Proceedings of International Conference on Distributed Systems, Software Engineering and Database Systems

# Experience with Building Reliable RPC Servers Based on a Fault Tolerant RPC Library

Jyh-Tzong Chiou, Charles Changli Chin, Shang-Rong Tsai
Department of Electrical Engineering
National Cheng Kung University, Tainan, Taiwan, ROC

**Abstract**

*This paper presents a fault tolerant RPC mechanism based on Sun RPC and IP multicast protocol. The fault tolerant RPC mechanism is provided as an RPC library, called libFTRPC, whose interface is compatible with that of conventional Sun's RPC library. Thanks to this compatibility, a reliable RPC server can be developed in the same way a conventional RPC server is constructed. The service reliability is ensured by replicating the server to a group of server replicas. Coordinator-cohort replication model in conjunction with read-one/write-all policy is used to guarantee state consistency between the server replicas.*

*We also present our experience with building several reliable servers, including the portmapper and NFS server, over our fault tolerant RPC mechanism. The results of performance evaluation shows that the reliable NFS server has the same performance as the single unreplicated NFS server has when serving read-only requests.*

## 1. Introduction

As a major interprocess communication mechanism in distributed environments, RPC (Remote Procedure Call) [1] has been widely provided and has become a popular method in constructing distributed applications. Its attractive advantage is simplicity. RPC makes constructing distributed programs an easy task by extending the conventional procedure call scheme to distributed environments and hiding from programmers the complications involved in communication, concurrent, and transmission errors. When writing RPC programs, programmers can use programming paradigm similar to conventional local procedure call while calling and called procedures are allowed to reside on different machines.

Unfortunately, despite the convenience provided by RPC, most existing RPC mechanisms lack in support for fault tolerance. Suppose a client makes a service request to a server; this server might fail due to underlying hardware failures while it is executing the requested procedure. The client will be suspended indefinitely awaiting the result of the call. If this server is responsible for providing important system services, its failure might cause a disaster in the system.

We have developed a fault tolerant RPC mechanism [10] by which an RPC server can be made fault tolerant by replicating it to a group of server replicas running concurrently on different nodes among the network. In the face of node failures, the server can continue to provide services as long as at least one server replica survives the failures. We adopt the coordinator-cohort method in conjunction with read-one/write-all policy to achieve the goal of fault tolerance. One of the server replicas is designated as the coordinator and the others act as cohorts. When serving requests from clients, the coordinator is responsible for coordinating the cohorts to maintain state consistency among the server replicas.

In the proposed RPC scheme, we have incorporated the process group abstraction [2,6,7] into the fault tolerant RPC model. The replicas of a server group are viewed as a process group. IP multicasting [3] is used as the underlying communication protocol for coordination between group members (server replicas). It helps to reduce the overheads of sending a message to all server replicas one by one. Furthermore, we also exploit the parallelism of the server replicas to have better performance when dealing with read-only requests.

In implementing the fault tolerant RPC mechanism, we have paid special attention to failure transparency and replication transparency. Existing clients can access the fault tolerant server group in the same way as they access conventional RPC server, without any modification to the client programs. In case of server failures, there is no need for clients to participate in the activities of server recovery.

This paper is structured as follows. In next section we introduce the overview of the proposed fault tolerant RPC mechanism. A new concept of Distributed Reliable Virtual Machines built on the fault tolerant RPC is also introduced. In section 3, the basic assumption, state machine assumption, for RPC servers is made. How a RPC server

