      34%       32.44%  1664      45.45   27.31      20.72%
readlink    8%        7.56%    388       9.19    23.69      4.19%
read        22%       22.12%  1135     49.07   43.23      22.37%
wrcache     0%        0.00%    0         0.00    0.00       0.00%
write       15%       15.81%   811       86.12  106.19     39.26%
create      2%        2.04%   105       10.87  103.60     4.95%
remove      1%        1.03%    53         6.26   118.20     2.85%
rename      0%        0.00%    0         0.00    0.00       0.00%
link        0%        0.00%    0         0.00    0.00       0.00%
symlink     0%        0.00%    0         0.00    0.00       0.00%
mkdir       0%        0.00%    0         0.00    0.00       0.00%
rmdir       0%        0.00%    0         0.00    0.00       0.00%
readdir     3%        2.98%   153       5.47   35.79      2.49%
fsstat      1%        0.97%    50         0.39    7.94       0.18%
167 sec 5129 calls 30.71 calls/sec 42.76 msec/call
```

The above result is obtained running nhfsstone in the middle of the night when there is minimum load on both the server and network (i.e. there are no users on the network). This is to ensure that the results of nhfsstone shows the actual response time of the server, and is not influenced by any other factors.

The model is fine tuned to achieve the desired response time which is comparable to the output of nhfsstone.

The bar graph below shows the response time experienced by the RPC requests of both nhfsstone run and the output of the model. The data is obtained from the output at the previous page.



Comparing the output of the model and nhfsstone shows that the model is able to produce results similar to that of nhfsstone. However few simulation run of the model may be needed to achieve result.

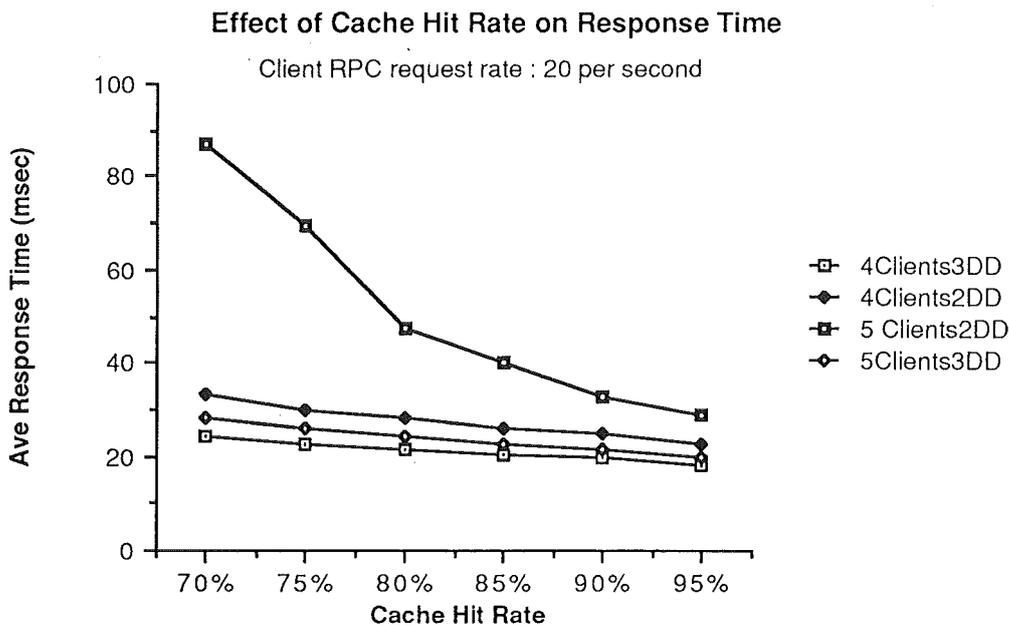
5 Simulation Results Of Interest

All the simulation results below are for ten minutes of simulation time. The bottleneck of the LAN during the peak ten minutes of the day is of interest in this experiment. No statistical test is carried out on the simulation results as this project is only interested in determining the bottleneck during the peak period in the network.

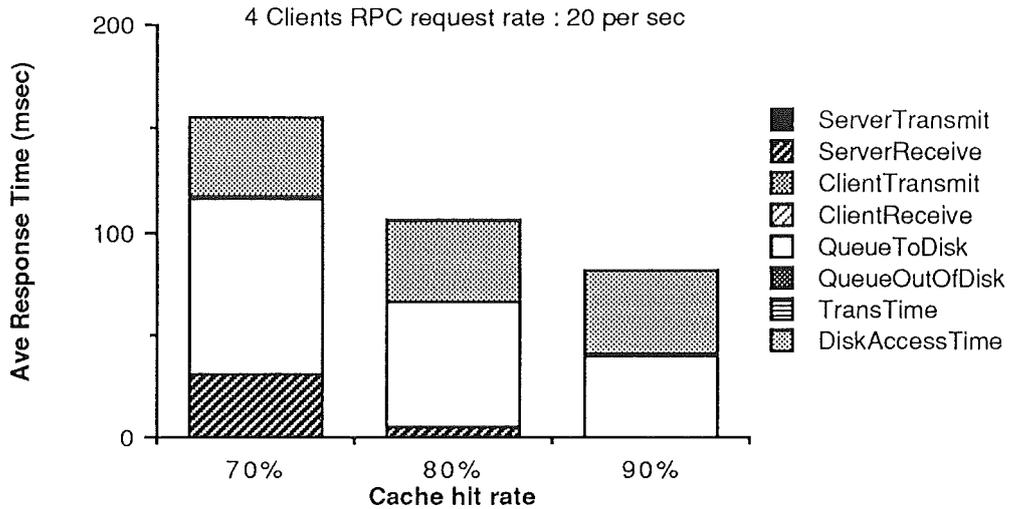
5.1 Server Cache Size

Increasing the size of the server cache will improve the cache hit-rate. However the exact effect of varying server cache size to the cache hit-rate is not known, e.g. increasing cache size from 2 mb to 4 mb will increase cache hit-rate but the exact percentage of improvement is unknown. It is therefore not possible to tell what the improvement of cache hit-rate from 70% to 80% corresponds to the actual increase in the size of server cache.

However, the graph below shows that increasing the cache hit-rate (therefore increasing the server cache size) will improve the response time especially when the network has lots of clients and server has few disk drives.



Components of Response Time as Cache Hit Rate varies



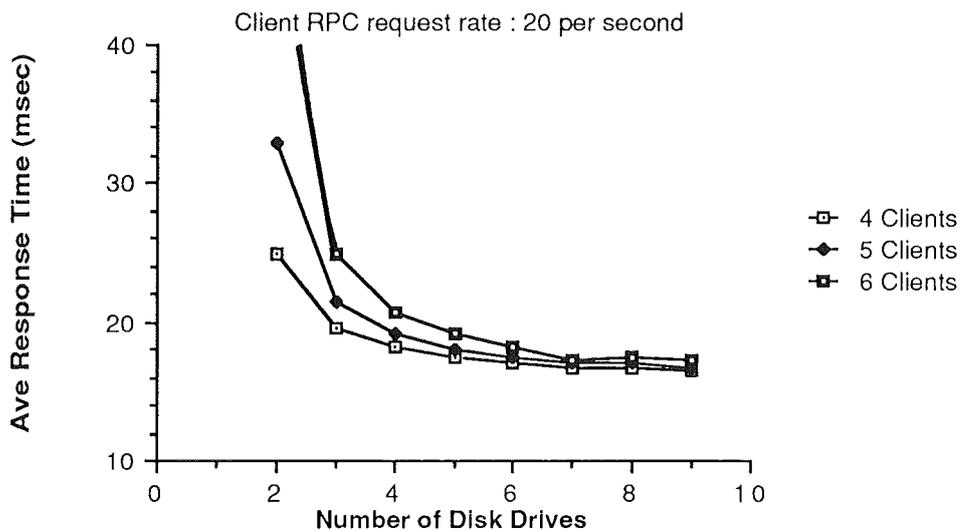
In terms of the components of response time, increasing the cache size decreases the wait time for the server to receive a packet and also the queue time to the disk drive. This is because more read request are satisfied by the cache and would not need a disk access. This frees the nfsd process more quickly and therefore reduces the wait time for other packets in the queue and reduces queue time to disk as less read disk access is needed.

However increasing server cache will **not** improve write request time as NFS follows a write-through policy.

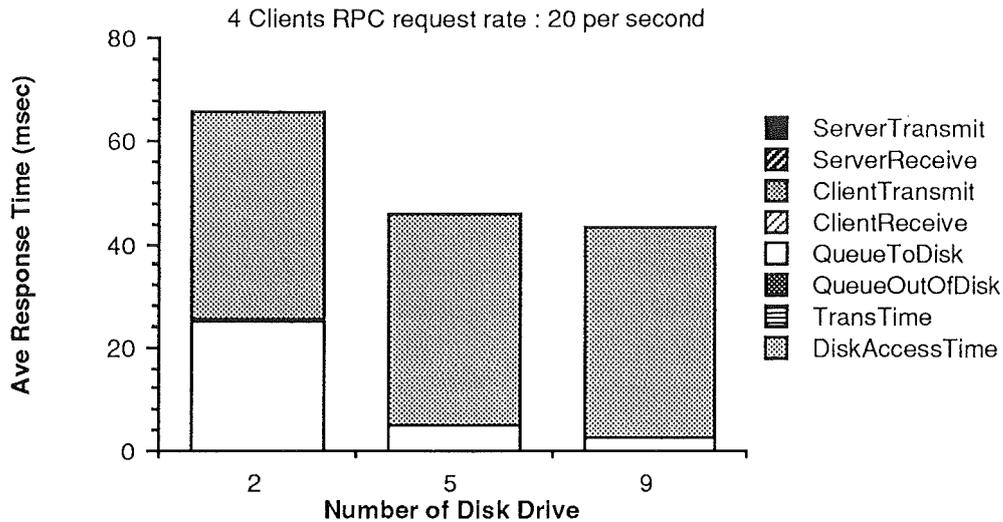
5.2 Number of Disk Drives

From the graph below, increasing the number of disk drives in a server does improve the response time significantly, especially when the server is servicing many of clients. The rate of improvement decreases as the number of disk drive increases. This is because the time taken to access the disk drive will remain the most significant factor in determining the response time.

Effect of Disk Drive on Response Time



Component of Response as number of disk drive varies



In terms of the components of response time, increasing the number of disk drives reduces the queue time to the disk. This is because the queue to the disk is spread equally among all the disks in the server.

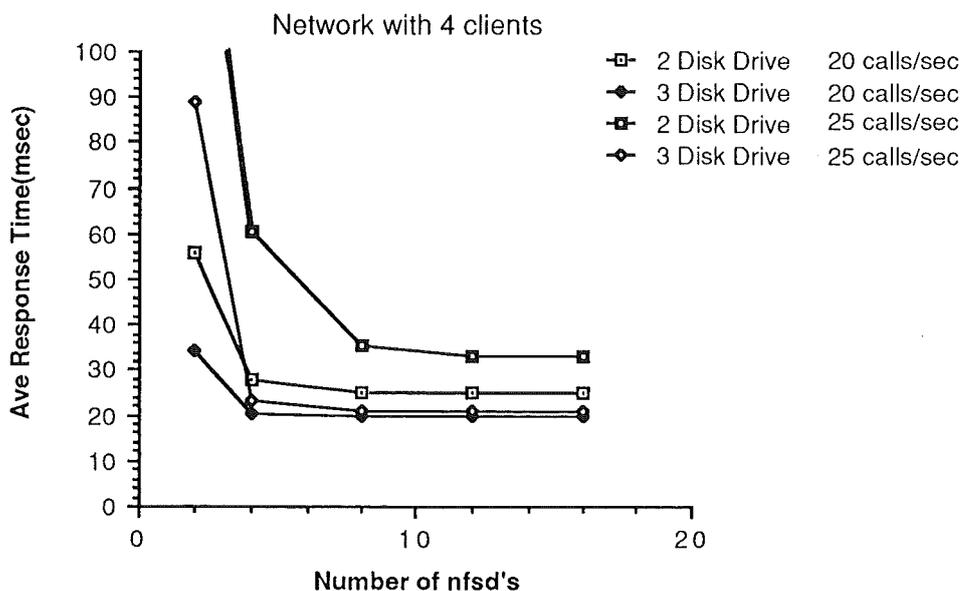
This solution to improve the response time is only possible if all the disk drives are equally likely to be accessed.

5.3 Number of NFSD Processes

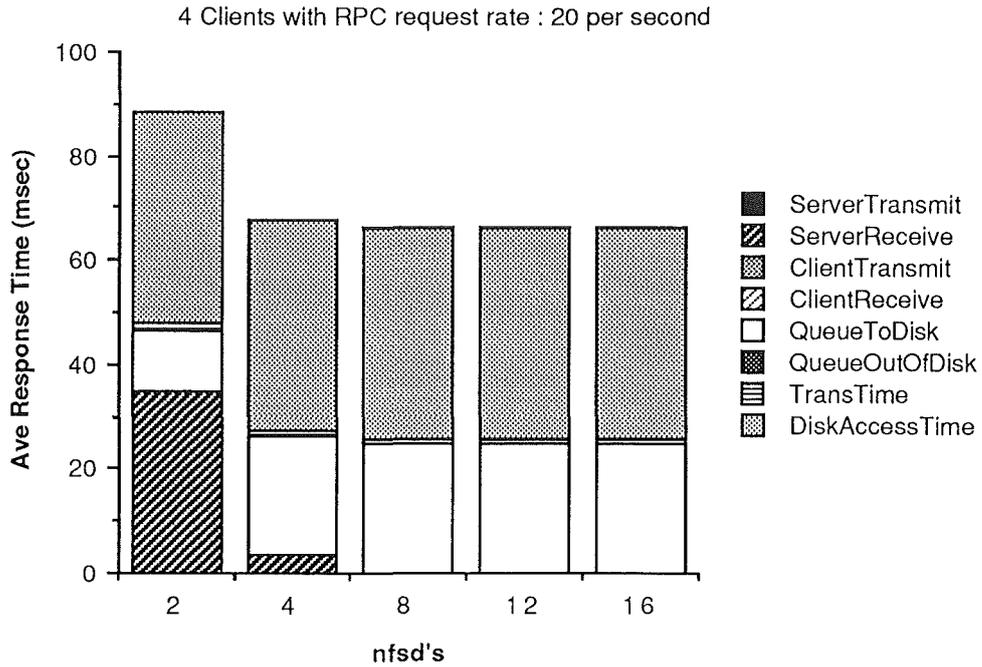
Increasing the number of NFSD processes improves the response time. The rate of improvement is visible in a server which has a small number of disk drives and experiencing a heavy load (in terms of the rate of RPC request per second).

