KEEPING TCP CONNECTIONS INTACT ACROSS SERVER FAILURES

by

RUI LI

B.E., Tsinghua University, 1995

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standards

The University of British Columbia

December 2000

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date Dec. 17, 2000

Abstract

Replication is widely employed to achieve fault tolerance and high availability. There are two common approaches: active replication and primary-backup replication. In the primary-backup approach, the service states are replicated on the backup server. When the primary server fails, the backup server takes over and continues the service.

In most present implementations based on TCP/IP communication, the TCP connections between clients and servers break if the primary servers crash. The topic of this thesis is to keep the connections intact across primary server failures - the backup server will take all the TCP connections automatically so that, from the client's point of view, no services will be influenced by the failures.

To achieve this goal, we implemented replication in the TCP layer. The information and data associated with the sockets are replicated on the backup server, so when a primary server fails, the backup server can reconstruct all the sockets. By using an ARP message to claim the IP address of the failed primary server, the backup server refreshes the routing tables in other nodes so the packets addressed to the failed primary server will be redirected to the backup server from then on. By this means, the backup server takes over seamlessly, without breaking the present TCP connections.

Table of Contents

Abstract ii

Acknowledgements vii

Table of Contents iii

List of Figures v

List of Tables vi

Chapter 1 1

1.1. Motivation 2

1.2. Design Issues 2

1.3. Thesis Outline 3

Chapter 2 4

2.1. Transactions 4

2.2. Message Logging and Checkpointing 5

2.3. Replication Models 6

2.4. One-copy Serializability 9

2.5. Request Distribution 12

2.6. BSD TCP Implementation 14

2.6.1. Essential Data Structures 15

2.6.2. Connection Establishment 20

2.7. NetVM 21

Chapter 3 23

3.1. Overview 23

3.2. Application Interface 27

3.3. Backup Environment Setup 28
3.4. Synchronization 28
3.4.1. Synchronization between the Application and the TCP Layer 29
3.4.2. Synchronization between the Primary Server and Backup Server 30
3.4.3. Synchronization between Data Replication and Control Block Replication. 31
3.5. Implementation 34
3.5.1. Backup Socket Management 34
3.5.2. Backup Buffer Management 35
3.5.3. Backup Option 36
3.5.4. Replication 37
3.5.5. Failure Detection and Takeover 41
3.6. Limitation 43

Chapter 4 45

4.1. Connection Setup 45
4.2. Data Communication 48
4.3. Takeover 50
4.4. Overall 52

Chapter 5 53

5.1. Conclusions 53
5.2. Future Work 54

BIBLIOGRAPHY 56

List of Figures

Figure 2-1 Replication Models 7
Figure 2-2 Some Variations 8
Figure 2-3 Synchronous update vs. asynchronous update 11
Figure 2-4 Socket Structure 16
Figure 2-5 IP Control Block 16
Figure 2-6 TCP Control Block 18
Figure 2-7 Mbuf Structure 19
Figure 2-8 Connection Establishment 21
Figure 2-9 NetVM Remote Write 22
Figure 3-1 System Architecture 24
Figure 3-2 Backup-Takeover Process 25
Figure 3-3 LKM 26
Figure 3-4 Sending / Receiving process on the primary 32
Figure 3-5 Imported Addresses 38
Figure 3-6 Handle unaligned data 39
Figure 3-7 bckpargs structure 40
Figure 3-8 Structure of Backup Information List 42
Figure 4-1 Connection Setup 47
Figure 4-2 Takeover Time on Different Data Sizes 50
Figure 4-3 Takeover Time on Different Connection Numbers 51

List of Tables

Table 4-1 Connection Establishing Time (in μs) 46
Table 4-2 Time of Communication (in μs) 48
Table 4-3 Overhead on Different Packet Sizes 49

Acknowledgements

Many people have given me help in various ways during my two years study at UBC. First I am greatly indebted to my supervisor, Dr. Norm Hutchinson, for his great guidance on my thesis project. His inspiration and encouragement always stimulated me to seek more knowledge. I am grateful to Dr. Mike Feeley, not only for him being the second reader of my thesis, but also for that he was always ready to offer help on my project. My former advisor, Dr. Son Vuong helped me extend my horizon in the network area. Thanks also to the folk in the Distributed Systems Lab, especially to Joon Suan Ong and Yvonne Coady, for their great help on my thesis.

There are still many fellows who influenced my study at UBC, but I cannot list all of their names. I would like to thank them for being supportive to me.

Finally I would like to thank my parents for always having confidence on me. I hope that they will take pride in me.

Chapter 1

Introduction

With the development of network technology, especially with the booming of the Internet, more and more services are becoming available on the network. Many services demand high availability and fault tolerance. Replication is a widely employed technique to achieve fault tolerance. The state of the service is replicated on several failure-independent servers, so that the service remains available even when a subset of the servers fails. From the client's viewpoint, there is only a single server providing continuous service.

There are generally two approaches to replication. One way is called active (state machine) replication, in which clients update the state on all the servers atomically. Another approach is called passive (primary-backup) replication. In this approach, there is one designated primary server that handles requests from clients, while all the other servers are backups to the primary server. When the primary server fails, one of the backup servers takes over and becomes the new primary server. Passive replication is widely used in commercial products because it is comparatively simpler and has lower overhead on clients.

TCP/IP, as the standard protocol used in the Internet, has become a platform of choice for many services because of the popularity of the Internet. However there is not a standard on how to achieve fault tolerant for services that use TCP/IP.

1.1. Motivation

In most present primary-backup implementations based on TCP/IP, replication is implemented in the application layer. As a result, every application has its own replication implementation. One drawback of implementing replication in the application layer is that when the primary server fails, the TCP connections to the primary server break down. The clients have to reestablish connections with the new primary server after they notice the break of the TCP connections. The clients and the new primary server need to negotiate to continue the service from some checkpoint. For a communication intensive service, this may require the retransmission of lots of data.

The goal of this thesis is to seek a way to keep the TCP connections intact across server failures. Because of the lack of necessary information associated with TCP connections in the application layer, it is impossible to implement replication purely in the application layer. By implementing replication in the TCP layer, we can get another benefit at the same time: providing a general-purpose replication service to applications.

1.2. Design Issues

On one hand, we have the information associated with the TCP connections when we implement replication in the TCP layer. On the other hand, we do not have the information in regard to the data processing in the application layer, which is necessary to reconstruct the receiving queue upon takeover. We provide an API that users may use for checkpointing to solve this problem.

To get the information necessary for recovery, replication must be performed frequently to keep the replicated data on the backup consistent with that on the primary. However, replication operations cause overhead and influence the performance of the system. We follow an event-triggered processing style to replicate data every time it is changed. Choosing the right point to replicate is a key issue in the system design.

Even with all the information needed, there is still a synchronization issue on takeover. The backup application and the TCP implementation must be closely synchronized to ensure that the responsibility for each byte of data is taken by either the application or the system but not both.

To investigate the feasibility of the idea of implementing replication in the TCP layer, we chose FreeBSD as our experimental environment because it is a stable UNIX system and its source code is free. We modified part of the TCP and socket code, so that the control blocks of a socket and the data sent and received through the socket can be replicated on the backup in time. Myrinet is selected as the communication medium between the primary and backup to minimize the communication overhead.

1.3. Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 introduces some background knowledge and compares our work to related work. In Chapter 3 we discuss the design and implementation of our system. Experimental results are given in Chapter 4. Finally we summarize the work accomplished and suggest the future work that may be extended from this thesis.

Chapter 2

Background Knowledge and Related Work

This chapter introduces some background knowledge and some research and techniques related to our work. The first three sections review several common techniques used to achieve fault tolerance. Section 2.5 discusses some implementations of request distribution. Some of them use techniques similar to those we used in our work. The BSD TCP implementation, the base we implement our project on, is introduced in Section 2.6. In the last section, we introduce the communication API we used between primary servers and the backup, NetVM.

2.1. Transactions

The concept of a transaction is widely used in fault-tolerant protocols. Transactions have four essential properties [Tanen95]: atomic, consistent, isolated and durable. Atomicity ensures that a transaction is all-or-nothing: either it happens completely or not at all. Consistency means a transaction does not violate system invariants. The third property is often cited as serializability, meaning the execution of a transaction never appears to overlap the execution of another transaction, and the system behaves as if all

the transactions are executed sequentially in some order. Durability means that once a transaction commits, the results of the transaction become permanent.

A common method of implementing transactions is to log history on stable storage. This method is often coupled with the two-phase commit protocol [Gray78] in distributed systems. In this protocol, the transaction committing involves two phases, preparation and committing. When a transaction is ready to commit, one of the group members initiates the two-phase commit by writing "prepare" in the log and sends the "prepare" message to the other members. All the other members check their states to see if they are ready to commit when they receive the message, and send back their decision. Only when the initiator receives "ready" from all the members could it write "commit" in the log and send the message to the other members. Otherwise the transaction is aborted. The "commit" in the log is the sign and is the only sign that the transaction has committed.

If a computation consists of a sequence of transactions then different forms of redundancy may be employed to achieve fault tolerance. E.g., by using a log, a transaction may be rolled back to the last consistent state and the computation restarted from there if an error is detected. By this means, fault tolerance is accomplished by time redundancy. Another popular technique is to employ physical redundancy to mask the failure of a component.

2.2. Message Logging and Checkpointing

Although transactions are widely used to accomplish fault tolerance, there are some situations in which it is hard to employ this technique. If the service state is determined by its initial states and the sequence of the messages sent and received, a similar technique, message logging, may provide fault tolerance as well. When a server fails, we

can always restore its initial state and roll forward by reapplying the messages saved in the log to reach the state before the failure. Message logging can be either sender-based or receiver-based.

Used alone sometimes [Dimmer85] [PR85], checkpointing is a technique that is often used with message logging. Checkpointing saves the service state on some stable storage, so when a server fails, we may simply restore the last checkpoint and roll forward from there instead of restoring the initial state.

Message logging protocols can be also categorized in two groups: pessimistic protocols [JZ87], which require the messages to be logged synchronously before they are sent out (sender-based) or processed (receiver-based); and optimistic protocols [DG96] [JV87] [SY85], which allow message logging to be performed in parallel with message sending or processing. Optimistic protocols introduce lower overhead but may cause inconsistency. Another approach is causal logging [AM98] [EZ92], which logs in volatile memory and piggybacks the log on every message. Because there is no stable storage involved, the performance is better than that in pessimistic protocols. On the other hand, the system states are consistent. Comparison of different approaches can be found in [AM98] [RAV98].

