Abstract:

An implementation of a reliable remote procedure call mechanism for obtaining remote services is described. The reliability issues are discussed together with how they have been dealt with. The performance of the remote call mechanism is compared with that of local calls. The remote call mechanism is shown to be an efficient tool for distributed programming.


Reliable Remote Calls for Distributed UNIX:
An Implementation Study *

By

F. Pansieri and S.K. Shrivastava

TECHNICAL REPORT SERIES

Series Editor: Mr. M.J. Elphick

Number 177
June, 1982

© 1982 University of Newcastle upon Tyne. Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
Bibliographical Details

PANZIERI, Fabio
Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1982.
(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 177.)

Added entries
SHRIVASTAVA, Santosh Kumar.
UNIVERSITY OF NEWCASTLE UPON TYNE.

Suggested classmarks (primary classmark underlined)
Dewey (18th):

001.64404
519.687

U.D.C.

Suggested keywords
DISTRIBUTED SYSTEMS
LOCAL AREA NETWORKS
RELIABILITY
UNIX

Abstract
An implementation of a reliable remote procedure call mechanism for obtaining remote services is described. The reliability issues are discussed together with how they have been dealt with. The performance of the remote call mechanism is compared with that of local calls. The remote call mechanism is shown to be an efficient tool for distributed programming.

About the author
Dr. Panzieri joined the Computing Laboratory of the University of Newcastle upon Tyne in June 1979, where he is currently a Research Associate.

Dr. Shrivastava joined the Computing Laboratory of the University of Newcastle upon Tyne in August 1975, where he is currently a Research Associate.
PANZIERI, Fabio
Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1982.
(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 177.)

Added entries
SHRIVASTAVA, Santosh Kumar.
UNIVERSITY OF NEWCASTLE UPON TYNE.

Suggested classmarks (primary classmark underlined)
Dewey (18th):
001.64404
U.D.C.: 519.687

Suggested keywords
DISTRIBUTED SYSTEMS
LOCAL AREA NETWORKS
RELIABILITY
UNIX

Abstract
An implementation of a reliable remote procedure call mechanism for obtaining remote services is described. The reliability issues are discussed together with how they have been dealt with. The performance of the remote call mechanism is compared with that of local calls. The remote call mechanism is shown to be an efficient tool for distributed programming.

About the author
Dr. Panzieri joined the Computing Laboratory of the University of Newcastle upon Tyne in June 1979, where he is currently a Research Associate.

Dr. Shrivastava joined the Computing Laboratory of the University of Newcastle upon Tyne in August 1975, where he is currently a Research Associate.
Reliable Remote Calls for Distributed UNIX: An Implementation Study

F. Pansieri and S.K. Srivastava

Computing Laboratory
The University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU
England.

Abstract
An implementation of a reliable remote procedure call mechanism for obtaining remote services is described. The reliability issues are discussed together with how they have been dealt with. The performance of the remote call mechanism is compared with that of local calls. The remote call mechanism is shown to be an efficient tool for distributed programming.

Introduction
We are currently in the process of implementing a UNIX® based distributed system with the objective of investigating various reliability issues in the design and implementation of such systems. Our current hardware configuration consists of a number of PDP11 computer systems (nodes) connected by the Cambridge Ring local area network[1]. Each node has at least 10 Mbytes of disc storage, runs the UNIX V7 operating system and supports a small group of users with broadly similar research interests. In this context, we envisage the distributed system as a loosely coupled system of largely autonomous nodes. This view has led us to assume that in such a system, users’ accesses to system resources will be confined largely to those local to their nodes and further, any remote access will typically be concerned with file manipulations. It was our task to implement appropriate protocols for inter-process communications. Two design decisions were taken at the outset: (1) to provide a uniform interface to both local and remote objects, and (2) to make no changes to the UNIX operating system. Since UNIX provides a procedure call based interface to all services (e.g. file open, close), it seems natural to provide access to remote services through a remote procedure call (RPC) mechanism. The implementation of such a remote call mechanism is the subject of this paper. In view of the second design decision, all of the implemented RPC software in effect runs as a UNIX application software, hence the RPC response can never be made as good as that from a local system call. To get reasonable remote responses, it was in our interest to cut down the RPC software to the barest minimum, thus implying the usage of simple communication protocols; an apparently conflicting requirement with that of achieving reliability. As the performance figures for the RPC show, we have managed to get acceptable response times for remote file accesses (the most likely use of the RPC); however this has not been achieved by compromising reliability.