The improvement reaches a maximum at about 8 NFSD processes. It improves the response time because the server is able to service more requests at any one time.

Effect of the number of NFSD on Response Time



Components of Response Time as the number of NFSD varies



Increasing the number of NFSD processes reduces the waiting time to the server. With more NFSD processes, the server is able to serve more requests at once. However it also increases the queue time to the disk as the server is able to handle more disk access requests at any one time.

This solution is only feasible when there is a mixture of read and write requests to the server. Too many disk access requests will increase the queue time to disk but decrease the queue time to the server.

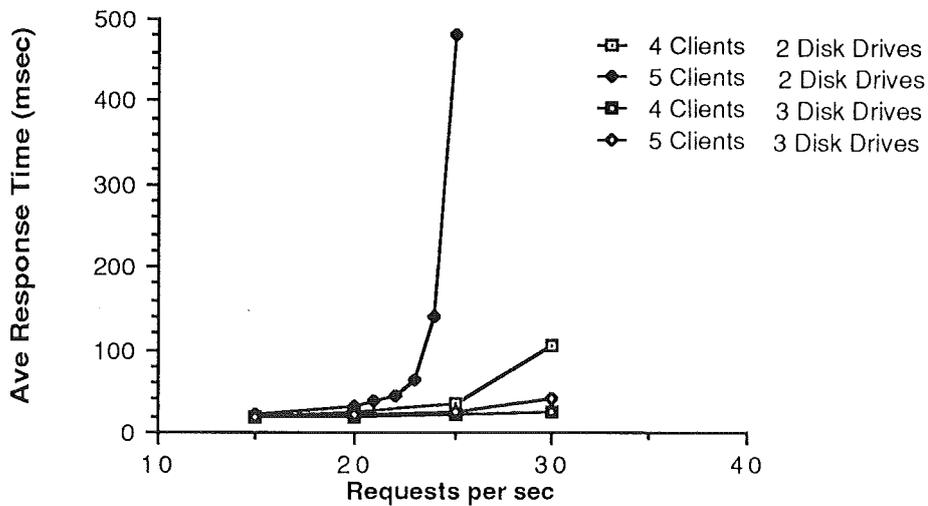
5.4 Load on the Server

The load on the server in this model is defined to be the number of RPC requests made by the clients per second.

As the load on the server increases, the cache hit-rate will decrease at a rate which is unknown at this stage as there is no study on this is available. The graph below therefore does not show a very accurate representation on NFS in real life but the effect experienced by the network will be approximately similar to the one shown in the graph.

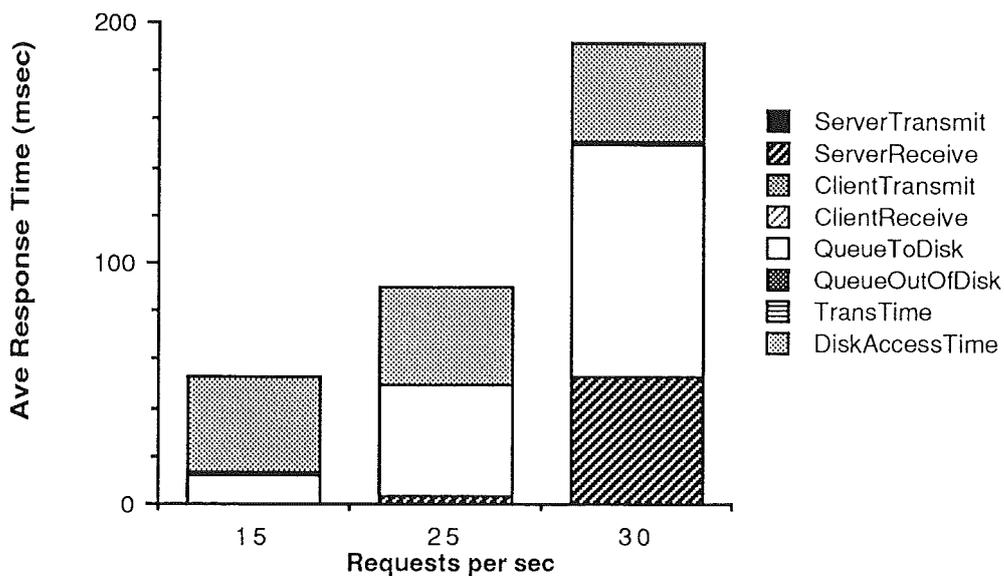
The graph below shows that as the load on the server increases, the response time also increases. The increase is more significant in a network that has more clients and less disk drives.

Effect on Response Time as Load Increases



Component of Response Time as Load increases

Network with 4 Clients

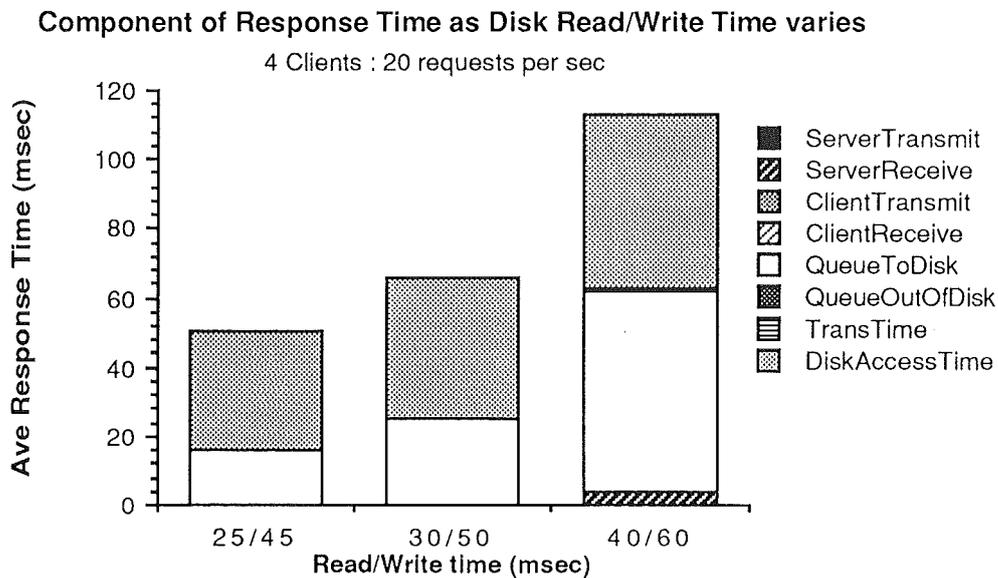
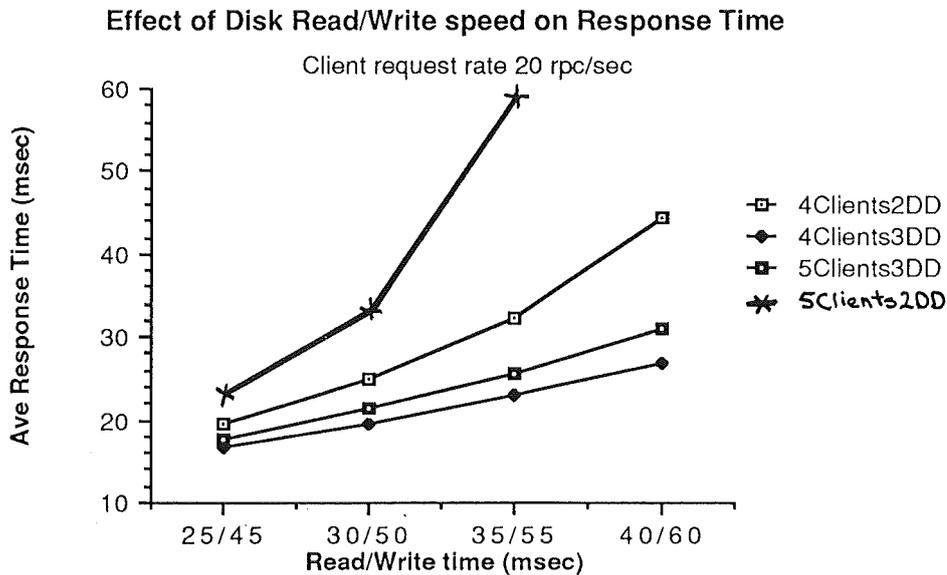


In terms of the components of response time, the increase in the rate of RPC requests by the clients will increase both the queue time to the server and queue time to the disk drive.

5.5 Disk Access Speed

The graph below shows the increase in response time as the disk read/write time is increased. It has the same effect as a slower disk drive.

Slow disk drives do not have much effect on a network with a small number of clients and a server that has lots of disks. However, if the number of clients on the network increases, the increase in response time will be tremendous.



The components of response time shows that as the disk access time increases, the queue time to the disk drive increases. The queue time to the server also increases as the number of NFSD processes freed by the server is decreased as more packets are being queued at the disk drive.

Improving on the disk access time is the most difficult as disk latency time and disk seek time improves at a slower rate compared to the processor speed and memory speed.

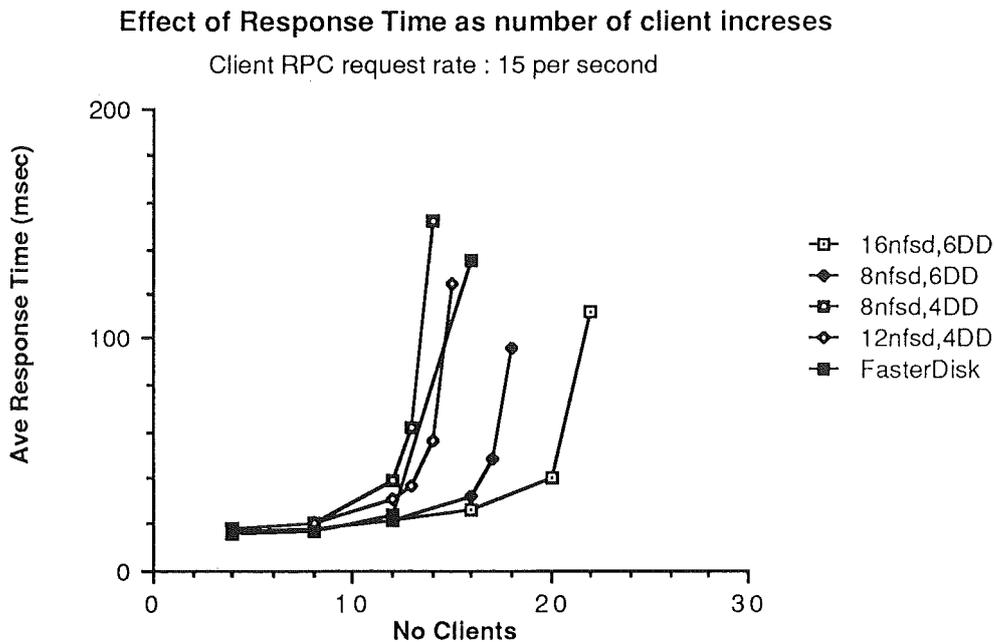
6 Case Study

In this case study, a hypothetical network is created and modelled and the response time for various network configuration is plotted in the graph below. It is similar to the configuration of the NFS network in the Department of Computer Science at the University of Canterbury.

Initially the server has eight NFSD processes and four disk drives. A network administrator would like to know the effect of increasing the number of NFSD processes to twelve. From the graph below, increasing the number of NFSD processes does improve the response time, but not very significantly.

Getting faster disk drives will also improve the response time but again not to a very significant amount. Faster disk drive means that the disk access time for a read or write request takes less time than the older one.

However increasing the number of disk drives to six shows a tremendous improvement to the response time, as shown in the graph below. As the number of disk drives is increased, the number of NFSD processes should also be increased as it helps to improve the response time.



Another way of using the above graph is to determine the number of clients that a server with a particular configuration can support. The network administrator will first have to determine an acceptable average response time that users will be able to tolerate. A horizontal line is then drawn on the graph intersecting the y-axis at the acceptable average response time. From there, the number of clients that a server is able to service is known.

7 CONCLUSION

From the results of the experiments carried out, it has been observed that the bottlenecks of the LAN are the queue of request packets to the server and the queue to the disk drive and not Ethernet.

These matters can be solved as follows :