2.3. Replication Models

In a client-server architecture, physical redundancy is to replicate the service states on multiple sites that fail independently. The clients even do not have to be aware of the existence of the multiple sites, they may use the service as if there is only one server providing the service.

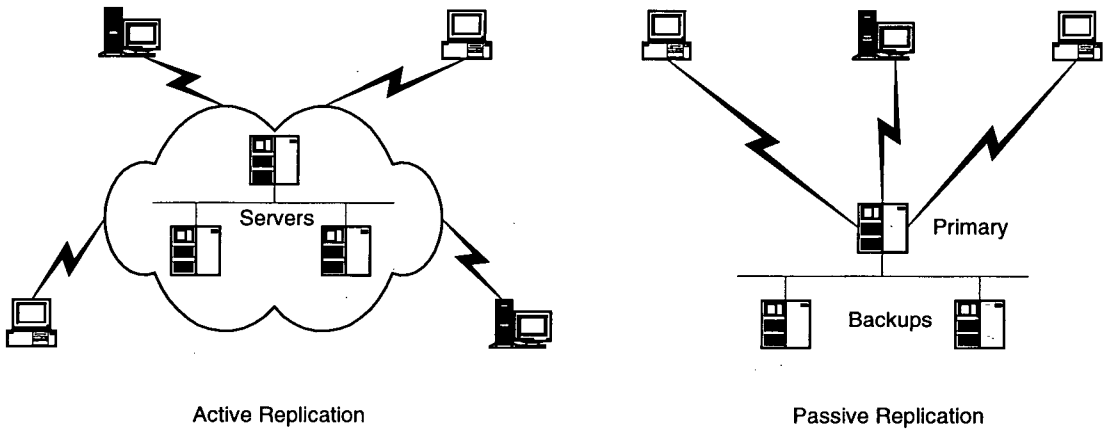


Figure 2-1 Replication Models

There are two widely employed replication approaches: active replication and passive replication. Active replication is also called the state-machine approach, in which all the replicated servers are identical. The service states on all active servers should be updated atomically, so all servers share the same view of the service states. In this way, every server is ready to continue the service if other servers fail. Passive replication, or in another word, primary-backup replication, designates one replica as the primary server, and all the other servers are its backup. In this approach, only the primary server handles requests from clients, and the primary server is normally responsible for updating the service states on backup servers periodically. If the primary server fails, one of the backups takes over and becomes the new primary.

There exist many replication model variations. In some implementations, client requests must be delivered to all the replicas atomically and in-order. The response to clients may be generated by voting [Gifford79]. Some implementations assume that the faults are fail-stop [Schneider83], and allow requests to be handled by only one of the replicas, and only the updates are broadcast among the replicas [LLG92]. There may

exist a front-end to handle client requests and forward them to appropriate servers. Many load balancing switch implementations [AYI97] [DCHKW97] [PABSDZN98] use this architecture. The front-ends may also run at client nodes [LLG92] to choose one of the servers to communicate with. There are also proposals to use location services to decide to which server to send requests [Trian95].

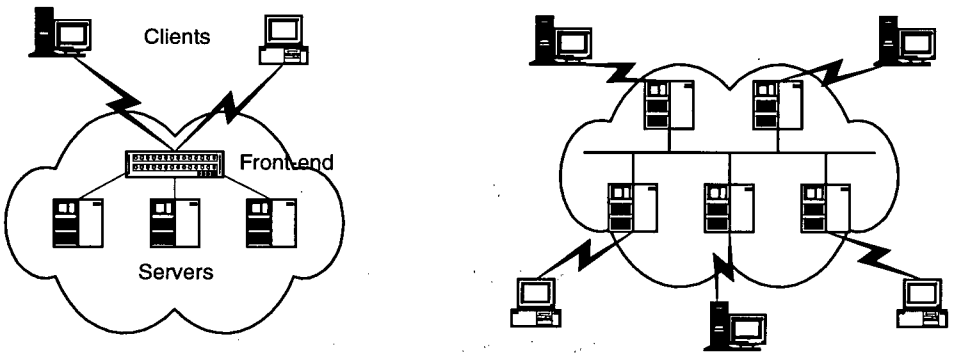


Figure 2-2 Some Variations

Generally in the state machine approach, the overhead associated with atomic ordered delivery of messages tends to slow down the response to client requests. Comparatively, the primary-backup approach has lower overhead and normally needs less resources on backup servers because the services don't have to be active on them. However, this approach tends to require longer recovery time since a backup must explicitly use some algorithm to recover the service according to the replicated states. This may cause a problem in that during the gap between the failure of the primary server and takeover by a backup, some client requests may get lost, which in turn may cause clients to give up. More discussions on this may be found in [BMST92].

Each of the two approaches has its advantage, and both approaches need to replicate service states on multiple servers. The replication, synchronization and associated

communication delay introduce great overhead, and many efforts have been made to reduce it [WB84][OL88][LLG92][RMDJ94].

2.4. One-copy Serializability

One principle of employing the replication technique is to provide clients the view that there is a single server providing service. This is termed as "one-copy serializability" [BG83], which has two aspects. First, the service is one-copy equivalent, i.e., multiple replications of an object appear as a single logical object to clients. Second, transactions may proceed on different replicas, but the system behaves as if all the transactions are executed in some serial order. Corresponding to the two properties, a replication method should include two sets of mechanism: a replica-management protocol to replicate all the data updates on the replicas, and a concurrency-control protocol to control the concurrent data access by multiple clients. In primary-backup protocols, the replication method is comparatively simple because only the primary server handles requests from clients. The concurrency control protocol in this approach is quite similar to that in non-replicated systems. The situation in the state-machine approach is much more complicated in that each replica may be eligible to handle client requests. The replicas must be closely synchronized to keep data consistent.

A simple implementation of one-copy serializability is the read-one-write-all protocol. In this protocol a read operation is executed on any replica, and the correctness of the operation is guaranteed because a write operation must be executed on all the replicas atomically. This protocol observes strict consistency: any read to an object returns the valued stored by the most write operation to the object.

Observing that the availability of write operations in the read-one-write-all protocol is severely restricted, Gifford presents a quorum protocol [Gifford79] in which a read operation involves a *read quorum* number of copies while a write operation involves a *write quorum* number of copies. The sum of the read and write quorums must exceed the total number of replicas. The quorum protocols [AB91] [Herlihy87] sacrifice the availability of read operations to increase that of write operations: write operations do not have to be executed on all replicas, but read operations must be executed on multiple replicas to get the most recent updated results by voting.

Two-phase Locking is a widely used technique for concurrency control in both read-one-write-all and quorum protocols. As the name suggests, two-phase locking includes two phases: a growing phase to acquire locks and a shrinking phase to release locks. A transaction should get all the locks needed during the growing phase. Lock acquiring and lock releasing should not be interleaved during the execution of a transaction. If a process fails to acquire all the locks needed, it may simply release all the locks acquired and try the two-phase locking again later.

Although two-phase locking ensures serializability, it causes high overhead. That is where the optimistic concurrency control [KR81] comes in. The idea behind it is to assume that the possibility of conflicts is fairly low, so everyone may just go ahead and do whatever is needed. If conflicts do occur, influenced transactions may be aborted. In contrast to the two-phase locking protocol, optimistic protocols normally update replicas asynchronously. This approach improves the system performance but may cause inconsistency. This is illustrated in Figure 2-3. Under synchronous updates, the primary sends update messages to backups synchronously when it receives a request from a client,

and it will not send the response to the client until it gets acknowledgement from all the backups. In asynchronous mode, the primary sends back the response at once, and the update message is scheduled some time in the future. The example is given in the primary-backup model, but it applies to the state machine model as well. More discussions on this can be found in [SL95].

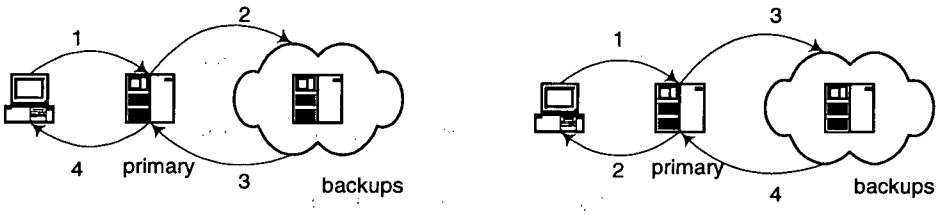


Figure 2-3 Synchronous update vs. asynchronous update

Many researchers have observed that serializability is too strict as a correctness criterion in many situations. A protocol may work well even if the data on replicas are temporally inconsistent, on condition that the inconsistency is bounded and the data will eventually reach consistency. Following this approach, many protocols achieve better performance than protocols observing one-copy serializability [DGP90] [PL91] [KG94] [SL95] [XSSRT96]. Research is also performed on the propagation protocol for weak consistency [PST97]. Rexford et al. employed a similar technique, window-consistency, for primary-backup replication [RMDJ94] [MRJ97].

The location-based paradigm [Trian92] [TT95] combines a quorum protocol with the asynchronous technique. In this protocol, servers are organized as in quorum protocols. In addition, the size of the write quorum should be larger than the half of the total replicas. Read / write operations are similar to those in quorum protocols except that the updating information is sent to location servers. The non-replication-like performance is

achieved by employing "asynchronous replicated lock acquisition". Instead of getting a lock from the number of replicas in the quorum synchronously, a client obtains the lock from a single replica called the leader replica. The leader replica sends back an acknowledgement and broadcasts lock-acquisition messages to other replicas, which in turn will obtain the lock locally. Thus the lock acquisition operation on non-leader replicas is performed in parallel with the client's execution of subsequent operations. Some deadlock-prevention mechanisms need to be employed.

2.5. Request Distribution

One of the replication variants we discussed in Section 2.1 has one front-end handling client requests and passing requests to appropriate servers. This architecture is widely used in the World Wide Web. Many web sites employ identical web servers to process client requests due to workload and fault-tolerance requirements. IBM and CISCO even designed hardware to support request distribution [CISCO] [IBM]. There is also much academic research in this area [AAPPS99] [APB96] [AYI97] [DCHKW97] [PABSDZN98].