1. RPC Semantics and Reliability Requirements
An implementation of a RPC mechanism essentially involves sending the client’s request as a message to the appropriate server and then receiving the server’s response which constitutes the end of the call. Since the client and the server are on different machines, any good RPC mechanism must cope effectively with the problems arising from a crash of one of the machines and any unreliability of the underlying data communication facility. These problems and their solutions are discussed in[2,3], and our particular approach to reliable RPC is described in[4]. Since the design of the RPC mechanism has already been described elsewhere[4], we shall not elaborate it here. However, for the sake of completeness, a very brief discussion on reliability problems and their solutions is presented here.

Figure 1 depicts the message exchanges for a call between a client and a server.

```
Client               Server
"request"             "work"
send (...)          ----->   receive (...);

("result")
```

Figure 1: simple RPC.

UNIX® is a Trademark of Bell Laboratories.
We assume that invocation of the message retry mechanism (in the interest of fault tolerance) can result in multiple messages directed at the destination. This can result in more than one execution of "work" (orphan executions \([2,3]\)) at the server. A simple sequence numbering scheme can solve this problem since duplicate messages can now be recognized and hence rejected (reliability requirements R1 and R2):

R1: client's request message must include a sequence number (SN) which must match that of the corresponding result message (note: all retries of a message contain the same SN).

R2: SN's must survive node crashes.

We assume that recovery from a node crash involves starting up the node from some initial state. However, if SN's after a crash recovery of a client's node are the same as before the crash, then there is the possibility of a server confusing new requests with old ones and thus refusing to accept some of these requests. Worse still, there is the possibility of a client accepting a wrong result: this could happen if the client's node crashes just after the "call request" has been sent and, after the crash recovery, the client makes a request to the same server with the same SN, then the possibility of the client accepting the results from the previous call certainly exists. The above analysis indicates the need for the requirement R2. A crash of the client's node in the middle of a call can also result in an orphan; so requirement R3 is needed:

R3: a server must detect an orphan and "undo" the work done before accepting a new call.

The above three reliability requirements must be met to ensure the "exact once" semantics of RPC (that is, a successful call implies one execution at the server; incomplete calls produce no side effects). As discussed elsewhere [4], we employ a network-wide unique sequence numbering scheme (based on the loosely synchronised clock approach\([5]\)) to meet requirements R1 and R2 in the RPC level software. However, we have chosen not to meet the requirement R3 at the RPC level; rather, such orphans are treated at a higher level: the "recovery" level concerned with the maintenance of atomicity of user programs\([6,7]\). These design decisions have allowed us to implement RPC quite cheaply based on a potentially unreliable message sending facility (datagrams). The following is the semantics of the RPC (where the "timeout" parameter specifies how long the client is willing to wait for a response):

```plaintext
remote_call(server ...;
  service...;
  var result...;
  var r_stat...;
  timeout...);
```

where `r_stat` is of type `status`:

```plaintext
status=(OK, not-done, absent, unable);
```

and parameters and results are passed by value.

The meaning of the call under various responses is given below:

- **status = OK**: The service specified has been performed exactly once by the server and the answers are encoded in "result".
- **status = not-done**: The server has not performed the service because it is currently busy (so the client can certainly re-issue the call if he wants to get an "OK" response).
- **status = absent**: The server is not available (so it is pointless for the client to retry).
- **status = unable**: The parameter "result" does not contain the answers; whether the server performed the service is not known (this response can be obtained when the timeout expires). If the service required has the idempotency property then the client can certainly re-issue the call without any harm; otherwise the services of the "recovery" level must be invoked to undo any side effects produced before reissuing the call \([7]\).