i) increasing the size of the server cache. (Section 5.1) This increases the hit-rate of the client requests and therefore reduces the queue to the disk drive. This solution also reduces the queue time to the server as more requests are able to be serviced faster (i.e without accessing the disk) therefore freeing NFSD processes faster and these freed NFSD processes are able to service other requests. However this solution is not feasible if most of the client requests are write requests. This is because NFS follows a write-through policy which requires all write requests to be written directly to the disk.

ii) increasing the number of disk drives. (Section 5.2) This option decreases the queue time to the disk drive. The idea behind this solution is to spread the queue to the disk drive among all the disks. It only works if all the disk drives are equally likely to be accessed by the client requests but this cannot be controlled by anyone.

iii) increasing the number of NFSD processes. (Section 5.3) This decreases response time by allowing the server to serve more requests at any one time. However it also increases the queue time to the disk drive as the server is able to serve more write requests at the same time.

iv) getting a faster disk drive. (Section 5.5) It reduces the time taken to read from the disk or write to the disk and will therefore decrease the queue time to the disk drive and the queue time to the server.

The simulation model of NFS created for this project can be used by a network administrator to experiment with the network configurations and observe the changes in response time that a user will experience as a result of that modification without making the changes to the actual network.

The model created can be used to answer the following questions :

- a) where is the bottleneck ?
- b) how many clients can the server support ?
- c) how many NFSD processes are worthwhile ?
- d) how much improvement would an extra disk drive give ?
- e) would a faster disk drive help ?

These results were shown in Section 6 where a case study was conducted on the network at the Computer Science Department of The University of Canterbury to answer the questions above.

As mentioned, the model created has its limitations especially on the modelling of the cache. The decrease in the server cache hit-rate as the load increases is also not known.

The following improvements could be made to the model:

- a) modelling of the server cache
- b) effect of cache hit-rate as the load on the server increases
- c) a better approximation for the disk read and write time

A better model of NFS would also allow the server cpu process to be represented as this will influence the response time tremendously.

APPENDIX 1
REFERENCES

REFERENCES

- [Agarwal, Hennessy, Horowitz 88]
Anant Agarwal, John Hennessy, and Mark Horowitz
Cache Performance of Operating Systems and
Multiprogramming Workload
ACM Transactions on Computer Systems,
Vol 6, No 4, Nov 1988 Pages 393-431
- [Birtwistle]
Graham M. Birtwistle
Discrete Event Modelling On Simula
Macmillan Computer Science Series 1979
- [Coulouris, Dollimore]
George F. Coulouris and Jean Dollimore
Distributed Systems Concepts and Design
Addison Wesley, International Computer Science Series 1988
- [Dupre-Blusseau 89] Floriane Dupre-Blusseau
Modelling the NFS service on an Ethetnet LAN
Proceedings of the Autumn 1989 EUUG Conference
Pages 99-110
- [Gee]
K.C.E. Gee
Introduction to LAN Computer Network
Macmillan Press London 1983
- [Lazowska, Zahorjan 1986]
Edward D. Lazowska and John Zahorjan
File Access Performance of Diskless Workstations
ACM Transactions on Computer Systems,
Vol 4, No 3, Nov 1986 Pages 238-268
- [Lyon, Sandberg 1989]
Bob Lyon, Russel Sandberg
Breaking Through the NFS Performance Barrier
SunTech Journal Autumn 1989, Pages 21-27
- [Malamud 1990]
Carl Malamud
Sharing the Wealth : RPCs Help Programs Go Places
Data Communications June 1990, Pages 61-70
- [Marriot P 1986]
Paul Marriott
NETMOS - A Local Area Network Simulation Context
Honours Project, Dept of Computer Science,
University of Canterbury, Christchurch, NZ.
- [Nelson, Welch, Ousterhout]
Michael N. Nelson, Brent B. Welch and
John K. Ousterhout
Caching in the Sprite Network File System
ACM Transactions on Computer Systems
Vol 6, No 1, February 1988, Pages 134-154

- [Nichols D.A. 1990] David A. Nichols
Multiprocessing in a Network of Workstations
Phd Thesis, School of Computer Science,
Carnegie Mellon University, CMU-CS-90-107
- [Smith 1985] Alan Jay Smith
Disk Cache -Miss Ration Analysis and Design
Considerations
ACM Transactions on Computer Systems
Vol 3, No 3, August 1985, Pages 161-203
- [Smith 1985] Alan Jay Smith
Cache Evaluation and Impact of Workload Choice
Proc of the 12th IEEE Conference on Computer
Architecture
1985, Pages 64 - 73
- [Srinivasan, Mogul 89] V. Srinivasan, Jeffery C. Mogul
Spritely NFS: Experiments with Cache-Consistency
Protocols
Proceedings of the 12th ACM Symposium on
Operating Systems Principles, Dec 1989, Pages 45-57

APPENDIX 2
PROGRAM LISTING