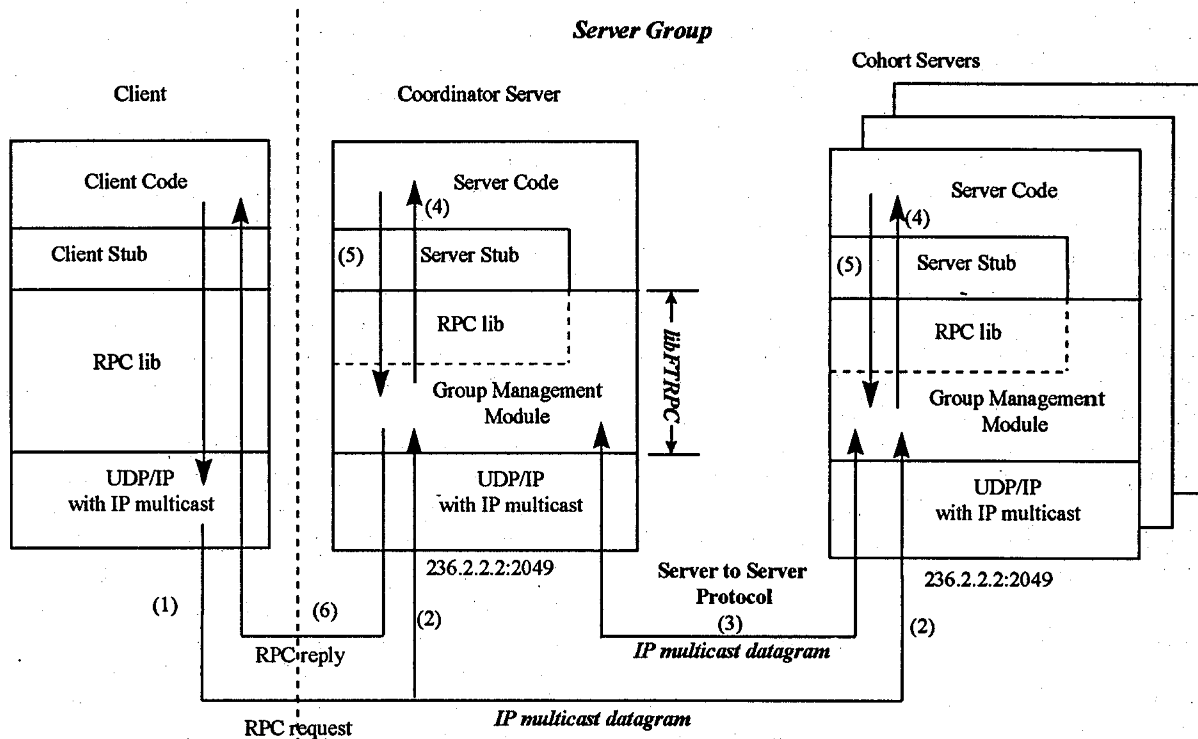


Figure 1. Structure of the Fault Tolerant RPC Mechanism

program initializes to join a server group and to run on a DRVM is also presented. In section 4, we present our experience with building fault tolerant RPC servers over our fault tolerant RPC mechanism. Finally, the conclusions of this paper are given.

## 2. Overview of Fault Tolerant RPC Mechanism

Our fault tolerant RPC mechanism is implemented over Sun RPC [8] and IP multicast protocol [3], and is intended to provide RPC programs with the same interface as what Sun RPC provides. With this compatibility feature, many servers based on Sun RPC, such as the Sun NFS server [9] and the NFS mount server, can be easily ported to be their fault tolerant counterparts.

### 2.1 Software Structure of Fault Tolerant RPC Library

The layered software structure of the fault tolerant RPC mechanism is depicted in Figure 1. Compared with standard Sun RPC scheme, the notable difference is that the RPC library is extended to a fault tolerant RPC library, libFTRPC, which includes an additional process group management module. Process group is an operating system level abstraction that has been proposed as a desirable support for enhancing system reliability, improving system parallelism, and simplifying the design of distributed applications. In the fault tolerant RPC environment, an RPC programmer can think of his server as a group of server replicas collaborating to provide fault tolerant services to their client processes. From the client's point of view, however, it has the illusion that it just faces a single unreplicated server. Thus, server group is transparent to clients. The group management module is responsible for coordinating the members of a server group in order to provide this replication transparency. In addition, this module also handle the details relevant to fault tolerances, such as reconfiguring server group when group member leaving or joining the group, recovering server's state when server member reboots from its failures, detecting failures of servers and, furthermore, balancing workload of requested calls among server members. The protocol used for process group management module in a server to cooperating with other server members is called the Server-to-Server Protocol (abbreviated as SSP).