Many of these implementations aim at distributing HTTP streams to balance workloads among different servers. SWEB [AYI97] implements this in the HTTP layer by URL redirection [BFF95]. The front end sends back a URL redirection message that includes the IP address of one of the back end servers when it receives an HTTP connection request. The client will set up a new TCP connection to the new IP address upon receiving the URL redirection message. ONE-IP [DCHKW97] presents two low-cost solutions that do not require modifying TCP implementations on servers to accomplish load balancing among servers. All the back end servers share the same IP

address by using address aliases. In the routing-based dispatching, a dispatcher (which may be the router) in the cluster acts as the front end to distribute TCP traffic to different servers according to the IP address of the client. In the broadcast-based dispatching, the client requests are broadcast to all servers, and every server has a filtering routine that filters out undesired packets. In both implementations, the client IP addresses are statically configured to dispatch to different servers, therefore this approach is not so flexible as the dynamic dispatching implementations.

LARD [PABSDZN98] and the Layer 5 switch [AAPPS99] are designed for content-based switching. The front-end in these implementations not only check the workload of the back-end servers but also check the content of an incoming request (mainly URL message [BMM94]) to dispatch it. This approach may achieve high cache hit rate by dispatching similar requests to the same back-end server. The complexity of this approach is that only after the TCP connection has been set up will the front-end be able to check the content of the higher layer requests. LARD implements this by a technique called "TCP handoff". A client sets up a connection to the front-end, which checks the content of the request and chooses one of the back-end servers. The front-end then forwards the established connection to the appropriate back-end server by sending a "handoff" request to it. After accepting the request, the back-end takes over the connection and sends to the client directly*. A technique similar to the TCP handoff, TCP-R (TCP Redirection) [FYT97] was originally designed for mobile computing, but can be used in the handoff situation with little modification. The L5 switching is similar to LARD, but it does not modify the TCP state machine. When it receives the request

* Only packets sent from the back-end to the client are direct, the packets sent from the client to the back-end still need to be forwarded by the front-end.

from a client, instead of sending a special request message, the switch sends a normal TCP connection request to a back-end with the initial sequence number selected by the client. With the two TCP connections, the switch acts as a medium between the client and the back-end. If the implementation is just like this, there is no doubt that the performance will not be good. The magic lies in the port controller on the switch. A classifier at the server port controller on the switch converts the sequence number in every packet from the back-end to the switch to the sequence number used between the switch and client. By this means, the two TCP connections splice and packets may bypass the CPU of the switch and be switched very quickly by the port controllers.

The magicrouter [APB96] is a more general-purpose design, which allows a user-level process to modify every packet passing a device driver. One of the applications of this technique is to change the packet header to redirect it to different destination, as what the port controller in L5 switch does. Shared memory is used to allow the kernel to expose packets to user level processes. The advantage of this approach is that the policy is made by the user process, so it is very flexible. The weakness is that in most systems, the user-kernel communication is slow, so the performance is greatly reduced.

2.6. BSD TCP Implementation

The TCP implementation in BSD UNIX is tightly coupled with the socket implementation. Introduced in 4.2BSD, sockets have become the most popular API for TCP/IP communication. Most TCP implementations in systems originated from BSD UNIX are based on the implementation in 4.4BSD-Lite, and are quite similar. The version of the code described in this thesis is FreeBSD 2.2.2 Release. The TCP

implementation is very complex. Here we can only introduce what is directly related to our implementation. More information can be found in [WS95]

2.6.1. Essential Data Structures

An application can use the socket system call to create a socket which is then used to send or receive data. The system call creates a socket structure inside the kernel, which contains pointers to protocol related data structures and queues for sending and receiving data (so_snd and so_rcv). A listening socket also contains queues for partially completed connections and ready to complete connections.

```
struct socket {
      short so_type;          /* generic type, see socket.h */
      short so_options;       /* from socket call, see socket.h */
      short so_linger;        /* time to linger while closing */
      short so_state;         /* internal state flags SS_*, below */
      caddr_t     so_pcb;                 /* protocol control block */
      struct      protosw *so_proto;      /* protocol handle */
/*
 * Variables for connection queuing.
 * Socket where accepts occur is so_head in all subsidiary sockets.
 * If so_head is 0, socket is not related to an accept.
 * For head socket so_q0 queues partially completed connections,
 * while so_q is a queue of connections ready to be accepted.
 * If a connection is aborted and it has so_head set, then
 * it has to be pulled out of either so_q0 or so_q.
 * We allow connections to queue up based on current queue lengths
 * and limit on number of queued connections for this socket.
 */
      struct      socket *so_head;  /* back pointer to accept socket */
      TAILQ_HEAD(, socket) so_incomp;     /* queue of partial
unaccepted connections */
      TAILQ_HEAD(, socket) so_comp; /* queue of complete unaccepted
connections */
      TAILQ_ENTRY(socket) so_list;  /* list of unaccepted connections
*/
      short so_qlen;          /* number of unaccepted connections */
      short so_incqlen;       /* number of unaccepted incomplete
                                 connections */
      short so_qlimit;        /* max number queued connections */
      short so_timeo;         /* connection timeout */
      u_short     so_error;   /* error affecting connection */
      pid_t so_pgid;          /* pgid for signals */
      u_long      so_oobmark; /* chars to oob mark */
/*
 * Variables for socket buffering.
 */
```

```
        struct      sockbuf {
                u_long      sb_cc;      /* actual chars in buffer */
                u_long      sb_hiwat;   /* max actual char count */
                u_long      sb_mbcnt;   /* chars of mbufs used */
                u_long      sb_mbmax;   /* max chars of mbufs to use */
                long  sb_lowat;   /* low water mark */
                struct      mbuf *sb_mb;      /* the mbuf chain */
                struct      selinfo sb_sel;   /* process selecting
read/write */
                short sb_flags;   /* flags, see below */
                short sb_timeo;   /* timeout for read/write */
        } so_rcv, so_snd;
#define     SB_MAX (256*1024) /* default for max chars in sockbuf */
#define     SB_LOCK     0x01  /* lock on data queue */
#define     SB_WANT     0x02  /* someone is waiting to lock */
#define     SB_WAIT     0x04  /* someone is waiting for data/space */
#define     SB_SEL      0x08  /* someone is selecting */
#define     SB_ASYNC    0x10  /* ASYNC I/O, need signals */
#define     SB_NOTIFY   (SB_WAIT|SB_SEL|SB_ASYNC)
#define     SB_NOINTR   0x40  /* operations not interruptible */

        caddr_t     so_tpcb;    /* Wisc. protocol control block XXX */
        void  (*so_upcall) __P((struct socket *so, caddr_t arg, int
waitf));
        caddr_t     so_upcallarg;     /* Arg for above */
};
```

Figure 2-4 Socket Structure

The field so_pcb in the socket structure is a pointer to a protocol control block, which is the IP control block for Internet domain sockets. The IP control block is used for both TCP and UDP communications. The structure contains a back pointer to the socket structure and a pointer to another protocol control block, e.g., the TCP control block. All

```
struct inpcb {
        LIST_ENTRY(inpcb) inp_list;/* list for all PCBs of this proto */
        LIST_ENTRY(inpcb) inp_hash;         /* hash list */
        struct      inpcbinfo *inp_pcbinfo;
        struct      in_addr inp_faddr;/* foreign host table entry */
        u_short     inp_fport;          /* foreign port */
        struct      in_addr inp_laddr;      /* local host table entry */
        u_short     inp_lport;          /* local port */
        struct      socket *inp_socket;     /* back pointer to socket */
        caddr_t     inp_ppcb;           /* pointer to per-protocol pcb */
        struct      route inp_route;    /* placeholder for routing entry */
        int         inp_flags;          /* generic IP/datagram flags */
        struct      ip inp_ip;  /* header prototype; should have more */
        struct      mbuf *inp_options;      /* IP options */
        struct      ip_moptions *inp_moptions; /* IP multicast options */
};
```