```
external class NETMOS = "/home/undergrad/honours/wah/PROJECT/NETSIM/net";
```

```
netmos begin
```

```
    ref(histogram) hist;
    ref(user) array users(1:50);
    ref(overall_packet_stat) overall_pkt_stat;
    ref(tally) o_response_time,o_into_disk_queue, o_out_disk_queue;
    ref(tally) o_ServerRecQwait, o_ClientRecQwait, o_disk_time,
        disk_access_response_time;
    integer i, num_clients,num_server, j, total_station;
    real length, read_time, write_time, IP_processing_time;
    integer TRAFFIC,GETATTR,SETATTR,LOOKUP,READLINK,READ,WRITE,CREATE,
        REMOVE,REaddir,FSSTAT;
    integer pkt_counter, write_count, in_cache_count,rest_count,num_trans;
```

```
COMMENT ----- NETWORK USER -----;
COMMENT
```

```
* Serves as the basic class for the network users i.e. the clients and servers.
* It's super class "USER" provides the interface to the network when
* transmission or reception of packets occurs.
```

```
*
```

```
* ;
```

```
    user class network_user;
    begin
```

```
        end;
```

```
COMMENT ----- DISK DRIVE-----;
COMMENT
```

```
* Simulates the disk drive for the NFS file server. It processes the incoming
* packet according to the service it requests.
* The request it accepts are
* WRITE, SETATTR, REMOVE, CREATE : each of these request is written to the
* disk ( NOT to memory ), NFS follows a write-through policy.
* OTHERS : the other request that reaches here has been checked by the file
* server,it means that the read request is not in CACHE, therefore needs
* to be obtained from the disk drive.
```

```
*
```

```
* Variables :
```

```
* in_pkt - reference to the incoming packet received by the disk drive.
* in_q,
* out_q - queue in and out of the disk_drive
* disk_q - condition queue to signal the disk drive of an incoming packet
```

```
* ;
```

```
    entity class disk_drive(net_server);
        ref(server) net_server;
```

```
    begin
```

```
        ref(packet) in_pkt;
        ref(disk_drive_q) in_q, out_q;
        ref(condq) disk_q;
```

```
    loop :
```

```
        disk_q.waituntil (in_q.full);
        in_pkt := in_q.takeout;
        if (in_pkt.service = WRITE) or (in_pkt.service = REMOVE) or
            (in_pkt.service = SETATTR)or (in_pkt.service = CREATE) then begin
            hold(write_time);
```

```

        o_disk_time.update(write_time);
        in_pkt.disk_time := write_time;
    end
else begin
    hold(read_time);
    o_disk_time.update(read_time);
    in_pkt.disk_time := read_time;
end;
in_pkt.dest := in_pkt.source;
in_pkt.disk_access := true;
out_q.putin(in_pkt);
net_server.interrupt(0);
repeat;

end *** DISK DRIVE ***;