As shown in Figure 1, in order to take advantage of fault tolerance, the server procedures need to be linked with server stub and the libFTRPC. The server stub is just the same stub as is used in conventional Sun RPC systems. To start up a service, a user can replicate his or her server

to a number of server replicas by running the same server program on different nodes among the network. When one server begins its execution, it uses SSP to join the server group and to gather the membership information of the server group. Then, it waits for the arrival of the request messages sent by client.

It is important to note that all server members in a server group must associate their services with the same port number and have their service sockets joined in the same IP multicast address, e.g., 236.2.2.2 as shown in Figure 1, in order to receive client requests and SSP protocol messages. For the outside world, this IP multicast address in conjunction with the server port number are the identifier of the server group. Hence, we can logically think that there is a single unreplicated server (actually, a group of servers) running on a virtual machine with 236.2.2.2 as its IP address. This conception leads to the DRVM (Distributed Reliable Virtual Machine) abstraction described subsection 2.3.

### 2.2 Read-one/write-all Coordinator-cohort Scheme

We adopt a variant of the coordinator/cohort approach as our replication technique to maintain state consistency among all server replicas in a server group. Here, we just give a brief description of this scheme in our fault tolernat RPC mechanism; for more details, refer to [10].

When a client makes a remote call, it invokes a client stub procedure to marshal the parameters of the remote call and to assemble this marshaled parameters with the remote call identifier into a request message, and then multicasts this request message to all members in the server group. In a server group, there are two types of server replicas: one of the server is designated as the coordinator and the others are cohorts. Upon receipt of the request message, instead of executing the requested procedure immediately, the cohorts put the request message in a request queue. Unlike cohorts, the coordinator forward a message concerning the client request to all cohorts. This forwarded message contains a sequence number and a message signature that can uniquely identify the original client's request message. The sequence number determines when this client request should be served with respect to other client requests.

After the forwarding phase, the servers call the dispatch routine of RPC library to dispatch this request and cohorts serve requests according to the forwarded messages from the coordinator. The SSP protocol will guarantee that all cohort serve all client requests in the same order as defined by the coordinator. Therefore, all servers in the same group will have the same server state if all of them start with the same initial state. Hence, one-copy serializability is guaranteed. After the requested procedure completes, the coordinator replies its return data to client on behalf of the server group.

Requests whose corresponding service routines do not modify server state could be handled in the same way as is described above. But, this seems unnecessary because such requests do not change anything. In our coordinator-cohort model, the replica coordination is relaxed for read-only requests in order to have better performance and response time.

When a read-only request issued from a client arrives in the server group, each server replica makes a decision on who should serve the request. This decision is made according to the server number of the server replica and the transaction id of the request. The server number is assigned to a server replica while it joins the server group and may change when other server replicas join or leave (crash) the group. The server number ranges from 1 to n; n is the number of server replicas in the server group. The transaction id is assigned randomly by RPC routine on the client side and is put in a request message. The server whose server number equals ((m.x_id mod n)+1) is selected to serve the read-only request, where m.x_id is the transaction id of the request message m. The other servers just ignore this request by discarding it.

To provide a way for server program to specify the read-only service routine, a library routine, read_only(), is included in libFTRPC. The read_only() routine takes the program number, version number and procedure number of the read_only service routine as the parameters.

With the support of read-only operation, we hope that the workload of the read-only requests can be distributed among the server hosts. Therefore, in case of multiple clients, the performance of the server group will be better than a single server if most of the requests are read-only.

Since clients use the IP multicast protocol to send requests to all servers, they can access the fault tolerant service transparently without the knowledge of the existence of the server replication. If there are failures occurring to servers, failure recovery can be taken care of entirely within the server group without clients getting involved. When the coordinator crashes, a cohort would be elected as new coordinator to continue to coordinate all other nonfaulty cohorts. The client can continue to access the service because the reformed server group still use the same group identifier as the old one. Using IP multicast protocol eliminates the need for clients to switch to the new coordinator from the failed one.