Figure 2-5 IP Control Block

the control blocks of the same protocol are chained in a doubly linked list so the system may locate any of them by searching the list linearly. The control blocks are chained in a hash table as well. For transport demultiplexing, the system can also find a control block quickly by using the hash table. The control blocks with the same hash code are chained in a special list.

```
struct tcpcb {
      struct      tcpiphdr *seg_next;     /* sequencing queue */
      struct      tcpiphdr *seg_prev;
      int   t_state;          /* state of this connection */
      int   t_timer[TCPT_NTIMERS];  /* tcp timers */
      int   t_rxtshift;       /* log(2) of rexmt exp. backoff */
      int   t_rxtcur;         /* current retransmit value */
      int   t_dupacks;        /* consecutive dup acks recd */
      u_int t_maxseg;         /* maximum segment size */
      u_int t_maxopd;         /* mss plus options */
      int   t_force;          /* 1 if forcing out a byte */
      u_int t_flags;
#define     TF_ACKNOW   0x0001  /* ack peer immediately */
#define     TF_DELACK   0x0002  /* ack, but try to delay it */
#define     TF_NODELAY  0x0004  /* don't delay packets to coalesce */
#define     TF_NOOPT    0x0008  /* don't use tcp options */
#define     TF_SENTFIN  0x0010  /* have sent FIN */
#define     TF_REQ_SCALE  0x0020/* have/will request window scaling */
#define     TF_RCVD_SCALE 0x0040/* other side has requested scaling */
#define     TF_REQ_TSTMP  0x0080/* have/will request timestamps */
#define     TF_RCVD_TSTMP 0x0100/* a timestamp was received in SYN */
#define     TF_SACK_PERMIT 0x0200/* other side said I could SACK */
#define TF_NEEDSYN      0x0400  /* send SYN (implicit state) */
#define TF_NEEDFIN      0x0800  /* send FIN (implicit state) */
#define TF_NOPUSH 0x1000        /* don't push */
#define TF_REQ_CC 0x2000        /* have/will request CC */
#define     TF_RCVD_CC  0x4000  /* a CC was received in SYN */
#define TF_SENDCCNEW    0x8000  /* send CCnew instead of CC in SYN */
      struct tcpiphdr *t_template;/* skeletal packet for transmit */
      struct inpcb *t_inpcb;      /* back pointer to internet pcb */
/*
 * The following fields are used as in the protocol specification.
 * See RFC783, Dec. 1981, page 21.
 */
/* send sequence variables */
      tcp_seq     snd_una;          /* send unacknowledged */
      tcp_seq     snd_nxt;          /* send next */
      tcp_seq     snd_up;           /* send urgent pointer */
      tcp_seq     snd_wl1;          /* window update seg seq number */
      tcp_seq     snd_wl2;          /* window update seg ack number */
      tcp_seq     iss;              /* initial send sequence number */
      u_long      snd_wnd;          /* send window */
/* receive sequence variables */
      u_long      rcv_wnd;    /* receive window */
      tcp_seq     rcv_nxt;    /* receive next */
```

```
        tcp_seq     rcv_up;     /* receive urgent pointer */
        tcp_seq     irs;        /* initial receive sequence number */
/*
 * Additional variables for this implementation.
 */
/* receive variables */
        tcp_seq     rcv_adv;    /* advertised window */
/* retransmit variables */
        tcp_seq     snd_max;    /* highest sequence number sent;
                                 * used to recognize retransmits
                                */
/* congestion control (for slow start, source quench, retransmit after
loss) */
        u_long      snd_cwnd;   /* congestion-controlled window */
        u_long      snd_ssthresh;/* snd_cwnd size threshold for
                                 * for slow start exponential to
                                 * linear switch
                                 */
/*
 * transmit timing stuff.  See below for scale of srtt and rttvar.
 * "Variance" is actually smoothed difference.
 */
        u_int t_idle;           /* inactivity time */
        int   t_rtt;            /* round trip time */
        tcp_seq     t_rtseq;    /* sequence number being timed */
        int   t_srtt;           /* smoothed round-trip time */
        int   t_rttvar;         /* variance in round-trip time */
        u_int t_rttmin;         /* minimum rtt allowed */
        u_long      max_sndwnd; /* largest window peer has offered */

/* out-of-band data */
        char  t_oobflags;       /* have some */
        char  t_iobc;           /* input character */
#define     TCPOOB_HAVEDATA   0x01
#define     TCPOOB_HADDATA    0x02
        int   t_softerror;      /* possible error not yet reported */

/* RFC 1323 variables */
        u_char      snd_scale;  /* window scaling for send window */
        u_char      rcv_scale;  /* window scaling for recv window */
        u_char      request_r_scale;  /* pending window scaling */
        u_char      requested_s_scale;
        u_long      ts_recent;        /* timestamp echo data */
        u_long      ts_recent_age;    /* when last updated */
        tcp_seq     last_ack_sent;
/* RFC 1644 variables */
        tcp_cc      cc_send;          /* send connection count */
        tcp_cc      cc_recv;          /* receive connection count */
        u_long      t_duration;       /* connection duration */

/* TUBA stuff */
        caddr_t     t_tuba_pcb; /* next level down pcb for TCP over z */
/* More RTT stuff */
        u_long      t_rttupdated;     /* number of times rtt sampled */
};
```

Figure 2-6 TCP Control Block

Given an IP control block, the corresponding TCP control block can be easily obtained through the pointer in the IP control block. Unlike the IP control block, which mainly contains static information about the communicating peer, the TCP control block contains much dynamic information, which is used to provide the TCP semantics:

```
/* header at beginning of each mbuf: */
struct m_hdr {
      struct      mbuf *mh_next;          /* next buffer in chain */
      struct      mbuf *mh_nextpkt; /* next chain in queue/record */
      caddr_t     mh_data;          /* location of data */
      int   mh_len;                 /* amount of data in this mbuf */
      short mh_type;          /* type of data in this mbuf */
      short mh_flags;         /* flags; see below */
};

/* record/packet header in first mbuf of chain; valid if M_PKTHDR set
*/
struct      pkthdr {
      struct      ifnet *rcvif;           /* rcv interface */
      int   len;              /* total packet length */
};

/* description of external storage mapped into mbuf, valid if M_EXT
set */
struct m_ext {
      caddr_t     ext_buf;    /* start of buffer */
      void  (*ext_free)       /* free routine if not the usual */
            __P((caddr_t, u_int));
      u_int ext_size;         /* size of buffer, for ext_free */
      void  (*ext_ref)        /* add a reference to the ext object */
            __P((caddr_t, u_int));
};

struct mbuf {
      struct      m_hdr m_hdr;
      union {
            struct {
                  struct      pkthdr MH_pkthdr; /* M_PKTHDR set */
                  union {
                        struct      m_ext MH_ext;     /* M_EXT set */
                        char  MH_databuf[MHLEN];
                  } MH_dat;
            } MH;
            char  M_databuf[MLEN];        /* !M_PKTHDR, !M_EXT */
      } M_dat;
};
```

Figure 2-7 Mbuf Structure

reliable communication, flow control and congestion control. The sequence numbers in the structure play important roles in detecting duplicate or lost packets. The TCP control block is a big structure as shown in Figure 2-6.

The mbuf, shown in Figure 2-7, which stands for "memory buffer", is the basic memory management unit for network communications inside the BSD kernel. The sending and receiving queues in the socket structure are organized as mbuf chains. The mbuf provides an easy way to manipulate data buffers, including prepending and appending data to buffers, removing data from buffers, and minimizing the amount of data copied for these operations. The mbuf can be varying-sized, containing up to 108 bytes of data. For more data than that, more than one mbuf can be chained together, or an external buffer of 2048 bytes can be employed. There is a pointer in the mbuf header pointing to the first byte of data, and a field storing the size of the data inside this mbuf. To add more data to an mbuf chain, we may simply append or prepend a new mbuf to the chain. Some mbufs can be deleted from the chain upon removing data. If part of the data in a mbuf needs to be removed, we may simply modify the data pointer to point to the new beginning, if the data is to be removed from the beginning. Otherwise we can modify the size field to reflect the data left in the mbuf, if the data is to be removed from the end. Mbufs are not only used for storing data, but also for IP address, port number and protocol control information.

2.6.2. Connection Establishment

The TCP connection establishment procedure is known as a three-way-handshake. In normal operations, a client (the active open side) sends a SYN to the server (the passive open side). The server in the "listen" state sends back a packet containing a SYN and an

ACK for the SYN from the client. Upon receiving the packet, the client sends back an ACK for the SYN from the server, which may be piggybacked on the first data packet or on the following data packet, and enters the "established" state. The server enters the same state after it receives the ACK. The normal procedure on the server side is shown in the Figure2-8.

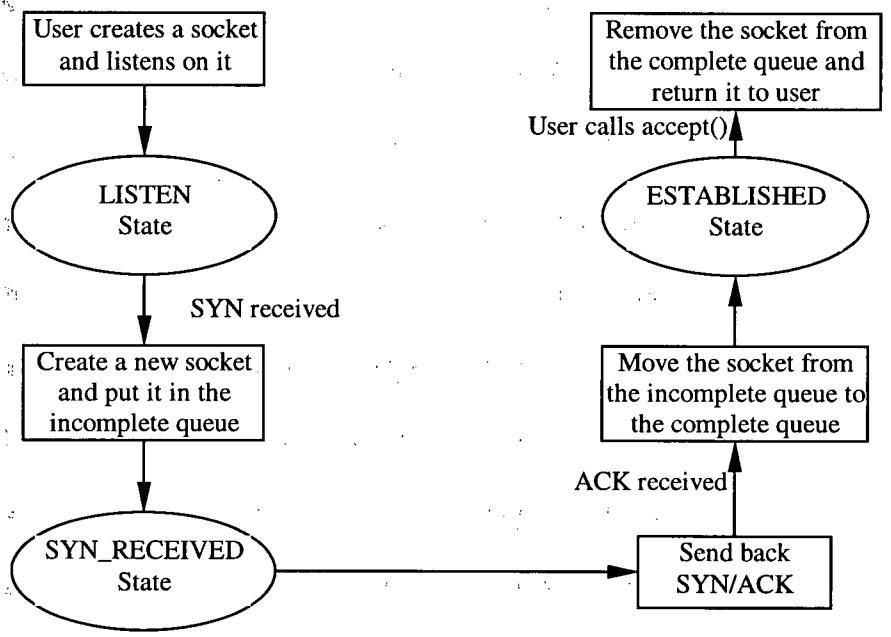


Figure 2-8 Connection Establishment

2.7. NetVM

NetVM is an API on the MyriNet Gigabit Network Interface developed by Joon Suan Ong at UBC. With NetVM, a process is able to read and write memory on a remote node directly. NetVM was originally designed for user level communication, so all the functions could only be invoked in user space. Ong modified it to fit with our requirement. The only function used in our project is the remote write function using physical addresses. The function interface is shown in Figure 2-9.

```
unsigned int
lkm_remote_write_phy(void *p,  /* local buffer pointer */
      unsigned int pa,         /* correspondent physical address */
      int len,                 /* buffer size */
      Tprot_key key,           /* local access key, unused in kernel */
      int rem_node,            /* MyriNet node id */
      int rem_id,              /* block id, unused */
      void *rem_p,             /* remote buffer pointer */
      unsigned int rem_pa,     /* remote buffer physical address */
      Tprot_key rem_key,       /* remote access key, unused */
      int flags                /* synchronization flag */
);
```

Figure 2-9 NetVM Remote Write

This kernel version remote write borrows the interface from the one for user level communication, so there are many unused parameters, which are used to enforce security in the user level version.

There are some constraints on using NetVM. First, like most communication APIs based on DMA, the data to be transferred must be DMAable. On IBM PCs, this means both the address and the length of the data must be multiples of 4. Second, the data cannot cross a memory page boundary. Normally the memory page size on PCs is 4096 bytes. Third, both the virtual memory address and the physical address must be used to achieve best performance. And last, the present version of NetVM does not provide a notification mechanism on data arrival events. An alternative message passing API implemented by Mricom, GM [Myr99] does provide notification but does not support remote memory operations.

Despite all these constraints, NetVM provides a convenient communication mechanism with low overhead. Without using explicit messages, NetVM releases the user process on the remote node from handling messages.

Chapter 3

Design and Implementation

The goal of our system is to provide a general tool for various applications to attain fault tolerance. The system must have these properties:

- Transparent to clients.
- Low overhead.
- Little involvement of user applications.
- Easy-to-use interface.

The failure that our design addresses is the failure of a server. A server may fail, but we assume the failure is fail-stop. We do not try to handle network partitions or Byzantine failures.