COMMENT ----- SERVER -----;
COMMENT
* Server acts as the file server in the network. receives packets from
* clients and replies to the client request.
* Eg : every WRITE request received will be sent directly to the
*       disk drive.
*       every READ request will be checked against the CACHE first to
*       improve the access speed. If NOT found, the request will be
*       sent to the disk drive for access.
*
* Variables :
* server_free - boolean variable indicating the server state.
* pkt,
* in_pkt - reference to the packets that the server receives
*          and generates.
* in_cache - the probability of hitting the cache by a request
* read_hit_rate - the probability of hitting the cache by a READ request
* getattr_hit_rate - the hit-rate for the getattr request
* fsstat_hit_rate - the hit-rate for the fsstat request
* disk_unit - the disk drive that belongs to this server
* nfsd       - the number of daemons that this server has to handle
*              client RPC request
* ;

network_user class server;
begin
    ref(bdist) in_cache, read_hit_rate,getattr_hit_rate, fsstat_hit_rate;
    ref(idist) disk_number;
    ref(disk_drive) array disk_unit(1:10);
    ref(packet) pkt, in_pkt;
    ref(condq) disk;
    ref(res) nfsd;
    integer disk_drive_per_server, num_nfsd;
    real cache_hit_rate, getattr_hit,read_hit;
    boolean server_free;

    procedure get_user_input;
    begin
        outf.outtext("Enter number of nfsd for server ");outf.outimage;
        num_nfsd := inint;
        outf.outint(num_nfsd,3);outf.outimage;
        outf.outtext("Enter number of disk drives ");outf.outimage;
        disk_drive_per_server := inint;
        outf.outint(disk_drive_per_server,3);outf.outimage;
        outf.outtext("Enter disk read time : ");outf.outimage;
        read_time := inreal;

```

```

    outf.outfix(read_time,2,6);outf.outimage;
    outf.outtext("Enter disk write time : ");outf.outimage;
    write_time := inreal;
    outf.outfix(write_time,2,6);outf.outimage;
    getattr_hit := 0.97;
    outf.outtext("Enter cache hit rate ");outf.outimage;
    cache_hit_rate := inreal;
    outf.outfix(cache_hit_rate,3,5);outf.outimage;
    read_hit := cache_hit_rate * 0.4;

end;

procedure initialize;
begin
    integer i;

    disk_number := new randint("DiskNum",1,disk_drive_per_server);
    in_cache := new draw("Cache",cache_hit_rate);
    getattr_hit_rate := new draw("GetattrHit",getattr_hit);
    read_hit_rate := new draw("ReadhitRate",read_hit);
    fsstat_hit_rate := new draw("FsstatHit",getattr_hit * 0.95);
    nfsd := new res ("NFSD",num_nfsd);
    hold(0.05);
    for i:= 1 step 1 until disk_drive_per_server do begin
        disk_unit(i) := new disk_drive(edit("DiskDrive",my_id),
            this server);
        disk_unit(i).schedule(0.0);
        disk_unit(i).in_q := new disk_drive_q("InQ",
            this network_user,true);
        disk_unit(i).out_q := new disk_drive_q("OutQ",
            this network_user,false);
        disk_unit(i).disk_q := new condq("DiskQ");
    end;

end;

procedure queue_to_disk(in_pkt);
    ref(packet) in_pkt;
COMMENT: queues a request packet to the disk;
begin
    integer which_disk;

    which_disk := disk_number.sample;
    disk_unit(which_disk).in_q.putin(in_pkt);
    disk_unit(which_disk).disk_q.signal;
end;

boolean procedure out_of_disk_full;
COMMENT: checks if there is any request that finished accessing
the disk;
begin
    integer i;

    out_of_disk_full := false;
    for i := 1 step 1 until disk_drive_per_server do
        if disk_unit(i).out_q.full then
            out_of_disk_full := true;
    end;

    procedure transmit_reply(out_pkt);
        ref(packet) out_pkt;
begin

```

```

        out_pkt.size := reply_packet_size(out_pkt.service);
        buffout.putin(out_pkt);
        nfsd.release(1);
end;

procedure get_pkt_from_disk;
begin
    integer i;
    ref(packet) pkt;

    for i := 1 step 1 until disk_drive_per_server do
        if disk_unit(i).out_q.full then begin
            pkt := disk_unit(i).out_q.takeout;
            transmit_reply(pkt);
        end;
    end;
end;

procedure receive_request_pkt;
begin
    while (buffin.full and (nfsd.avail > 0)) do begin
        pkt_counter := pkt_counter + 1;
        nfsd.acquire(1);
        in_pkt := buffin.takeout;
        in_pkt.server_rec_qtime := time - in_pkt.timein +
            IP_processing_time;
        o_ServerRecQwait.update(time - in_pkt.timein +
            IP_processing_time);
        if in_pkt.service <> TRAFFIC then
            check_request(in_pkt)
        else begin
            in_pkt := none;
            nfsd.release(1);
        end;
    end;
end;

procedure check_request(in_pkt);
ref(packet) in_pkt;
begin

    switch case := GETATTR, SETATTR, LOOKUP, READLINK, READ, WRITE,
        CREATE, REMOVE, READDIR, FSSTAT;

        goto case(in_pkt.service);
WRITE : SETATTR : CREATE : REMOVE :
    write_count := write_count + 1;
    queue_to_disk(in_pkt);
    goto stop;
READ : if read_hit_rate.sample then
    goto hit_the_cache
    else
    goto disk_access;
LOOKUP : READDIR : READLINK :
    if in_cache.sample then
        goto hit_the_cache
    else
        goto disk_access;

GETATTR : if getattr_hit_rate.sample then
    goto hit_the_cache
    else
        goto disk_access;

```

```

FSSTAT : if fsstat_hit_rate.sample then
    goto hit_the_cache
else
    goto disk_access;

hit_the_cache : in_cache_count := in_cache_count + 1;
    in_pkt.dest := in_pkt.source;
    transmit_reply(in_pkt);
    goto stop;

disk_access : rest_count := rest_count + 1;
    queue_to_disk(in_pkt);
    goto stop;

stop :
end;

get_user_input;
initialize;

loop:
    server_free := true;
passivate;
    server_free := false;
    if out_of_disk_full then
        get_pkt_from_disk;
    if (buffin.full and nfsd.avail > 0 ) then
        receive_request_pkt;
    repeat;

        end *** SERVER ***;

COMMENT ----- CLIENT -----;
COMMENT
* Acts as the client in the network. It generates requests to the server.
* The request it generates are
* WRITE,SETATTR,CREATE,REMOVE : writes to the disk
* READ,READLINK,GETATTR,FSSTAT,LOOKUP,REaddir : reads a page in the server
* cache or if it not in the cache, the request require access from
* the disk.
* Variables :
* pkt - reference to the packet that it generates to the server
* in_pkt - reference to the packet that it receives from the server
* rpc_request - the probability distribution of the type of rpc requests
* this client calls
* next_service - get the next request type required by the client
* which_server - get the address of the server which the next request
* requires
* packet_stat_collector - collects the statistics of the requests generated b
* by each client individually.
* next_trans_time - the time for the generation of the next request packet.
* This is needed to ensure that changes in server parameter
* do not influence the generation time of client requests.
* hold_time - the amount of time to "sleep" while waiting for the next event
* ;

network_user class client;
begin
    ref(packet)pkt, in_pkt;
    ref(client_rpc_request) rpc_request;
    ref(pkt_stat_collector) packet_stat_collector;
    ref(rdist)next_service;
    ref(idist) which_server;

```

```

    real next_trans_time, hold_time;

    procedure initialize;
    begin
        next_service := new uniform(edit("Request",my_id),0,1);
        which_server := new randint("WhichServer",1,num_server);
        rpc_request := new client_rpc_request(my_id);
        packet_stat_collector := new pkt_stat_collector("Pkt Stat");
    end;

    procedure generate_pkt;
    begin

        pkt := new packet(this client);
        pkt.dest := which_server.sample;
        pkt.service := rpc_request.get_service(next_service.sample);
        pkt.size := rpc_request.get_packet_size(pkt.service);
        buffout.putin(pkt);
    end;

    procedure receive_pkt;
    begin
        real res_time;

        while interrupted <> 0 do begin
            interrupted := 0;
            pkt := buffin.takeout;
            if pkt.service <> TRAFFIC then begin
                res_time := time - pkt.send_time;
                pkt.client_rec_qtime := time - pkt.timein
                    + IP_processing_time;
                o_response_time.update(res_time);
                packet_stat_collector.update(pkt.service,res_time,pkt);
                o_ClientRecQwait.update(time - pkt.timein
                    + IP_processing_time);
            end;
            pkt := none;
        end;
    end *** receive packet ***;

    initialize;

    hold_time := thinktime.sample;
    next_trans_time := hold_time + time;
loop :
    if time < next_trans_time then
        hold (next_trans_time - time)
    else begin
        hold_time := thinktime.sample;
        next_trans_time := hold_time + time;
        hold(hold_time);
    end;
    if interrupted = 0 then begin
        generate_pkt;
    end
    else begin
        receive_pkt;
    end;
    repeat;
end;

```

COMMENT ----- TRAFFIC ----- ;