Owing to the use of IP multicasting, the fault tolerant RPC program can only run on nodes with installation of IP multicast protocol, which has already been provided for a variety of UNIX systems. Since IP multicast can support only UDP transport, our fault tolerant RPC mechanism can't support RPC programs running over TCP transport.

Since we have kept most of the libFTRPC interface compatible with the original SUN RPC, most of the server code can be retained when an existing server is made fault tolerant. What the programmer needs to do is to add procedures to deal with transferring server state from the coordinator to a new joining server, as described in section 3. We have taken advantage of this feature to port the portmapper, Sun NFS server and mount server into our fault tolerant environment by making very little modifications to their existing server programs.

### 2.3 DRVM-Distributed Reliable Virtual Machine

As described above, a server is replicated to a group of server members running on a collection of nodes connected by a network. Figure 2 shows a possible configuration where there are four groups of servers distributed on the network. The four groups are PMAP (portmapper), NFSD (Sun NFS server), MOUNTD (NFS mount server) and a user-provided server (server developed by application programmers). Each of NFSD group and MOUNTD group consists of three server members, and each of PMAP group and user-provided server is assigned a two-member group.

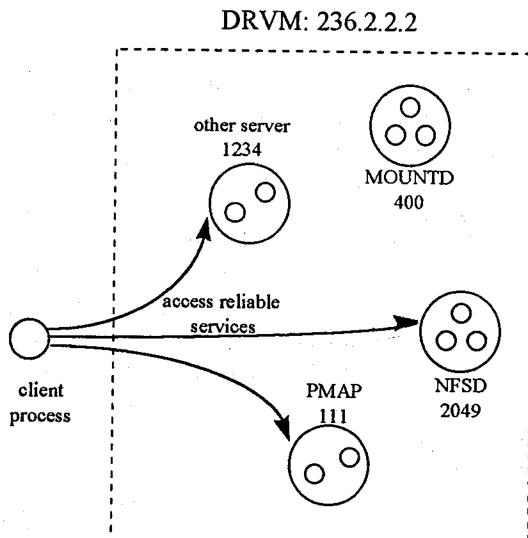


Figure 2. DRVM - Distributed Reliable Virtual Machine

The numbers below the group names are the UDP port numbers used by service ports on which the servers wait for the arrival of requests. Suppose that all the servers associate their service ports with the same IP multicast address, e.g. 236.2.2.2. Then, for example, a client can access the NFS service by sending a request message destined for port 2049 at host with IP address equal to 236.2.2.2. Clients can make requests to the other server groups in similar manner. Given this, from the client's point of view, it seems as if there were a host with IP address 236.2.2.2 and four servers were running on it.

We have modeled this abstract host as a Distributed Reliable Virtual Machine (DRVM), as illustrated in Figure 2. From the client's point of view, a DRVM is a reliable host that provides reliable service via its designated IP multicast address. When writing RPC programs in our fault tolerant RPC environment, a programmer can imagine that his server programs would run on the DRVM. As in standard Sun RPC system, his server program may choose to register its service with its DRVM's portmapper and client programs can look up the port number of any remote service supported by a fault tolerant server group from the portmapper.

## 3. State Machine Assumption and Server Initialization

In the following discussion, we define that a server consists of state variables, which represent the server state, and operations, which read the state variables and/or change the server state. The program variables and service routines of a server program implement its state variables and operations respectively. A client makes a request to execute a service routine, therefore applying an operation on the server state. Execution of each service routine is atomic; i.e., it can't be intercepted by other service requests. Besides, server program is assumed deterministic. That is, if all server replicas begin execution with the same initial state, all of them will reach to the same server state, generate the same sequence of responses and outputs when given the same sequence of requests.

In fact, the above definition and assumption consider a server to be a so-called state machine [5]. And, a server group implements a fault tolerant state machine. The key, as is pointed out in [5], for implementing a fault tolerant state machine is to ensure Replica Coordination; i.e., all replicas receive and process the same sequence of requests. In the fault tolerant RPC mechanism, replica coordination is taken care of by the server-to-server protocol (SSP), which is the primary task of the process group management module in Figure 1.