3.1. Overview

As shown in Figure 3-1, we follow the primary-backup approach. Similar to [Barlett87], every primary server only has one backup in our model. This decision is based on the fact that the failure possibility of the primary server is fairly low and our implementation is not designed to handle Byzantine failure. A benefit of this approach is that it avoids synchronization among backup servers. Different from [Barlett87], several primary servers may share the same backup.

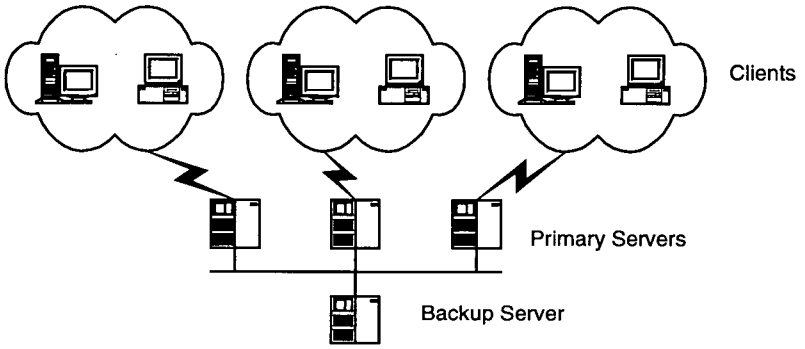


Figure 3-1 System Architecture

Applications using this system achieve fault tolerance through a way quite similar to message logging protocols. The difference is that the message and checkpoint logs are not saved on some stable storage, but on a backup server. Instead of rebooting the server and restarting the applications on it after a server crashes, the backup server takes over and continues providing the services.

Figure 3-2 shows the normal scenario of the working process of the backup system, which comprises of 3 phases. In the setup phase, a primary begins with creating a listening socket and notifies the backup, which in turn creates a backup environment for the socket and returns the information to the primary. Every time it accepts a connection request from a client, the primary creates a new socket and creates a new back environment on the backup. The second phase is the communication phase, in which the primary works similar to normal TCP servers, except that it replicates every packet received and sent and the control blocks of the communication sockets on the backup. The primary also sets a heartbeat flag on the backup to indicate that it is still alive. At the same time, the application checkpoints periodically. The takeover phase begins when the primary crashes. The backup assumes that the primary has failed after there has been no

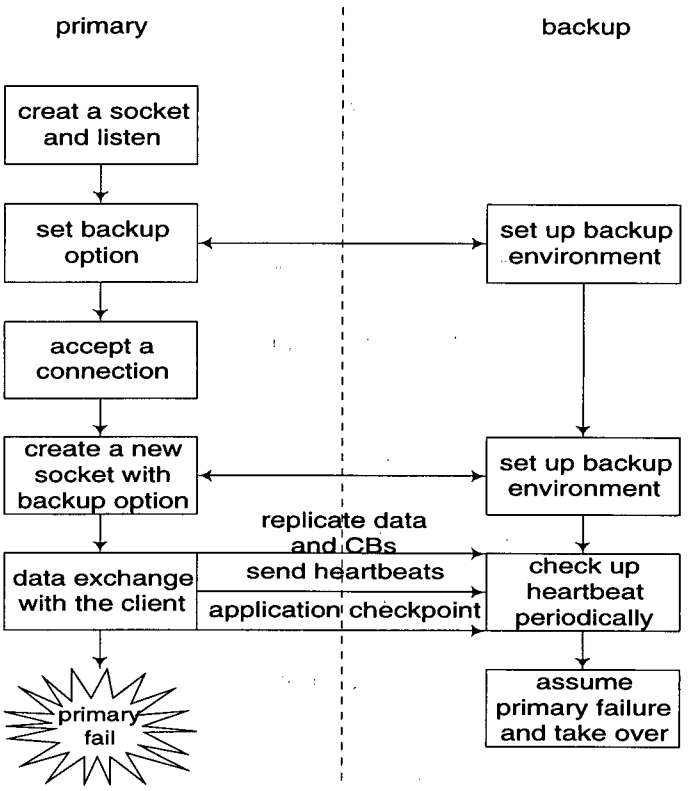


Figure 3-2 Backup-Takeover Process

heartbeat signal from it for some time, and takes over. The sockets are reconstructed with the data received and sent since the last checkpoint. An application is executed to use these sockets to resume communication with the clients.

There are four modules implemented on the primary server. The first one is the environment-setup module which sends requests to the backup server to set up the backup environment and records configuration information concerning these connections. Its counterpart, the release module, notifies the backup server to free the resources allocated, and releases related local resources. The replication module is responsible for replicating the incoming and outgoing data and related control information on the backup side. The last module is a timer that generates heartbeat signals periodically to tell the backup that it is still alive.

Referring to Figure 3-2, the system on the backup server is divided into two parts: a backup daemon running in user space and a LKM (Loadable Kernel Module) running in the system kernel which provides a system call. The backup daemon has two main tasks. The first one is to invoke the backup system call, which loops, processing backup requests from the primaries. Second, the backup daemon executes appropriate applications to take over when the LKM detects a primary server failure and forks a new process to return to the user space. The LKM sets up the backup environment for a socket on a primary when it receives a backup request and destroys the environment when it

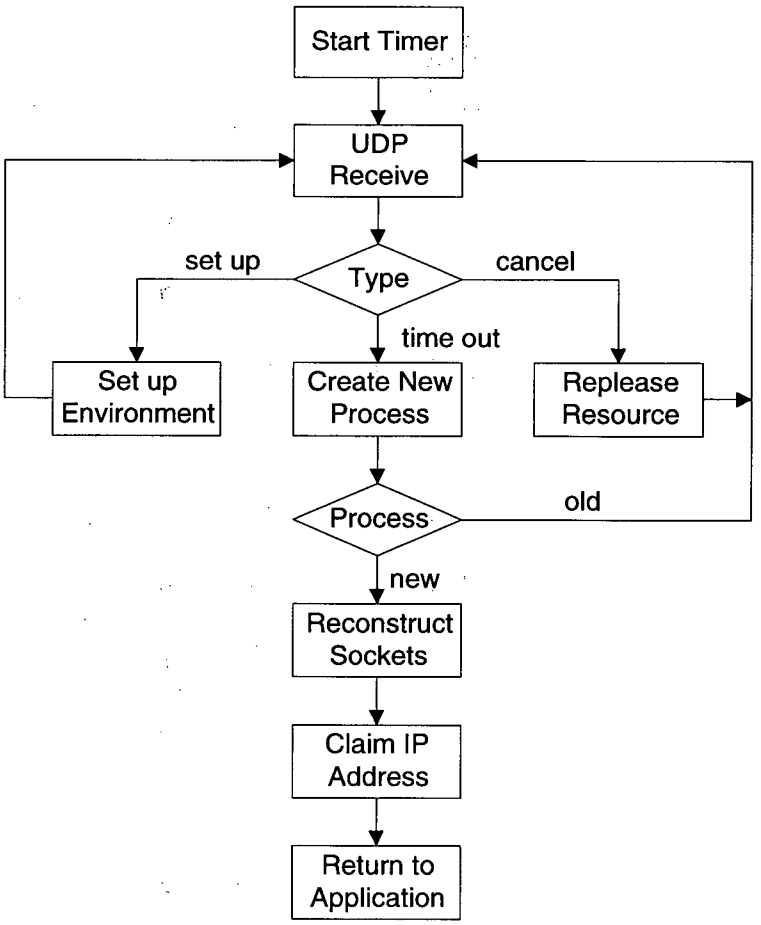


Figure 3-3 LKM

receives a cancellation request. A timer checks the heartbeats from the primary servers periodically and notifies the system of the primary failure event if there is no heartbeat from a primary server for some period. The process forks on fail-over events. The original process keeps on looping, while the new process handles the takeover, including claiming the IP address of the failed primary, recovering all the sockets backed up for the failed primary and creating file descriptors for the sockets to return to the backup daemon. The daemon executes appropriate applications and passes the corresponding file descriptors to them.

3.2. Application Interface

The system is running on the server side and is transparent to clients, so there is no need to modify the code on the client side at all. On the server side, the system replicates all application TCP messages on the backup server. The state of an application is determined by its initial state and the TCP messages that it has received and sent.

The system also provides an easy way to checkpoint an application. By this means, an application can write a checkpoint log on the backup server. When the primary server fails, the application can restart from the last checkpoint on the backup server instead of restarting from the initial state. The requirement is that the application must be able to recover according to the checkpoint. The system can reconstruct all the sockets used on the failed primary server and pass them to the application. The data received and sent since the last checkpoint can be recovered as well.

In summary, to use the system, an application should:

- On the primary server:
    - Set the backup option

- Checkpoint

- On the backup server:
  - Be able to recover from the checkpointed state.

3.3. Backup Environment Setup

The system provides the backup function as an option to applications. The socket is selected as the basic unit to set this option. An application can use the backup function by setting the backup option on a socket. For a socket with the backup option, the system creates the backup environment on the backup server for the socket, including creating a socket and related data structures, allocating data buffers to replicate incoming and outgoing packets, and allocating a log buffer to store checkpoint information. Every socket with the backup option has a corresponding socket for it on the backup server. Sockets without the backup option are processed as usual.

For a listening socket with the backup option set, all the sockets created for connections accepted by it inherit the backup option automatically. This is suitable for most applications based on TCP. The listening socket does not transfer any data, it is only involved in handling connection-establishing requests. It is the sockets created through the listening socket, not the listening socket itself, that are used to exchange user data.

3.4. Synchronization

To keep the service states on the primary server and the backup server consistent, the system must be well synchronized. There are several synchronization issues here. First, the application and TCP layer must share the same view on data upon takeover, i.e., each byte of data is either handed by the application or by the kernel, but not both. Second, replicated data in the TCP layer on the backup server must be consistent with that on the

primary server, so that the application gets the same flow of data no matter which server it communicates with. Third, on the backup server, the TCP state must be consistent with the data replicated.

3.4.1. Synchronization between the Application and the TCP Layer

Message replication is done in the TCP layer. To recover from a failure faster, the application server must have some mechanism, e.g., using a checkpoint log to record the service state periodically. The TCP layer is unaware of the service state in the application layer, so checkpointing must be done in the application layer. Thus we have two logs handled by different layers.