```

COMMENT
* Generates outsize traffic to the network to any of the clients or server in
* the network The purpose of it is just to create transmission delays in the
* network.
* ;
  network_user class traffic_generator;
begin

  procedure receive_pkt;
begin
  ref(packet) in_pkt;

  interrupted := 0;
  in_pkt := buffin.takeout;
  in_pkt := none;
end;

  procedure generate_traffic;
begin
  ref(packet) pkt;

  pkt := new packet(this traffic_generator);
  pkt.service := TRAFFIC;
  buffout.putin(pkt);
end;

  loop :
    hold(thinktime.sample);
    if interrupted = 0 then
      generate_traffic
    else
      receive_pkt;
    repeat;
end;

```

```

COMMENT ----- DISK DRIVE QUEUE -----;

```

```

COMMENT

```

```

* Queue used by the server to queue the packets that requires disk access.

```

```

*

```

```

* Variables :

```

```

* t current time

```

```

* head

```

```

* tail reference to the packets in the queue

```

```

* avail number currently waiting in the queue

```

```

* zeroes number of times queue is empty.

```

```

* ;

```

```

queue class disk_drive_q(u, into_disk);

```

```

  ref(network_user) u;

```

```

  boolean into_disk;

```

```

begin

```

```

  ref(packet) head, tail;

```

```

  ref(tally) queue_time;

```

```

  integer avail, zeroes;

```

```

  real t;

```

```

  procedure report;

```

```

begin

```

```

  real span;

```

```

  t := time;

```

```

span := t - resetat;
writetrn;
  outf.image.sub(24,7).putint(obs);
  outf.outint(maxlength,6);
  outf.outint(avail,6);
  outf.setpos(44);
if span < epsilon then
  outf.outtext(minuses.sub(1,10))
else
  printreal((qint + (t - lastqtime)*avail)/span);
  outf.outint(zeroes, 6);
  outf.outtext(" ");
  if obs > 0 then
    printreal ( cum/obs)
  else
    outf.outtext(minuses.sub(1,10));
  outf.outimage;

  end *** report ***;

procedure reset;
begin
  obs := zeroes := 0;
  maxlength := avail;
  lastqtime := resetat := time;
  qint := cum := 0.0;

  end *** reset ***;

boolean procedure full;
begin

  if avail > 0 then
    full := true
  else
    full := false;
  end;

  procedure putin(pkt);
  ref(packet) pkt;
begin

  if avail = 0 then
    head :- tail :- pkt
  else begin
    tail.link :- pkt;
    tail :- pkt;
  end;
  pkt.time_in_queue := t := time;
  qint := qint + ( t - lastqtime) * avail;
  lastqtime := t;
  if avail = 0 then
    zeroes := zeroes + 1;
    avail := avail + 1;
  if avail > maxlength then
    maxlength := avail;
  end *** putin ***;

  ref(packet) procedure takeout;
begin
  ref(packet) pkt;

```

```

real qtime;

pkt :- head;
if head == tail then begin
  head :- none;
  tail :- none;
end
else
  head :- head.link;
  avail := avail - 1;
  qtime := time - pkt.time_in_queue;
  if into_disk then begin
    o_into_disk_queue.update(time - pkt.time_in_queue);
    pkt.disk_in_qtime:= time - pkt.time_in_queue;
  end
else begin
  o_out_disk_queue.update(time - pkt.time_in_queue);
  pkt.disk_out_qtime := time - pkt.time_in_queue;
end;
t := time;
qint := qint + ( t - lastqtime) * avail;
obs := obs + 1;
lastqtime := t;
cum := cum + t;
takeout :- pkt;
end *** TAKE OUT ***;

head :- tail :- none;
avail := maxlength := 0;
zeroes := obs := 0;
qint := cum := 0.0;
join(serverqq);
end *** DISK DRIVE QUEUE ***;

```

```

COMMENT ----- PACKET -----;
COMMENT

```

```

* Used by both clients and servers as the means of communication.

```

```

*

```

```

* Variables :

```

```

* send_time time when this packet is sent.

```

```

* Used to obtain the response time.

```

```

* service the type of service requested by the client to the server

```

```

* and the response from the server.

```

```

* time_in_queue used to obtain the spent by packet in the queue to the

```

```

* disk drive.

```

```

* disk_time the amount of time the request spent at the server disk.

```

```

* server_rec_time the amount of time the packet spent in the queue before

```

```

* the server receives it.

```

```

* client_rec_time the amount of time the packet spent in the queue before

```

```

* the client receives it.

```

```

* disk_in_qtime time the packet spent queueing into the disk drive.

```

```

* disk_out_qtime time the packet spent queueing out of the disk drive.

```

```

* disk_acces to indicate if the packet accessed the disk.

```

```

* ;

```

```

message class packet;

```

```

begin

```

```

  integer service;

```

```

  real send_time, time_in_queue,disk_time;

```

```

  real server_rec_qtime, client_rec_qtime,

```

```

    disk_in_qtime, disk_out_qtime;

```

```

  boolean disk_access;

```

```

        disk_access := false;
        send_time := time;

end *** PACKET ***;

COMMENT ----- REPLY PACKET SIZE -----;
COMMENT
* This procedure is called by the server to obtain the size of the packet for
* the RPC request made by the client. The packet sizes are obtained from
* nfs.h header file.
* ;
integer procedure reply_packet_size(service);
integer service;
begin
integer pkt_size;

switch case := GETATTR, SETATTR, LOOKUP, READLINK, READ, WRITE, CREATE,
REMOVE, READDIR, FSSTAT;

goto case(service);

SETATTR : pkt_size := 68; goto stop;
GETATTR : pkt_size := 68; goto stop;
LOOKUP : pkt_size := 104; goto stop;
READLINK : pkt_size := 1028; goto stop;
READ : pkt_size := 1518; goto stop;
WRITE : pkt_size := 68; goto stop;
CREATE : pkt_size := 104; goto stop;
REMOVE : pkt_size := 64; goto stop;
READDIR : pkt_size := 64; goto stop;
FSSTAT : pkt_size := 64; goto stop;

stop : reply_packet_size := pkt_size * 8;
COMMENT: convert it into bits;
end;

COMMENT ----- CLIENT RPC REQUEST -----;
COMMENT
* Stores the cumulative probability distribution of the client request input
* by the user A table is then created to determine the request that is required
* by the client.
* ;
class client_rpc_request(my_id);
integer my_id;
begin
real array request_prob(1:10);
real sum, temp;

procedure initialize;
begin
outf.outtext("Enter probability for GETATTR : ");outf.outimage;
temp := inreal/100; sum := sum + temp;
request_prob(GETATTR) := sum;
outf.outtext("Enter probability for SETATTR : ");outf.outimage;
temp := inreal/100; sum := sum + temp;
request_prob(SETATTR) := sum;
outf.outtext("Enter probability for LOOKUP : ");outf.outimage;
temp := inreal/100; sum := sum + temp;
request_prob(LOOKUP) := sum;
outf.outtext("Enter probability for READLINK : ");outf.outimage;

```

```

    temp := inreal/100; sum := sum + temp;
request_prob(READLINK) := sum;
    outf.outtext("Enter probability for READ : ");outf.outimage;
    temp := inreal/100; sum := sum + temp;
request_prob(READ) := sum;
    outf.outtext("Enter probability for WRITE : ");outf.outimage;
    temp := inreal/100; sum := sum + temp;
request_prob(WRITE) := sum;
    outf.outtext("Enter probability for CREATE : ");outf.outimage;
    temp := inreal/100; sum := sum + temp;
request_prob(CREATE) := sum;
    outf.outtext("Enter probability for REMOVE : ");outf.outimage;
    temp := inreal/100; sum := sum + temp;
request_prob(REMOVE) := sum;
    outf.outtext("Enter probability for REaddir : ");outf.outimage;
    temp := inreal/100; sum := sum + temp;
request_prob(REaddir) := sum;
    outf.outtext("Enter probability for FSSTAT : ");outf.outimage;
    temp := inreal/100; sum := sum + temp;
request_prob(FSSTAT) := 1.0;

end;

integer procedure get_packet_size(service);
integer service;
begin
integer pkt_size;

switch case := GETATTR, SETATTR, LOOKUP, READLINK, READ, WRITE,
CREATE, REMOVE, REaddir, FSSTAT;

goto case(service);