*Server Initialization*

When a server replica starts up, it needs to join the server group that it belongs to. By joining server group, a replica can learn the membership information of the group. Also, joining server group allows the server replica to acquire the current server state, i.e., the values of the state variables, if there exist other acting servers. Thus it can have the correct state consistent with the rest of the server replicas in the group.

The user-written initialization code for a fault tolerant

server program is quite similar to that of a traditional Sun RPC server program. One exception is that it begins with a routine named MAIN() rather than main(). When starting a server program, a user has to specify an IP multicast address as the first arguments for the server program. This IP multicast address is considered as the IP multicast address of the DRVM on which the server is assumed to execute. The main() routine provided in the libFTRPC keeps this IP address in a global variable. After the system-provided main() completes, the user's MAIN() is then called and the rest of the arguments following the IP multicast address are passed to it.

In the MAIN() routine, a server replica creates a UDP socket and binds a local port to the socket. It then creates a UDP handle using svcudp_create() in libFTRPC and calls the svc_register() to declare its program number, version number and UDP handle to the RPC library. The svc_register() also registers the program number, version number and port number with the DRVM portmapper if the server choose to do it. Note that, at this point, svc_register() automatically associates the UDP socket with the user specified IP multicast address. Therefore, when the server replica waits on the UDP socket, it can receive the messages destined for its server group executed on the specified DRVM. MAIN() routine ends with calling svc_run(), which starts with activities of joining group.

When svc_run() is called, it first multicasts a Join_Group request to the group that it belongs to. On receiving the Join_Group request, the coordinator of the server group is responsible for transferring current server state to the new joining server and to inform all server replica the new group membership information. To transfer server state to the joining server, server program needs to provide two procedures to be used by the coordinator and joining server respectively. transfer_state() is used to pack the server state in a message and return a pointer to the message. The coordinator call transfer_state() to get server state and sends the state message to the joining server. On receiving the state message, the joining server calls restore_state() to initialize its state.

If there is no currently active server in the group, this replica becomes the coordinator of the server group after several Join_Group requests have been multicast and a fixed period of time has expired. After joining a server group, a server waits on its service port for the arrival of the client's requests.

## 4. Application Examples - Portmapper and NFS Server

This section presents our experience with porting existent non-replicated RPC servers into the DRVM environment using the fault tolerant RPC library described in the paper. The example servers are portmapper and NFS server, which are two important and classical applications using Sun RPC. We ported these existing servers in order to demonstrate the compatibility feature of the libFTRPC and discuss the transparency issues arisen from applying IP multicasting to fault tolerant server group.

If dynamic on-line recovery is a major concern, the state of a server program should be accurately defined for state transfer. When a server attempts to join an existing server group, the current server state needs to be sent to the new joining server, whose state can then be updated correctly before normal operation can proceed. In common case, this server state may consist of only a set of variables, and can be collected, compacted in messages without difficulty, and transferred efficiently. In more complex cases such as NFS server, on the other hand, the whole exported file tree may need to be transferred. This could last as long as several or tens of minutes. The processing of client requests which arrive during the time period of state transfer is deferred until the completion of the state transfer.

### 4.1 Portmapper

As in conventional Sun RPC environment, the portmapper provides a way for clients to look up and find the port numbers of server programs running on DRVM. The portmapper program maps RPC program and version numbers to transport specific port numbers. It makes dynamic binding of remote programs on DRVM possible. The important data structure that portmapper maintains is a list of mappings, called pmaplist. Each of the mappings maps the tuple (program_number, version_number, protocol) for one already existing server to the port number that the server used. This list of mappings comprises the server state of the portmapper.

The following four steps are what we have done to port portmapper of the DRVM environment:

1. Substitute MAIN() for main() in portmapper program.
2. Before svc_run() is called, call read_only(), which is used to declare PMAPPROC_GETPORT, PMAPPROC_DUMP, PMAPPROC_CALLIT as read-only service routines.
3. Define the procedures, transfer_state() and restore_stat(), for state transfer as listed in Figure 3.
4. Link the portmapper program with our libFTRPC library.