The RPC mechanism whose implementation is described in the rest of the paper meets the above specification provided the following conditions are met: (i) a node's hardware components (e.g. CPU, clock, ring Interface) are working according to their specifications; (ii) the ring is working according to the specifications; and (iii) the UNIX system of a node is also working according to the specification. If failure modes of these three subsystems were "clean" (working or not working) then a failure of subsystem (i) or (ii) will constitute a node crash and has the net effect of all of the ongoing calls on that node not succeeding and any services provided by that node becoming unavailable; a failure of the ring has the effect of all of the nodes becoming disconnected (so a node will be unable to obtain remote services). In practice of course, failures are rarely "clean", nor are precise system specifications available. So, whenever it is suspected that a subsystem is not working properly, the best strategy is to convert that failure to a clean node or ring failure by switching off appropriate power supply or by stopping the operating system as the case may be. In the case of a crashed node, once it has been repaired, it can be re-inserted in the system dynamically; there is no need for either stopping the entire system to accommodate the new node or to specifically inform the live nodes that a new node has been added to the system.
2. RPC Implementation

This section describes the user interface supported by the RPC software and certain aspects of its implementation. The implementation has been performed in C language; however for the sake of readability, the algorithms have been described in a pseudo language ("["] and ") stand for "begin", "end" and comments are enclosed within quotes). But first, a few remarks of general interest are in order:

(a) The message passing system employed by the RPC mechanism uses a naming scheme based on "port" numbers (integer values). A message is delivered to a given port at a given node; so the process that is "attached" to that port becomes the recipient of the messages directed at that port. Some higher level "name server" will typically be required through which various servers can publish their port numbers for receiving requests. The following two primitive operations are available:

(i) send_msg(destination...; message,...; var msg_status,...);

where "msg_status" is of type "status", "destination" is a record containing the node number (each node has a unique number) and the port number is an array of bytes. The response "OK" implies that the message has been delivered to the appropriate port; while the response "absent" means that the node is not connected to the ring. The response "not-done" indicates that the message was rejected possibly because the recipient is busy (so the sender can certainly retry). The response "unable" indicates a ring malfunction during the transmission: the message may or may not have reached the destination.

(ii) receive_msg(at:port_number: var node:source; var message,...);

This procedure receives a message directed at port "at" from the specified source "node". If source = "any" then messages from any node directed at "at" are accepted (in any case, "node" will contain the node number of the sender).

(b) The sequence number (SN) used in a message is derived by concatenating the current value of the local clock of the node and the node number (clock value, node number); a function get_sn(...) has been implemented that returns a sequence number.

(c) The formats of call and return messages are shown in figure 2.

The caller supplied information includes the port number for receiving the reply, server operation and the necessary parameters. The maximum length of a message has been fixed to

<table>
<thead>
<tr>
<th>&lt;= 2 bytes -&gt;</th>
<th>&lt;= 2 bytes -&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence--</td>
<td>sequence--</td>
</tr>
<tr>
<td>number-- P1</td>
<td>number-- P1</td>
</tr>
<tr>
<td>reply port</td>
<td>reply port</td>
</tr>
<tr>
<td>OP-code</td>
<td>OP-code</td>
</tr>
<tr>
<td>parameters P2</td>
<td>parameters P2</td>
</tr>
<tr>
<td>results P2</td>
<td>results P2</td>
</tr>
</tbody>
</table>

'call' message  

Figure 2: call and return message formats.

that necessary to return a page of data as a result.

2.1. User Interface

In addition to the "remote_call(...)" operation available to clients, two operations - to be employed by the servers - are also provided by the RPC interface: (i) the operation "get_work(...)" is used by a server to receive a call request, and (ii) the operation "send_result(...)" is used by a server to send the results of the executed call.

The "remote_call(...)" operation transfers an array of bytes (parameter "service") to the named server (i.e. to the appropriate port) and returns an array of bytes (parameter "result"). It is left to servers and clients to view these byte arrays as structured objects. A client's view of the 'call' message is the portion P2 of figure 2(a); the remote_call software constructs the portion P1 of the message, thus hiding unnecessary details of sequence numbering and reply port from the client. Similarly, a client only sees the portion P2 of the returned message (figure 2(b)). A number of 'pack' procedures have been provided for packing simple typed variables (integers, strings, etc.) onto an array of bytes; a complementary set of 'unpack' procedures are also available for constructing typed variables from an array of bytes. Here is a rather simplified description of how a client can perform a remote file read operation (assume that the client knows the node number and the port number of the file server).