SETATTR : pkt_size := 64; goto stop;
GETATTR : pkt_size := 64; goto stop;
LOOKUP : pkt_size := 64; goto stop;
READLINK : pkt_size := 64; goto stop;
READ : pkt_size := 64; goto stop;
WRITE : pkt_size := 1518; goto stop;
CREATE : pkt_size := 68; goto stop;
REMOVE : pkt_size := 64; goto stop;
REaddir : pkt_size := 64; goto stop;
FSSTAT : pkt_size := 64; goto stop;

stop : get_packet_size := (pkt_size * 8);
COMMENT : convert it into bits ;
end;

integer procedure get_service(prob);
real prob;
COMMENT: get the service that is required by the client by checking
against the table of cummulative probability;
begin
integer service_type;

service_type := GETATTR;
while ( prob > request_prob(service_type) ) do
service_type := service_type + 1;
get_service := service_type;
end;

```

```

sum := 0.0;
initialize;

end *** CLIENT REQUEST ***;

```

```

COMMENT ----- PKT STATISTIC -----;
COMMENT

```

```

* A Record to store the statistic of the RPC calls timings.

```

```

*

```

```

* ;

```

```

class pkt_stat;
begin
integer obs;
real total_time_taken;

```

```

end;

```

```

COMMENT ----- PKT STATISTIC COLLECTOR -----;
COMMENT

```

```

* It is the statistics collector of all the traffic that this network generates.

```

```

* It "hacks" the statistic collectors used, i.e. the TALLIES to obtain the data and

```

```

* output it in a more presentable fashion.

```

```

*

```

```

* ;

```

```

tab class pkt_stat_collector;
begin
ref(pkt_stat) array stat(1:10);
ref(tally) server_trans_qtime, server_rec_qtime,
client_trans_qtime, client_rec_qtime,
disk_in_qtime, disk_out_qtime,
trans_time, disk_response_time,
disk_time;
ref(tally) f_server_trans_qtime, f_server_rec_qtime,
f_client_trans_qtime, f_client_rec_qtime,
f_trans_time, f_response_time;
integer i;

```

```

procedure reset;

```

```

begin

```

```

integer i;

```

```

for i := 1 step 1 until 10 do begin

```

```

stat(i) :- new pkt_stat;

```

```

stat(i).obs := 0;

```

```

stat(i).total_time_taken := 0.0;

```

```

end;

```

```

num_collision := pkt_counter := in_cache_count := 0;

```

```

write_count := rest_count := 0;

```

```

end;

```

```

procedure print_tally_data(tally_ptr);

```

```

ref(tally) tally_ptr;

```

```

begin

```

```

if tally_ptr.obs = 0 then

```

```

outf.outtext("-----")

```

```

else

```

```

outf.outfix(tally_ptr.sum/tally_ptr.obs,3,10);

```

```

outf.outtext(" ");

```

```

outf.outfix(tally_ptr.min,3,10);outf.outtext(" ");

```

```

outf.outfix(tally_ptr.max,3,10);

```

```

outf.outimage;
end;

procedure print_request_in_cache;
begin
    ref(tally) ptr;

    outf.outtext("          STATISTICS OF REQUESTS IN CACHE");
    outf.outimage;
    outf.outtext("          *****");
    outf.outimage;outf.outimage;
    outf.outtext("Number of request in cache is ");
    outf.outint(f_response_time.obs,5);outf.outimage;
    outf.outtext(" Title          Average   Min     Max ");
    outf.outimage;
                                outf.outtext("-----");
outf.outimage;
    for ptr :- f_response_time,f_server_trans_qtime, f_server_rec_qtime,
              f_client_trans_qtime,f_client_rec_qtime,f_trans_time DO begin
        if ptr == f_server_trans_qtime then
            outf.outtext("Server Transmission QTime ")
        else if ptr == f_server_rec_qtime then
            outf.outtext("Server Receiving QTime  ")
        else if ptr == f_client_trans_qtime then
            outf.outtext("Client Transmission QTime ")
        else if ptr == f_client_rec_qtime then
            outf.outtext("Client receiving QTime  ")
        else if ptr == f_trans_time then
            outf.outtext("Transmission Time      ")
        else
            outf.outtext("Response Time          ");
            print_tally_data(ptr);
        end;
    end;
end;

procedure print_request_that_access_disk;
begin
    ref(tally) ptr;

    outf.outimage;outf.outimage;outf.outimage;
    outf.outtext("          STATISTICS OF REQUESTS THAT ACCESS THE DISK");
    outf.outimage;
    outf.outtext("          *****");
    outf.outimage;outf.outimage;
    outf.outtext("Number of disk access is ");
    outf.outint(disk_response_time.obs,5);outf.outimage;
    outf.outtext(" Title          Average   Min     Max ");
    outf.outimage;
                                outf.outtext("-----");
outf.outimage;
    for ptr :- disk_response_time, server_trans_qtime, server_rec_qtime,
              client_trans_qtime, client_rec_qtime,disk_in_qtime,
              disk_out_qtime,trans_time, disk_time DO begin
        if ptr == server_trans_qtime then
            outf.outtext("Server Transmission QTime ")
        else if ptr == server_rec_qtime then
            outf.outtext("Server Receiving QTime  ")
        else if ptr == client_trans_qtime then
            outf.outtext("Client Transmission QTime ")
        else if ptr == client_rec_qtime then
            outf.outtext("Client receiving QTime  ")
        else if ptr == disk_in_qtime then

```

```

        outf.outtext("Time Queuing into Disk  ")
    else if ptr == disk_out_qtime then
        outf.outtext("Time Queuing out of Disk ")
    else if ptr == trans_time then
        outf.outtext("Transmission Time      ")
    else if ptr == disk_response_time then
        outf.outtext("Response Time          ")
    else
        outf.outtext("Disk Access Time      ");
        print_tally_data(ptr);
    end;
end;
end;

```

```

procedure report;
begin
    integer i, total_calls;
    real total_time, msec_per_call, percentage;

    total_calls := 0;
    total_time := 0.0;
    for i:= 1 step 1 until 10 do begin
        total_time := total_time + stat(i).total_time_taken;
        total_calls := total_calls + stat(i).obs;
    end;
    outf.outimage;
    outf.outtext("          RPC CALLS BY THE CLIENT");outf.outimage;
    outf.outtext("          *****");
    outf.outimage;outf.outimage;
    outf.outtext("Operation Percentage Calls  sec msec/call time%");
    outf.outimage;

```

```

    for i := 1 step 1 until 10 do begin
        if i = 1 then outf.outtext(" getattr ") else
        if i = 2 then outf.outtext(" setattr ") else
        if i = 3 then outf.outtext(" lookup  ") else
        if i = 4 then outf.outtext(" readlink ") else
        if i = 5 then outf.outtext(" read    ") else
        if i = 6 then outf.outtext(" write   ") else
        if i = 7 then outf.outtext(" create  ") else
        if i = 8 then outf.outtext(" remove  ") else
        if i = 9 then outf.outtext(" readdir ") else
            outf.outtext(" fsstat  ");
        outf.outfix((stat(i).obs/total_calls)*100,3,10);
        outf.outtext("%");
        outf.outint(stat(i).obs,7);
        outf.outfix(stat(i).total_time_taken/1000,3,10);
        if stat(i).obs <> 0 then begin
            msec_per_call := stat(i).total_time_taken/stat(i).obs;
            outf.outfix(msec_per_call,3,11);
        end
    else
        outf.outtext(minuses.sub(1,11));
        percentage := (stat(i).total_time_taken/total_time) * 100;
        outf.outtext(" ");
        outf.outfix(percentage,3,7);
    outf.outimage;
    end;
    outf.outimage;
    outf.outfix(length/1000,2,8);outf.outtext(" secs ");
    outf.outint(total_calls,5);outf.outtext(" calls ");
    outf.outfix(total_calls/(length/1000),3,10);

```