The aim of the system is to make the failure of a server transparent to clients, so we cannot expect client applications to contribute to failure recovery. Everything should be done on the server side. Because data is replicated on the server side, the system must ensure that every TCP message is handled "exactly once".

For incoming data, this means that every packet must be handled by the application exactly once. When the backup server takes over and restarts from the last checkpoint, the TCP layer is responsible for passing all the packets sent or received since the last checkpoint to the newly restarted application. The client application is oblivious to the failure and will neither restart from the checkpoint nor resend these packets. For outgoing data, there must be some mechanism to prevent the restarted application from resending the data that has been sent by the failed primary before it failed.

The synchronization between the TCP layer and the application layer is achieved by using sequence numbers. TCP is a stream oriented reliable communication protocol, in which sequence numbers are exploited to detect lost and duplicated data. This makes the

sequence number a good candidate to log the transaction history. The only requirement is that the server application also uses sequence numbers to log transactions. By this means, the takeover server application may notify the TCP layer about the sequence numbers of the last bytes it sent and received, which can be retrieved from the checkpoint log. The TCP layer will then remove the inconsistent data from the sending and receiving queues. By doing this, it is guaranteed that the application will receive from the last byte it received and there will be no duplicate data in the sending queue.

3.4.2. Synchronization between the Primary Server and Backup Server

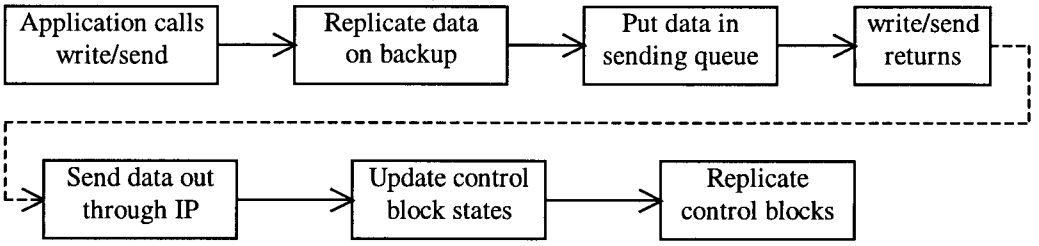
To ensure that the system may recover from primary server failure at any point, the data on the backup server and the primary server must be consistent. When a data packet arrives from the network, the device driver is invoked at the interrupt level. The IP layer then calls the appropriate transport layer routine to handle it according to the packet type. For a TCP packet, the TCP layer checks the packet header, and if it is the expected packet puts it in the receiving queue waiting for the application to fetch the data. An acknowledgement for this packet is arranged to be sent back some time in the future. Normally the application retrieves the data asynchronously, and the packet is deleted from the queue after that. When the acknowledgement arrives, the sender deletes the corresponding packet from the sending queue and will not try to send it again. If the primary crashes right after it sends back the acknowledgement for an un-replicated packet, the sender of that packet will think the packet has been received and remove it from the sending queue. When the backup takes over, there will be no way for it to get this packet again. From this analysis it is obvious that an incoming packet must be replicated not only before the data is retrieved by the application, but also before the

acknowledgment is sent back. Replicating data before it is put in the receiving queue satisfies these requirements.

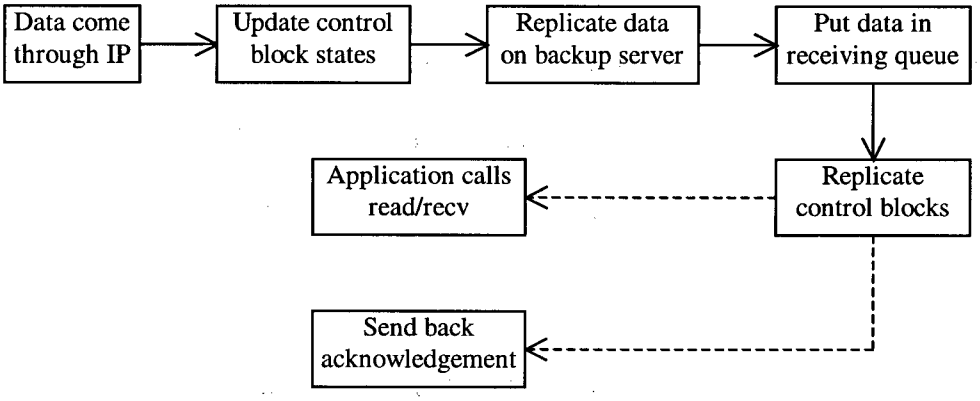
Without the backup system, the socket layer merely puts the packet in the sending queue when an application sends a packet through a socket. The TCP layer will send the packet out through the IP layer at some convenient time. Because the send routine returns to the application immediately after the packet is put in the queue, the application may write the checkpoint before the packet has been sent. It is necessary to replicate the data before the send routine returns or delay the checkpoint until the acknowledgement of the packet has been received, so that the backup server may send out the packet if the primary server fails in such a scenario.

3.4.3. Synchronization between Data Replication and Control Block Replication

The TCP control block plays an important role in providing reliable communication. More specifically, the sequence numbers stored in the control block are the key to preventing duplicate or missing data. Because the replicated sockets on the backup server are not really communicating with clients, the TCP layer will not update their states automatically. In our model, the TCP control blocks on the backup server are just copies of those on the primary server. This causes a potential problem in that the replicated data and control block may be inconsistent on the backup.



Sending data



Receiving data

Figure 3-4 Sending / Receiving process on the primary

One way to guarantee that the data and control blocks are consistent is to replicate data and the corresponding control block atomically. Extra effort would be needed if we made the data replicating and control block replicating atomic. We followed another approach -- simply replicating them separately. The correctness of this approach is guaranteed by the TCP functionality of providing reliable communication. It may happen that the control block states and the data states on the backup server are not consistent. However, when we reconstruct sockets on the backup server, we create data queues according to the control blocks. By doing so, the possibility of inconsistency is eliminated and at the same time, no necessary information is lost. Consider the two scenarios of replicating data on the primary in Figure 3-3.

In the first scenario, there are two points where failure may cause inconsistency. First, the primary fails after the data has been replicated on the backup server, but not put in the queue or has been put in the sending queue but the send system call has not returned. Because the primary fails before the send system call returns, the application on the primary will not log this packet as "sent". After takeover, the restarted application on the backup server will re-send the data. If there is no special mechanism here, then the data will be appended to the end of the sending queue, which has included the same data. This inconsistency between the application and TCP layer has been discussed in Section 3.4.1.

Second, if the primary server fails when some packets have been sent out, but the control block on the backup server has not been updated, then the control block states are inconsistent with the service states. However, this is well handled by TCP itself. According to the stale control block on the backup server, these packets have not been sent yet, so they will be re-sent. The duplicate packets will be detected on the client side and discarded.

Similarly, there exists an inconsistent point in the second scenario. If the primary server fails between the data replication and control block replication, then the states on the backup server are inconsistent. There are more data in the backup buffer than what the control block shows. Neither the primary server nor the backup server will send acknowledgement for this received packet, so the client will re-send it. If the backup server copies the data from the backup buffer to the receiving queue as it is, then there will be duplicate data in the receiving queue. The solution is that by checking the

"rcv_nxt" sequence number in the control block, only data consistent with the control block will be copied to the receiving queue.

3.5. Implementation

We implemented the system on IBM PCs running FreeBSD 2.2.2-Release. The computers are connected through MyriNet. On the primary server side, we modified part of the socket and TCP code to support the replication needed, since the replication operation closely interacts with the present TCP implementation. It is almost impossible to implement the replication as a Loadable Kernel Module (LKM). On the other hand, the module on the backup server is comparatively independent of other parts of the kernel, which allows us to implement it as a LKM. Implemented as an LKM, the system is more portable. A computer can begin to provide the backup service without rebooting.

3.5.1. Backup Socket Management

As introduced in Chapter 2, IP control blocks are linked in a list in BSD systems. TCP timers search the list to deliver TCP timeout events. The system also uses a hashtable to accelerate demultiplexing. Our system creates a corresponding backup socket on the backup for each socket on a primary that needs to be backed up. If we use the standard routine to create backup sockets, the IP control blocks of these sockets will be linked in the system list and the system hashtable. This may cause the backup server to handle the backup sockets mistakenly, e.g., trying to send out unacknowledged packets, accepting packets oriented to a primary server, etc.

We solve this problem by creating a backup list and a hashtable on the backup using the same data structures as the BSD system kernel does. Instead of being linked to the

system list and hashtable, the IP control block of a backup socket is linked to the backup ones until the takeover.

3.5.2. Backup Buffer Management

For each socket to be backed up, the backup server creates a backup socket and allocates two data buffers for it, one for the outgoing data, and the other for the incoming data. Unlike the sending or receiving queues in the BSD TCP implementation, which are organized as mbuf linked lists, the buffers are allocated as continuous memory regions at the time the backup environment is set up. This approach avoids the overhead of allocating a small piece of memory every time some data needs to be replicated, but has the risk of wasting resources. The buffers are organized as fixed-sized circular lists, and it is the applications' responsibility to use them wisely. If there is too much data transmitted between two checkpoints, old data in the buffers may be overwritten by new data. The application must checkpoint in time to avoid data loss.

Two auxiliary buffers are used to keep track of data boundaries in the sending buffer and the receiving buffer respectively. The contents of the buffers are pairs of sequence numbers and data pointers (see Figure 3-3), which are used to record the address of the data beginning from the specific sequence number. The length of the data can be obtained by the difference of two sequence numbers of neighboring pairs. Like the data buffers, these buffers are also organized as circular lists. The reason for using these buffers is explained in Section 3.5.4.

In BSD systems, the sending/receiving queues are organized as mbuf lists, so the backup server needs to copy the data from the buffers to create the mbuf lists when it takes over. If the data is stored in mbuf lists when it is replicated, it will be faster to take

over. However, this approach introduces high overhead to data replication, and as a result, influences the system performance. Moreover, mbufs are managed by the kernel specially. Remote writing to mbufs is not really possible.