In UNIX, a file read operation must specify the file descriptor (an integer value), the address of buffer for storing the data and the number of bytes to be read. The client needs the following variables:
2.2. Some Implementation Details

The program below shows those details of the "remote_call(...)" that are to be done with the provision of fault tolerance and mapping of message responses onto the status return of the call:

begin "remote_call(...)" implementation"
...
sm := get_sm(...);
at := get_port(...); "get port number"
retry := 0;
done := false;
repeat "call request to server"
{ send_msg(server, msg, msg_status);
  case msg_status of
    ok : { done := true }
    absent : { r_stat := absent; return }
    not_done : { retry := retry + 1;
                 if retry = MAXTRY then
                   { r_stat := not_done; return }
                 }
    unable : { retry := retry + 1;
               if retry = MAXTRY then
                 { r_stat := unable; return }
               }
  }
until done

end;
end;"case"
until not done;
within limit do
{ repeat
  receive_msg(at, server_node, result);
  until result.sm = sm
  <EXPIRED> { r_stat := unable; return }

end;

In the current implementation, MAXTRY has been set to 10. A typical timeout period for a file read operation should be set by the user to a value ranging between 1 and 2 seconds, depending on the system load.

A server has to maintain certain global data which is initialized at the node start-up time. This data is the last largest sequence number (llsn) received from a given node, and is initialized to the current clock value of the server's node:

var llsn : array[...] of sm;
for i = 1 to maxnode do
llsn(i) := get_sm(...);

Thereafter, the above data structure is utilized by the get_work(...) procedure to accept a valid request. The send_result(...) operation uses a similar technique to that of remote_call(...) for transmitting the result:

procedure get_work(...);
...
begin
  src := any;
  receive_msg(port, src, msg);
  sm := src.sm in msg;
  until sm > llsn(src);
  llsn(src) := sm;
...
end;

3. Clock Management

As stated before, a sequence number at a given node is constructed out of the current clock value and the unique node number. All the clocks of the system are kept loosely synchronized with each other so that they represent roughly the same physical time [5]. A simple way to achieve this objective was described in [4]. All that is necessary is for each node to maintain two processes: (i) a broadcaster process that regularly (say once every few minutes) sends its current time to the rest of the nodes; and (ii) a synchronizer process whose task is to receive the time sent by others and to advance its clock if it is behind that of a given sender.

Three practical problems were faced in the implementation of the above scheme.

(1) A client with a slow clock can experience difficulty in obtaining services from a server if the server relies on its own
clock for deciding whether to accept or reject a request.

(11) For the sake of efficiency, we would like that a broadcaster need only send its time to those nodes that are currently "up". The Cambridge Ring can accommodate up to 258 nodes, though our ring is currently rather sparsely populated. This suggests the need for a dynamically maintained "up" list at every node such that if a new node is inserted in the ring, the broadcasters of the other nodes eventually discover this fact (and thus can start sending their times to the new node) and similarly if a node is removed from the ring, this fact is also discovered by the rest of the "up" nodes.

(111) The fact that clocks are always advanced implies that "fast" clock errors will accumulate and in particular a runaway clock can advance the network time far ahead of the physical time. This suggests that a facility for setting clocks back is needed. Of course, this must be performed without compromising the security offered by the sequence numbering scheme.

The only way of avoiding problem (1) was to maintain the clock values of the other nodes at the server's node (the '1sm' array mentioned in the last section). The solutions adopted to solve the remaining two problems will now be described. The solution adopted for setting back the clocks is quite practical if this operation were not invoked frequently (certainly true in our case as only a few adjustments are needed every month). When the clocks are being set back, users are likely to encounter some difficulty in obtaining remote services; however the system quickly stabilises (within a few minutes). The authority for setting clocks back is vested in the broadcaster of one node only — this special broadcaster will be referred to as the Time Lord. The Time Lord can send three types of messages:

**sy**ych: this is the normal message containing the current time;

**go**back: this message is a "get prepared to set time backwards" message.

**set**: this message contains the new time.

All the other broadcasters can only send the "sync" messages. The algorithms for a broadcaster (not a Time Lord) and a synchroniser of a node are as shown. Two shared variables "dir" and "uplist" are maintained and protected by a mutex semaphore.