```

    outf.outtext(" calls/sec ");
    outf.outfix(total_time/total_calls,3,10);outf.outtext(" msec/call");
    outf.outimage;outf.outimage;outf.outimage;

    print_request_in_cache;
    print_request_that_access_disk;
    outf.outimage;outf.outimage;
end;

procedure update(service, time_taken,pkt);
    integer service;
    real time_taken;
    ref(packet) pkt;
    COMMENT: updates the statistic collectors whenever the client receives a
    response from the server;
begin
    stat(service).obs := stat(service).obs + 1;
    stat(service).total_time_taken := stat(service).total_time_taken
    + time_taken;
    hist.update(time_taken);
    if pkt.disk_access then begin
    COMMENT: update statistics for request that ACCESS the DISK;
        server_trans_qtime.update(pkt.server_trans_qtime);
        server_rec_qtime.update(pkt.server_rec_qtime);
        client_trans_qtime.update(pkt.client_trans_qtime);
        client_rec_qtime.update(pkt.client_rec_qtime);
        disk_in_qtime.update(pkt.disk_in_qtime);
        disk_out_qtime.update(pkt.disk_out_qtime);
        trans_time.update(pkt.trans_time);
        disk_time.update(pkt.disk_time);
        disk_response_time.update(time - pkt.send_time);
        disk_access_response_time.update(time - pkt.send_time);
    end
    else begin
        f_server_trans_qtime.update(pkt.server_trans_qtime);
        f_server_rec_qtime.update(pkt.server_rec_qtime);
        f_client_trans_qtime.update(pkt.client_trans_qtime);
        f_client_rec_qtime.update(pkt.client_rec_qtime);
        f_trans_time.update(pkt.trans_time);
        f_response_time.update(time - pkt.send_time);
    end;
end;
COMMENT: initialise the statistic collectors;
server_trans_qtime := new tally("ServerTransQT");
server_rec_qtime := new tally("ServerRecQT");
client_trans_qtime := new tally("ClientTransQT");
client_rec_qtime := new tally("ClientRecQT");
disk_in_qtime := new tally("InDiskQTime");
disk_out_qtime := new tally("OutDiskQTime");
trans_time := new tally("TransTime");
disk_time := new tally("Disk Time");
disk_response_time := new tally("DiskResTime");

f_response_time := new tally("QuickResTime");
f_server_trans_qtime := new tally("FSvrTransQT");
f_server_rec_qtime := new tally("FSvrRecQT");
f_client_trans_qtime := new tally("FCltTransQT");
f_client_rec_qtime := new tally("FCltRecQT");
f_trans_time := new tally("FTransTime");

join(pkt_stat_collector_q);

```

```

end;

COMMENT ----- OVERALL PACKET STAT -----;
COMMENT
* Collects the statistics for all the packets that a client generates. This gives
* the overall average response time for all the clients combined
* ;
tab class overall_packet_stat;
begin

    procedure print_input_data;
    begin
        outf.outtext("Number of servers are ");outf.outint(num_server,4);
        outf.outimage;
        outf.outtext("Number of clients are ");outf.outint(num_clients,4);
        outf.outimage;
        outf.outtext("Disk READ time is ");outf.outfix(read_time,3,6);
        outf.outimage;
        outf.outtext("Disk WRITE time is ");outf.outfix(write_time,3,6);
        outf.outimage;
    end;

    procedure print_tally_data(tally_ptr);
        ref(tally) tally_ptr;
    begin
        if tally_ptr.obs = 0 then
            outf.outtext("-----")
        else
            outf.outfix(tally_ptr.sum/tally_ptr.obs,3,10);
            outf.outtext(" ");
            outf.outfix(tally_ptr.min,3,10);outf.outtext(" ");
            outf.outfix(tally_ptr.max,3,10);
            outf.outimage;
        end;

    procedure print_pkt_stats;
    begin
        ref(tally) ptr;

        outf.outimage;outf.outimage;outf.outimage;
        outf.outtext("Number of collision is ");
        outf.outint(num_collision,6); outf.outimage;
        outf.outtext("Number of packets is ");outf.outint(pkt_counter,10);
        outf.outimage;
        outf.outtext("Total request in cache is ");
        outf.outint(in_cache_count,6); outf.outimage;
        outf.outtext("Total write to disk is ");outf.outint(write_count,6);
        outf.outimage;
        outf.outtext("Total reads not in cache is ");
        outf.outint(rest_count,6); outf.outimage;
        outf.outtext(" Title           Average   Min     Max ");
        outf.outimage;
                                outf.outtext("-----");
        outf.outimage;

        for ptr :- o_response_time, o_server_transQwait, o_serverRecQwait,
            o_client_transQwait, o_clientRecQwait,o_out_disk_queue,
            o_into_disk_queue,o_trans_time,o_disk_time DO begin
            if ptr == o_server_transQwait then
                outf.outtext("Server Transmission QTime ")
            else if ptr == o_serverRecQwait then
                outf.outtext("Server Receiving QTime  ")

```

```

        else if ptr == o_client_transQwait then
            outf.outtext("Client Transmission QTime ")
        else if ptr == o_clientRecQwait then
            outf.outtext("Client receiving QTime  ")
        else if ptr == o_into_disk_queue then
            outf.outtext("Time Queuing into Disk  ")
        else if ptr == o_out_disk_queue then
            outf.outtext("Time Queuing out of Disk ")
        else if ptr == o_trans_time then
            outf.outtext("Transmission Time      ")
        else if ptr == o_response_time then
            outf.outtext("Response Time        ")
        else
            outf.outtext("Disk Access Time      ");
            print_tally_data(ptr);
        end;
    end;
end;

```

```

procedure report;
begin
    print_input_data;
    print_pkt_stats;
end;

    join(overall_stat_q);
end;

```

COMMENT ----- MAIN PROGRAM & DECLARATIONS -----;

```

procedure read_input;
begin
    character ans;

    outf.outtext("Enter simulation length : ");outf.outimage;
    length := inreal;
    outf.outfix(length,3,10);outf.outimage;
end;

procedure initialize;
begin
    TRAFFIC := 0;
    GETATTR := 1;
    SETATTR := 2;
    LOOKUP := 3;
    READLINK := 4;
    READ := 5;
    WRITE := 6;
    CREATE := 7;
    REMOVE := 8;
    READDIR := 9;
    FSSTAT := 10;

    pkt_counter := 0;
    write_count := 0;
    in_cache_count := 0;
    rest_count := 0;
    num_collision := 0;
    IP_processing_time := 1.5;
    COMMENT: IP processing time assumes no fragmentation and
            reassembling time;
    read_input;

```

```

overall_pkt_stat := new overall_packet_stat("PktStat");
disk_access_response_time := new tally("DiskResTime");
o_response_time := new tally("ResponseTime");
o_into_disk_queue := new tally("InDiskQTime");
o_out_disk_queue := new tally("OutDiskQTime");
o_disk_time := new tally("DiskUsageT");
o_trans_time := new tally("TransTime");
o_server_transQwait := new tally("SvTransQwait");
o_client_transQwait := new tally("ClTransQwait");
o_ServerRecQwait := new tally("ServRecQwait");
o_ClientRecQwait := new tally("CliRecQwait");
end;

procedure create_server;
begin
integer i;

outf.outtext("Enter number of server : ");outf.outimage;
num_server := inint;
outf.outint(num_server,3);outf.outimage;
for i := 1 step 1 until num_server do begin
users(i) := new server("Server",i,
new randint("Msg",64,1518),
new randint("Dest",1,2),
new poisson("Think",150));
users(i).schedule(now);
end;
end;

procedure create_client;
begin
integer i, num_trans;

outf.outtext("Enter number of clients : ");outf.outimage;
num_clients := inint;
outf.outint(num_clients,3);outf.outimage;
for i := (num_server + 1) step 1 until (num_clients + num_server)
do begin
outf.outtext("Enter number of transactions per second for the client : ");
outf.outint(i-1,3);outf.outimage;
num_trans := inint;
outf.outint(num_trans,4);outf.outimage;
users(i) := new client(edit("client",i),i,
new randint(edit("msg",i),64,1518),
new randint(edit("dest",i),1,2),
new poisson(edit("think",i),(1000.0 /num_trans)));
users(i).schedule(now);
end;
end;

procedure generate_network_traffic;
begin
integer traffic, i;
real load;

i := num_server + num_clients + 1;
total_station := i;
outf.outtext("Enter amount of network traffic per second : ");
outf.outimage;
traffic := inint;
outf.outint(traffic,4);outf.outimage;
if traffic < 10 then

```

```

    load := 100
else
    load := 1000/traffic;
    users(i) :- new traffic_generator("Traffic",i,
        new randint(edit("msg",i),64,1518),
        new randint(edit("dest",i),1,i),
        new poisson(edit("think",i),load ));
    users(i).schedule(now);
end;

initialize;
create_server;
create_client;
generate_network_traffic;
waitstn :- new condq("condq");
waitstn.all := true;
masterdist :-loaddistance(total_station ,0.01);
net :- new enet("Ethernet",10000, 0.000000001,5,2,total_station,
    users,num_server);
hist :- new histogram("ResponseTime",0,1000,50);
hold(5000);
reset;
hold(length);
end;

```