In fact, these four steps can be generally used for porting existing RPC servers, not limited to portmapper, to the DRVM environment. Step 2 is optional; it is needed only if there are read-only service routines to be declared

to increase performance.

```
1  void* transfer_state()
2  {
3  static   u_long stat_buf[8*1024/sizeof(u_long)];
4  struct mess_stat *st_msg;
5  XDR      pmlxdr;
6  char      *pmlbuf = (char *)stat_buf+    \
                       sizeof(struct mess_stat);
7  struct pmaplist *pml;
8  int       *addr;
9           st_msg = (struct mess_stat*)&stat_buf[0];
10          st_msg->flags = 0;
11          xdrmem_create(&pmlxdr, pmlbuf,       \
               100*sizeof(struct pmaplist), XDR_ENCODE);
12          if (!xdr_pmaplist(&pmlxdr, &pmaplist)) {
13                  printf(" xdr_pmaplist         \
                       encoding error \n");
14                  exit(0);
15          }
16          st_msg->stat_len = XDR_GETPOS(&pmlxdr);
17          return((void*)stat_buf);
18 }
19 struct pmaplist *newlist =   (struct pmaplist*)0;
20 restore_state(stat_msg)
21 struct mess_stat *stat_msg;
22 {
23 XDR       pmlxdr;
24 char      *pmlbuf = (char *)stat_msg+          \
                  sizeof(struct mess_stat);
25 struct pmaplist *pml;
26          xdrmem_create(&pmlxdr, pmlbuf,              \
                 stat_msg->stat_len, XDR_DECODE);
27          if (!xdr_pmaplist(&pmlxdr, &newlist)) {
28                printf(" xdr_pmaplist   decoding error \n");
29                exit(0);
30          }
31          pmaplist = newlist;
32 }
```

Figure 3. state transfer procedures for portmapper

Figure 3 shows the procedures for state transfer used in portmapper. When a new portmapper is submitted to join an existing portmapper group, the coordinator in the portmapper group will call transfer_state() to obtain the server state. The transfer_stat() routine encodes the pmaplist into XDR format, puts it in a state message using XDR routine for pmaplist (line 12) and returns a pointer to the state message (line 17). The coordinator then sends the state message to the new joining portmapper. Upon receiving the state message, the new joining portmapper calls restore_state() to decode the pmaplist into local host representation and to install the new mapping list as its current server state.

After the program is compiled, the portmapper can be started by the following command

rpc.pmap 236.2.2.2

where rpc.pmap is the name of the executable and 236.2.2.2 is the IP multicast address of DRVM on which the portmapper will be run. If there are any arguments for the server, put them after the IP multicast address. The libFTRPC takes the first argument as the multicast address and associates the service socket, which is created by the server to receive requests, with the IP multicast address on behalf of the submitted server.

From this example, it is obvious that the porting becomes an easy task. The reason for this is that the interface provided by libFTRPC is compatible with that of Sun RPC library.

## 4.2 NFS Server

The Sun Network Filesystem (NFS) protocol provides a collection of remote procedures that allow a client to access files on a server. It is designed to be portable across different machines, operating systems, network architectures, and transport protocols. This portability is achieved through the use of RPC primitives built on top of XDR.[9] The NFS has been widely used in Unix environments to make file sharing easier. Since reliability of file server is important to a computer system, some researches had focused to build reliable NFS servers. Here, we show that the task of building a reliable NFS server is simplified when it is built on top of the libFTRPC.

The source code from user-level Linux NFS version 2.1 is used for our porting. The goal is to replicate a file tree onto a collection of hosts, on which a NFS server group is responsible for providing file service.

The data structure used in the NFS protocol to reference a file is a file handle, which is provided by the NFS server and used by clients to specify which file to operate on. On serving a client request, the NFS server needs to know the exact file corresponding to the file handle in order to do the operation on it. The user-level Linux NFS server version 2.1 uses the i-node numbers, major device numbers and minor device numbers for all directories or files in the path name to encode the file handle. This way, NFS server can convert from the pathname to file handle and vice versa, without difficulty. However, these numbers for various copies of a replicated file are usually different on different hosts. This leads to a problem that the file handle of a replicated file generated on a server cannot be recognized by servers on other hosts even if all the servers have the same replicated file tree. This obviously violates the deterministic assumption.