"Shared variables"

```
var dir : (forward, backward);
uplist : array [1..maxnodes] of (UP, DOWN);
mutex : semaphore; "Initially 1"
```

"BROADCASTER ALGORITHM"

```pascal
var retry : integer; done : boolean;
begin
P(mutex);
dir := forward;
V(mutex);
set_clock(0); "Initialise clock"
for i := 0 to max_node do
begin
retry := 0; done := false;
repeat
send "I am alive" synch message
send msg(..., 0, msg_status);
P(mutex);
case msg status of
ok : { uplist(i) := UP; done := true }
absent : { uplist(i) := DOWN; done := true }
unable, not done : { if retry := retry + 1;
if retry = max then
uplist(i) := DOWN }
end "end case"
V(mutex);
until retry = max or not done
end "end of broadcaster initialisation program"
cycle
delay(t); "wait for a minute"
P(mutex);
if dir = forward then
{ get sm(...);
send the sequence number as synch messages to all "UP" nodes }
V(mutex)
end "end cycle"
end; "end broadcaster"
```

"SYNCHRONISER ALGORITHM"

```pascal
var receive msg(pt, src, message);
P(mutex);
case message type of
goback : dir := backward;
synch : { if dir = forward then
if advance clock if necessary;
if uplist(src) = DOWN then
uplist(src) := UP }
set : { if dir = backward then
{ set_clock(message.time + 1tick);
dir := forward }
end "end case"
V(mutex)
end; "cycle"
```

When a node comes "up", the broadcaster of that node sends "I am alive" messages to all the possible nodes in the system. The responses received from the send message operations are used for the construction of the "uplist". We have assumed that all the synchronisers have got the same port number, so a broadcaster's messages always go to synchronisers. Also, note that the synch message with time 0 will not affect local time at any node. A broadcaster only broadcasts its time if it is going forward.

The synchroniser's task is to analyse the
messages directed to it and act accordingly. The algorithm for the Time Lord is given below (the synchroniser at the Time Lord’s node is identical to other synchronisers). The user at the Time Lord’s node has to supply a GO BACK command which is caught by the Time Lord who then sends “go back” messages to all the “up” nodes. This has the net effect of stopping all the broadcasters. When the Time Lord gets the user supplied new time, it broadcasts it which has the net effect of initialising all the clocks. Note that no special action is needed for a node to set up its clock when it comes up—it’s synchroniser will get a clock value that will result in the update of the clock.

"TIME LORD ALGORITHM"

---- "initialisation part, same as a BROADCASTER" -----

cycle
  if GOBACK command from the user then
    send “go back” messages to all “up” nodes, including own synchroniser;
    get a new clock value from the user;
    send “set” messages to all the “up” nodes, including own synchroniser;
  else (same as a BROADCASTER)
end;

Finally, all the servers must also discover the fact that clocks have been set back - so that they can adjust their lsn arrays (mentioned in section 2.2); otherwise requests will continue to be rejected until such time as the "new" time catches up. A simple means of performing this task is shown below - a slight modification to the get_work(...) procedure is needed. A “reject” count is maintained and whenever this count exceeds a clock value, (say 5), it is suspected that clocks have been put back, so the lsn array is initialised.

"modified get_work() algorithm"

procedure get_work(...);
begin
  reject := 0;
  repeat
    src := any;
    receive_msg(port,src,msg);
    if sn <= lsn(src) then
      reject := reject + 1;
    until sn>lsl(src) or reject=MAX;
    if reject=MAX then
      for i := 1 to maxnode do
        lsn(i) := sn;
    else
      lsn(src) := sn;
      reject := 0;
    end
  end;

A few remarks regarding the actual UNIX implementation are perhaps in order. In UNIX only files can be shared between unrelated processes; so the shared variables of our algo-

rithms are kept in a file. This has not caused any performance problems since both the broadcast and the synchroniser have very little active processing to perform. The maintenance of up lists and the facility of putting clocks back certainly add complications. Nevertheless the resulting algorithms are still fairly simple with very little message and computation overheads.