3.5.3. Backup Option

Our system provides applications the backup functionality as an option. If an application sets the option, our system will back up the indicated socket on the backup server. The socket is the basic unit for replication, so we reuse the socket option function in the operating system instead of providing a new routine. The socket option setting routine is pretty flexible. It may be used to set options for the socket layer, the TCP layer or the IP layer, which normally modifies socket data structures, TCP control blocks or IP control blocks respectively. Some socket options are just used to set flags in the so_options field of socket structures, whereas other socket options and TCP, IP options may be used to set or get system configurations. The setsockopt() system call provides an easy way to pass a variable length parameter.

Because our system is designed for TCP communication, we implement the backup option in the TCP layer. An application needs to use level IPPROTO_TCP to call setsockopt() so that the option is handled correctly in the TCP layer. Setsockopt() can be used to set or cancel the backup option. To set the option, the application should pass a structure containing the IP address and port number of the backup server as a parameter, which will be used to indicate the backup server to send requests to. This information is saved in the kernel for later use, e.g., to cancel the option.

Sockets created by the "accept" system call should inherit the backup option automatically so that applications do not need to set the option for every accepted socket.

This is done in two steps. As shown in Figure 2-8, a new socket is created when the listening socket receives a SYN packet. The pointer to the backup server information is taken as a flag that the socket needs to be backed up. This pointer is copied to the newly created socket, so the new socket will be backed up on the same backup server as the listening socket is. The second step, setting up the backup environment, is done in the accept() system call. The TCP layer follows the same steps to set up the backup environment as those used in setting the backup option. Because tcp_input, the routine handling the SYN packet, is executed at interrupt level, whereas UDP is used in setting up backup environments, it is hard to implement the setup inside tcp_input.

3.5.4. Replication

Replication includes data replication and control block replication. Both of them are implemented by using NetVM. The reason we chose NetVM is that considering the large amount of data to be replicated, we tried to minimize the overhead of replication. Myrinet provides a fast communication platform and NetVM is a convenient communication API available to us on Myrinet. The IP control block, which contains the static information of a connection, is replicated only once, right after the backup environment is set up. The TCP control block, which has fields that change with every data transmission, is updated frequently. Both the virtual address and the physical address of a remote buffer are needed to use NetVM. A primary server imports the addresses from the backup server when the backup environment is established.

```
struct bkpaddrs{
    void    *snd_buf;    /* virtual address of the sending buffer */
    int     snd_buf_pa;  /* physical address */
    void    *snd_pnt;    /* address of the pointer buffer for sending */
    int     snd_pnt_pa;  /* physical address */
    void    *rcv_buf;    /* address of the receiving buffer */
    int     rcv_buf_pa;  /* physical address */
    void    *rcv_pnt;    /* addresses of the pointer buffer */
    int     rcv_pnt_pa;
    struct  tcpcb *tcb;  /* addresses of the TCP control block */
    int     tcb_pa;
    struct  inpcb *inp;  /* addresses of the IP control block */
    int     inp_pa;
    void    *log;        /* addresses of log */
    int     log_pa;
};
```

Figure 3-5 Imported Addresses

As introduced in Chapter 2, NetVM uses DMA to achieve fast data transmission. This requires that messages be word aligned. Normally variable addresses satisfy the alignment requirement of DMA, but the length of the data sent by user applications does not always fit with this criterion. This causes a problem when we use NetVM to replicate data. We solve this problem by copying extra data. For example: suppose that we need to copy 22 bytes of data whose offset from the beginning of a memory page is 0x0023. Instead of copying the exact 22 bytes of data, we copy 28 bytes of data from the offset 0x0020. The operation satisfies the "DMAable" requirement of NetVM. Now the problem is how to notify the backup server about this so that the padding will not be taken as normal data by the backup server.

Since NetVM simply writes data to a remote node without notification, the only possible solution is either send a notification to the backup server or use another remote write to log this on the backup server. Because the backup server does not care about the replicated data until a primary server fails, we follow the second approach. That is why we use two auxiliary pointer buffers to store this information.

Still using the above example and assuming that the 22 bytes data starts with sequence number 1: Figure 3-5 shows that extra bytes are copied from the primary server to the backup server. We can find the address of the real data from the pointer buffers, and by reading the next sequence number we know that only 23-1=22 bytes data in this segment are valid.

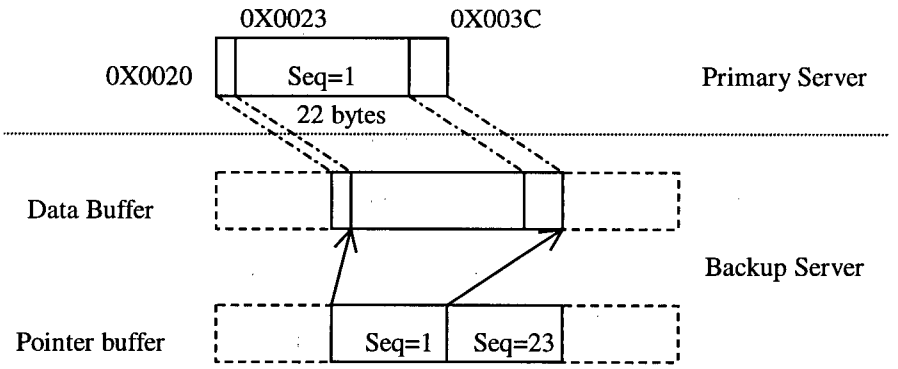


Figure 3-6 Handle unaligned data

There are still several problems to deal with. First, the data in sending/receiving queue is stored in mbuf lists instead of continuous buffers. Second, NetVM cannot cross 4KB page boundaries. Third, both the data buffer and pointer buffer are organized as circular lists. Special handling is needed when the end of buffer is reached. Furthermore, we want to use as few remote writes as possible. Otherwise, remote writes may introduce high overhead and consume pointer buffers very quickly.

The size of an mbuf is smaller than 4K bytes, so an mbuf is always allocated within one memory page [MBKQ96]. Thus the read operation on a primary server when we use NetVM does not cross the page boundary. The buffers on the backup server are allocated as multiples of 4K, so the end of a buffer is also the end of a page. The system uses the remote write function in NetVM to replicate the mbufs one by one. Each time an mbuf is processed, the page boundary on the backup server is checked. If the data size in an mbuf

is larger than the space left in the current page on the backup server, the data needs to be copied in two sections.

The writing of the pointer buffer is delayed until all the mbufs in a queue have been replicated or unaligned data is reached. This implementation saves many remote writes on aligned data. The backup server reconstructs the sending and receiving queues according to the pointer buffers, so if the primary server fails after some data is replicated but the corresponding pointer is not set, the data will not be copied upon takeover. The writing of sequence number and pointer pairs is overlapped. Every time a bunch of data is replicated in a remote buffer, the beginning sequence number and starting address and the ending sequence number and ending address are written in the corresponding pointer buffer, and the first pair overwrites the second pair written last time. By this means, the backup server may get the size of the last bunch of data while no extra space is used to save the data size.

A bckpargs structure (Figure 3-6) is used on the primary server to record the offsets of the next byte to write in different buffers. An offset may be round up if the end of the corresponding buffer is reached. The replication operation is fully executed on the primary server side.

```
struct bkpargs{
    struct bkpaddrs bckpaddrs;/* buffer addresses on the backup */
    u_long snd_off;           /* offset in sending buffer */
    u_long snd_pnt_off;       /* offset in sending pointer buffer */
    u_long rcv_off;           /* offset in receiving buffer */
    u_long rcv_pnt_off;       /* offset in receiving pointer buffer */
    u_long log_off;           /* offset in log */
};
```

Figure 3-7 bckpargs structure

3.5.5. Failure Detection and Takeover

We simply use heartbeat signals to detect the failure of primary servers. A timer on the primary server sets a flag on the backup server using NetVM every 0.2 seconds, and a timer on the backup server checks the flags every one second. If a flag has not been set within 1 second, the backup server will consider the primary server to have failed and will take over. The timers on the primary server and backup server are implemented in different ways. On the primary server, we use the TCP timer directly. In the BSD implementation, TCP uses two timer routines to check on TCP timeout events. One is called the "quick timer", which interrupts every 0.2 seconds; the other is the "slow timer", which interrupts every 0.5 seconds. We use the quick timer to set the flag on the backup server. On the backup server, we use the "timeout" function inside the kernel to schedule a routine that checks on the "alive" flags. The flags are stored in a linked list. The routine checks and clears all the flags every 1 second. This allows up to four heartbeat signals to be lost before a failure is signaled.

If a flag is not set since the last timeout event, the timer appends a special packet to the receiving queue of the blocked UDP socket which is waiting for the requests from primary servers and wakes up the socket. We implement it this way because the timeout is an interrupt event, which is not related to any process, whereas we need the process information to create file descriptors for the sockets to be recovered. Other alternatives were also considered. One choice is to send a signal to the blocked process, which will interrupt the blocked socket from waiting and cause the system call to return. The application must use another backup system call to reenter the kernel and again wait for a request. Another alternative is to use two processes inside the kernel, one of which is

used to handle requests, the other just sleeps until the timer routine sends a signal to it when a primary server failures. Both of the approaches are more complicated than the first alternative that we have adopted.

When the blocked socket is woken up, the system may find the failure of a primary server from the received special packet, which contains the IP address of the failed server. A new process is created to handle this takeover, and the old process loops back and blocks on the UDP receive waiting for new requests.

The newly created takeover process first searches the backup socket list (see Figure 3-7) for IP control blocks corresponding to the failed primary server according to the IP address. For every socket found, the replicated data is copied from backup buffers to the sending and receiving queues. Incoming data between the last checkpoint and rcv_nxt is copied to the sending queue. For incoming data, the snd_nxt is used as the start point, and the last sequence number in the pointer buffer is taken as the end mark.

```
struct pntlnk{
    char    *snd_buf, *snd_pnt;/* buffer pointers */
    char    *rcv_buf, *rcv_pnt;
    struct  inpcb *ip;          /* corresponding IP control block */
    struct  pntlnk *next;       /* next backup socket in chain */
};
```

Figure 3-8 Structure of Backup Information List

All corresponding IP control blocks are moved from the backup control block list to the system control block list so that they may accept incoming packets. The system creates a file descriptor for every socket associated with the new process, which will be directly used by the application to send and receive data. The backup server adds the IP address of the failed primary server to the network interface as an alias, which causes an ARP message to be broadcast in the local network. Every node in the local network, including the router, will update their corresponding routing entry accordingly. All the

messages addressed to that IP address will then be redirected to the backup server. Finally, the new process returns from the system call to the daemon with all the file descriptors. Then the backup daemon executes a special version of the corresponding server application, which can recover according to the checkpoint log, and passes all the file descriptors to it, and the server application may continue the service from then on. Our system passes the file descriptors to the application as command line arguments. This requires the application be able to get the file descriptors by this means and continue communication with them.