To have unique, global file handle for a replicated file in NFS server group, we used the pathname of the replicated file to encode its corresponding file handle. The method guarantees that file handles of a replicated file

generated by all NFS servers are the same and that the pathname can be reconstructed from its corresponding file handle if all copies of the replicated file have the same pathname on all servers.

In addition to fault tolerant NFS server group, we also need a mount server in order to allow clients to get the first file handle when clients mount the remote file systems. The mount server return the file handle referencing the mounted directory if the client has access permission to the mounted directory. Of course, this file handle returned by mount server need to be recognized by the NFS server group. The client's mount requests are recorded in a mount list; each entry contains a pair of the client hostname and the mounted directory pathname.

The steps to port mount server are similar to those used for portmapper program. Two procedures of mount protocol, MOUNTPROC_DUMP, MOUNTPROC_EXPORT, can be declared as read-only procedure in step 2. The server state of the mount server is the list of remote mounted filesystems. State transfer procedures listed in Figure 3 are modified to transfer the current mount list from the coordinator to a new joining server in order to meet the requirements of mount server.

Porting NFS server is complicated by transferring all files in exported filesystems from coordinator to a new joining server if on-line recovery is required. This can be achieved by the cooperation of the state transfer procedures, transfer_state() and restore_state(). The transfer time may last several minutes to tens of minutes.

In our current porting of NFS server, we don't support on-line recovery. Instead, we replicate (use Unix commands) a group of NFS server, each with identical file systems to support fault tolerance. This greatly simplifies the porting because the NFS protocol is designed to be stateless. The stateless NFS server need not maintain any protocol state information about any of its clients in order to function correctly.[9] Therefore, no other server state than the state of the exported file system needs to be transferred.

Again, we apply the four steps described above to port NFS server to DRVM environment. The read-only service routines include NFSPROC_GETATTR, NFSPROC_LOOKUP, NFSPROC_READLINK, NFSPROC_READ, NFSPROC_READDIR, and NFSPROC_STATFS. The transfer_state() and restore_state() are made null procedures.

The resulting system described in this section is like the DRVM shown in Figure 2. The mount server and NFS server register their service ports with the DRVM's portmapper. A client of NFS server may look up service ports from portmapper, mount the NFS file system from the mount server and then it can access the replicated files on the NFS server.

From the experience with porting above three RPC servers, we conclude that libFTRPC has great help in developing fault tolerant RPC servers. libFTRPC conceals server replication from server program, so that a server program may be developed as a conventional RPC program rather than a distributed fault tolerant program. Moreover, since most interface of libFTRPC is kept compatible with the interface supported by Sun RPC library, only limited changes need to be made to existing servers for the benefit of fault tolerance.

### 4.3 Performance Measurement of Fault Tolerant NFS server

The fault tolerant RPC mechanism described in this paper has been supported as a library on SunOS, FreeBSD, and Linux. We also had ported portmapper, mount server and NFS server into the DRVM environment on top of the Linux operating system, as described in previous section. We measured the performance of the fault tolerant NFS server. The hardware configuration used to generate the

| | Read 1k bytes per RPC request | Read 2k bytes per RPC request | Read 4k bytes per RPC request | Read 8 k bytes per RPC request |
|---|---|---|---|---|
| non-FT NFS | 50.07 | 38.11 | 26.56 | 22.92 |
| FT NFS server | 50.53 | 38.72 | 27.71 | 23.64 |

Table 1. Times (seconds) for sequentially reading an 8MB file from NFS server and server groups

| | Write 1k bytes per request | Write 2k bytes per request | Write 4k bytes per request | Write8 k bytes per request |
|---|---|---|---|---|
| non-FT NFS | 46.67 | 34.40 | 26.33 | 21.51 |
| 1 server replica | 47.07 | 34.87 | 26.54 | 21.57 |
| 2 server replicas | 64.24 | 45.46 | 33.08 | 28.35 |
| 3 server replicas | 73.58 | 53.22 | 43.31 | 32.48 |

Table 2. Times (seconds) for sequentially writing an 8MB file to NFS server and server groups

performance results was several 90 MHz Pentinum PCs connected by 10Mbps ethernet.