We conclude this section by pointing out two aspects of the clock management scheme: (i) Let “d” be the maximum clock drift, then the minimum crash recovery time of any crashed node must be greater than “d” to guarantee that the SN’s after a crash are greater than SN’s before the crash; (ii) A crash of the Time Lord’s node cannot be tolerated when clocks are being put back. If crash recovery time is known, then the observation (i) indicates how much drift between clocks is tolerable, which then can be used to calculate how often clocks need be synchronised. Although we have not made a detailed study of clock drift, sending synchronising messages once every few minutes (say 2 minutes) results in only a small drift (2 – 3 seconds per week) which appears acceptable.

4. Performance Measurements

This section presents some of the results obtained from initial tests carried out to assess the performance of the RPC mechanism. Our aim was to compare the data transfer rates as seen by the user for local and remote operations. The measurements were carried out between two PDP 11/45 computers and involved (for the distributed case) a program on one cpu to read 50 KBytes of data from an already opened file on the other cpu. Various block sizes for the transfer were utilised: from 16 bytes to 512 bytes (one page). The standard “time” facility of UNIX was used for obtaining the times which are accurate to a millisecond. The graph of figure 3 shows the results obtained, where “local” figures refer to the results obtained when only one cpu was utilised.

From this graph it is seen that when the unit of transfer is a page of data, the inter machine transfer time is about 40% slower than the local transfer rate. This degradation is tolerable since initial user experience indicates that most users have been unable to differentiate between a local file operation and a remote one. Some additional performance data are presented in table 1 (a block size of 512 bytes was used and the same file was used for the first test).

As a matter of interest, the data transfer rate obtained by utilising a high level virtual circuit based protocol (Byte Stream Protocol) was also measured: the result obtained was 1.46 KBytes against 5.36 KBytes - confirming the opinion expressed in [3] that sophisticated protocols are often undesirable. The process to process data transfer rate (involving no disc accesses) for the distributed case appears to be rather small if one takes into account the fact that the
Figure 3: data transfer rate.

<table>
<thead>
<tr>
<th>Test</th>
<th>local</th>
<th>remote RPC</th>
<th>remote ESP</th>
</tr>
</thead>
<tbody>
<tr>
<td>open; read; close</td>
<td>9.12 KB/s</td>
<td>5.36 KB/s</td>
<td>1.46 KB/s</td>
</tr>
<tr>
<td>process to process</td>
<td>41 KB/s</td>
<td>7.4 KB/s</td>
<td>---</td>
</tr>
<tr>
<td>(Unix pipe)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>open or close file</td>
<td>18 usec</td>
<td>60 usec</td>
<td>60 usec</td>
</tr>
</tbody>
</table>

Table 1: additional performance data.

Ring bandwidth is 1.25 Mbytes/sec! The reason for this is that the Ring itself transmits two bytes of data at a time and our Ring interfaces are interrupt driven. So transmission of a block of data involves considerable interrupt handling overheads (the maximum raw data transfer rate of the message passing system is about 8 Kbytes/sec). We are in the process of upgrading the interface hardware to provide the DMA (Direct Memory Access) access to CPU memory - this will considerably enhance the performance of the RPC mechanism.

The only part of the RPC mechanism that appears as bit complex is the clock management scheme needed to meet the reliability requirement R2 (see section 1). However, as the clocks are only loosely synchronised (a node sends its time to the other "up" nodes once every few minutes), processing overheads for keeping clocks approximately the same are not appreciable. Thus the cost of attaining reliability has been kept low.

5. Concluding Remarks

The RPC mechanism is operational and available to our colleagues for programming basic facilities of a distributed system (e.g., a file server at each node). As stated earlier, the RPC software does not prevent all the cases that can generate orphans. Our next task is to design a layer that supports the abstraction of atomic actions. This level will include the mechanisms for concurrency control, recovery and two phase commit[8] and as a result will deal effectively with any outstanding orphans.

6. Acknowledgements

This work was supported by the Science and Engineering Research Council (U.K.) and by the Royal Signals and Radar Establishment (U.K.). The datagram facility was implemented by W. P. Sharpe of Science and Engineering Research Council. Acknowledgement is also due to our colleague L. F. Marshall whose knowledge of the UNIX system proved invaluable.

References