Table 1 and Table 2 show the measurement results. The tests ran with both the NFS servers and the client running at user-level. Since the NFSPROC_READ procedure was declared to be read-only, requests for file reading are served by read-one policy without any server coordination needed. Thus the read throughput is independent of the number of servers in a server group. Nevertheless, requests for NFSPROC_WRITE procedure are served by write-all policy and coordination between server replicas is necessary to guarantee that all server replicas serve all file writing requests in the same order.

## 5. Conclusions

This paper has presented a novel fault tolerant RPC mechanism, which is based on Sun RPC and IP multicast protocol. The mechanism is provided as a RPC library with which a RPC server can be easily made fault tolerant by replicating the server to a group of server replicas. The coordinator-cohort method with read-one/write-all scheme is used as the replication technique of the fault tolerant RPC mechanism.

The use of IP multicast has the advantages of replication transparency and failure transparency. A client of a fault tolerant server group can be shielded from the details in coordination between server replicas. When failures occur in a server group, the client is freed from the need for switch to a new server even in the face of coordinator failures. Besides, using IP multicast also has reduced the overhead incurred from the coordination among server replicas. This transparency aspect has been enable existing client programs to continue to use services provided by fault tolerant server groups without any modification to the client programs.

We have successfully implemented a fault tolerant RPC library, called libFTRPC, on the SunOS, FreeBSD, and Linux operating systems. Since most interface of the fault tolerant RPC library is compatible with the standard Sun RPC, existing server programs based on Sun RPC can benefit from the fault tolerant library with only minor modification being made. We has ported a user level NFS server into its fault tolerant version by using the fault tolerant RPC mechanism described in the paper. Our experience with building the fault tolerant NFS server on libFTRPC has shown that the fault tolerant RPC library can greatly ease the design and implementation of fault tolerant RPC server.

Performance measurements of the reliable NFS server shows that it has the same read throughput as the unreplicated NFS, though the coordination overhead degrades the write performance. Considering that the NFS server group may concurrently serve multiple read-only requests, we believe that it will perform better than the unreplicated NFS server if there are multiple clients making read-only requests at the same time.

## References

[1] Andrew D. Birrell and Bruce J. Nelson. "Implementing Remte Procedure calls," *ACM trans. on Computer Systems*, Vol. 2, No. 1, pp. 39-59, February, 1984.

[2] Luping Liang, Samuel T. Chanson, and Gerald W. Neufeld, "Process Groups and Group Communications: Classifications and Requirements," *IEEE Computer*, Vol 23, No.2, pp. 56-65, February 1995.

[3] S. Deering, "Host Extensions for IP Multicasting," RPC 1054, Stanford University, May 1988.

[4] Hector Garcia-Molina, "Elections in a Distributed Computing System," *IEEE Trans. on Computers*, Vol. c-31, No. 1, pp. 48-59, January 1982.

[5] Fred B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, Vol. 22, No. 4, pp. 299-319, December 1990.

[6] M. Frans Kaashoek and Andrew S. Tanenbaum, "Group Communication in the Amoeba Distributed Operating System," *In Proc. of the Eleventh International Conference on Distributed Computer Systems*, pp. 222-230, May 1991.

[7] Kenneth Birman, Andre Schiper, and Pat Stephenson, "Lightweight causal and Atomic Group Multicast," *ACM Trans. on Computer Systems*, Vol. 9, No. 3, pp.272-314, August 1991.

[8] Sun Microsystems, Inc. RPC: Remote Procedure Call Protocol Specification, Version 2. RFC 1057, June 1988.

[9] Sun Microsystems, Inc. NFS: Network File System Protocol Specification, RFC 1094, Network Information Center, SRI International, March 1989

[10] Jyh-Tzong Chiou, Charles Changli Chin and Shang-Rong Tsai, "A Fault Tolerant RPC Mechanism Based on IP Multicast," *Proc. 1996 Workshop on Distributed System Technologies & Applications*, 1996