Before it uses these newly created sockets to communicate with the clients, the restarted server application should retrieve the log for each socket through a system call and continue from the last checkpoint. It also needs to synchronize with the TCP layer through a special system call. The TCP layer gets the sequence number of the next byte to send by this means, so it may remove all the inconsistent data from the queue.

3.6. Limitation

The system is designed to support multiple primary servers with multiple TCP connections. However, there are some factors constraining its scalability. First, the backup buffers are allocated inside the kernel when the backup environment is set up. This approach avoids the overhead of allocating memory every time when a packet is replicated, but limits the amount of the pending data. Currently the sending buffer and the receiving buffer are both 128K bytes. Considering that a TCP packet can be as big as 64K bytes, the buffer is really not so big. However, even with such a small buffer size, the backup server still runs out of memory quickly. The reason is that in our present implementation, the buffer is allocated from a submap inside the kernel. This is the

standard behavior of the kernel malloc() of BSD Unix [MBKQ96]. The default size of the submap is about 19M in our system. The total buffer size allocated for a backup socket is about 400K, so the maximum number of the living sockets that the backup server can handle is about 40. Even without the constraint of the submap, the total number of sockets that can be handled is still limited by the physical memory size. This is because the buffers are allocated inside the kernel as unpaged memory, which cannot be swapped out.

Second, the throughput of the system is constrained by the communication capability of NetVM. NetVM puts sending messages in a queue before sending them through Myrinet. When the queue is full, no more messages can be put in the queue. In our testing, when there are 3 clients sending messages without pause, the NetVM begins reporting queuing errors.

Besides these constraints, checkpointing in our system is only partially implemented. The first version of checkpointing only allows the application to log the sequence number of the last byte of data it has handled. The second version of checkpointing allows the application to log with a customized data structure, but the system call that will be used to synchronize the restarted application and the TCP layer are unimplemented.

Chapter 4

Evaluation

We test the system performance on a cluster of 266 MHz Pentium II PCs with 128MB of RAM connected by a Myrinet network with a LANai (version 3.0). Clients are connected to the servers by 100 Base-T Ethernet.

There are two main aspects to evaluate the performance of a replication system: the overhead caused by replication in during normal operation, and the gap between the primary failure and takeover. Three groups of experiments are conducted to measure the overhead of setting up the TCP connection, the overhead of communication and the time required between the failure detection and returning the reconstructed sockets to the application.

4.1. Connection Setup

In this experiment, a client sends 10 groups of 40 connection requests to the primary server. After each group of requests, both the client and the server close these connections, and the client sends another group of connection requests. For comparison, we conducted the same experiment to set up connection to a socket without the backup option. The average time on setting up a connection in each group is shown in Table 4-1.

When it listens on a socket, a server can set the value of the backlog, which determines how many connection requests may be put in the waiting list. The results show that when this value is big enough, the connection set-up time to a socket with the backup option is similar to that of a socket without this option. This benefits from the two-phase backup environment setup. As shown in Figure 4-1, when a connection request arrives, the system only sets the backup option to indicate the connection should be replicated, but does not communicate with the backup server. An acknowledgement is then arranged to be sent. Only when the "accept" system call comes from the application, will the system communicate with the backup server to set up the backup environment.

	backlog = 0		backlog = 50	
	normal	backup	normal	backup
1	62,990	1,413,841	376	422
2	397	1,424,753	374	440
3	415	1,424,754	387	454
4	397	1,424,759	400	470
5	412	1,424,733	412	478
6	428	1,424,737	443	448
7	453	1,424,758	466	466
8	476	1,424,758	486	482
9	508	1,424,758	455	509
10	472	1,424,752	441	535
Avg	440*	1,423,660	424	470

Table 4-1 Connection Establishing Time (in μs)

When the backlog is set to 0, only when the prior connection request has been processed will a new connection request be handled. Table 4-1 shows that the connection setup time to a backup socket increases dramatically in this situation. Comparatively, the connection setup time to a socket without the backup option is almost not influenced except for the first group.

* This value does not count the result of the first group.

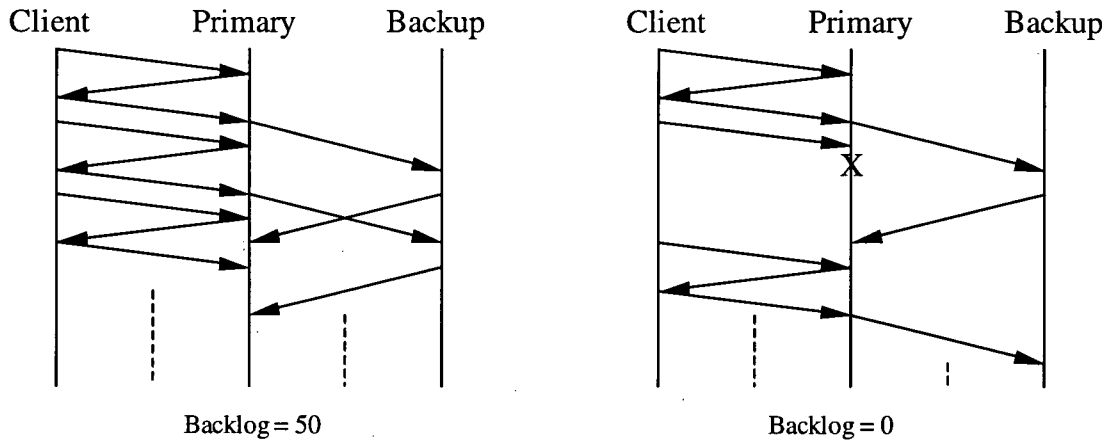


Figure 4-1 Connection Set up

To explain the surprisingly long time on connection establishment, we must go over the 3-way handshake of TCP connections. As introduced in Section 2.6.2, the server sends back an acknowledgement for a connection request with a SYN, and puts the newly created socket in an incomplete connection queue. When the client receives the acknowledgement, an acknowledgement for the SYN is sent back and the connection system call returns to the user. Upon receiving the acknowledgement, the server moves the created socket from the incomplete queue to the complete queue, and wakes up the process blocked by the accept system call, if applicable. By now the 3-way handshake is finished.

In our system, setting up the backup environment on the backup server introduces overhead to the last step of the 3-way handshake, as shown in Figure 4-1. Upon receiving the acknowledgement, the server moves the created socket to the complete queue, but waking up the user process is delayed until the backup environment has been set up on the backup server. When the backlog length is set to 0, if a new connection requests arrives during this period, it will be simply dropped because there is a pending socket in the complete queue [Stevens 94]. The client has to resend the connection request after

time out, and this time, the request is processed at once. This explains why the average connection establishment takes so long time in the backup system. Comparatively, in the normal system, in most cases when the second connection request arrives, the prior created socket has been removed from the complete queue and passed to the user. As a result, the connection request is processed immediately instead of being dropped. There is often some chance in the very beginning that a request is dropped because of the privious request has not been fully processed, so the average connection establishment time in the first group is much longer than those in the other groups.

4.2. Data Communication

Two experiments are conducted to test the overhead of data communication caused by replication. In the first experiment, the client application exchanges data package of 1000 bytes with the server. As in the Stop-and-Wait protocol, the client does not send the next packet until it receives the response to the privious packet. The results shown in Table 4-2 are the total time for exchanging 5000 packets. The time for exchanging data with a socket with the backup option is about 4.0% longer than the time for a socket without this option.

	1	2	3	4	5	Average
Normal	10,755,143	10,763,547	10,771,293	10,774,137	10,770,663	10,766,957
Backup	11,202,858	11,206,439	11,188,651	11,167,897	11,202,721	11,193,713

Table 4-2 Time of Communication (in μs)

In the second experiment, we measure on the overhead on different packet sizes. Similar to the first experiment, the client sends the next packet when it receives the response from the server for the previous packet. Different from the first one, the response is just an acknowledgement of 1 byte. Since big packets may be segmented by

the TCP layer, the server needs to collect all the data in a packet before it sends back the acknowledgement.

From Table 4-3 we can see that the overhead caused by replication is decreasing with the packet size growth. The reason is that the replication is packet based. The overhead on each packet is relatively constant, almost independent of the packet size. When a packet is too big to fit in a DTU of the lower layer, the TCP layer segments it to smaller packets. The DTU size of Ethernet is normally 1500 bytes, so the size of the data in a packet received by the receiver is normally 1460 bytes (1500 DTU – 20 TCP Header – 20 IP Header). This explains why the relative overhead is comparatively stable when the packet size is bigger than 1460.

Packet Size(B)	256	1,024	4,096	16,384
Normal(μs)	2,731,144	6,308,314	19,463,870	112,992,111
Backup(μs)	3,106,565	6,712,500	19,756,464	114,294,780
Overhead	13.7%	6.4%	1.5%	1.2%

Table 4-3 Overhead on Different Packet Sizes

We also test the throughput with and without replication by sending 5000 packets of 1 K bytes continuously without waiting for the response. There is no obvious difference between the two sets of results. The result shows that the replication is not the bottleneck of the communication, which can be done in parallel with the TCP communication.

A major reason that the system did not introduce high overhead to the communication is that we used the asynchronous communication mode of NetVM. In the asynchronous mode, the remote write simply puts the message in a local queue and returns. The message is sent out later. Since it is not blocked on replication, the system achieved good performance, but introduced the risk of inconsistency. If the synchronous mode is used, the overhead will be much higher.

4.3. Takeover

The takeover time depends on many factors, such as the total number of connections replicated, the number of sockets to be reconstructed, the size of the data to be copied to the sending and receiving queues, etc. We used a simplified program to test the approximate takeover time. The system takes several steps to take over:

1. Failure detection.
2. Searching for control blocks belonging to the failed primary server.
3. Copying data to the sending and receiving queues of the sockets to be reconstructed.
4. Using ARP message to add the IP address of the failed primary server as an alias.
5. Forking a new process.
6. Creating file descriptors associated to the new process for the reconstructed sockets and copying them to the user space.

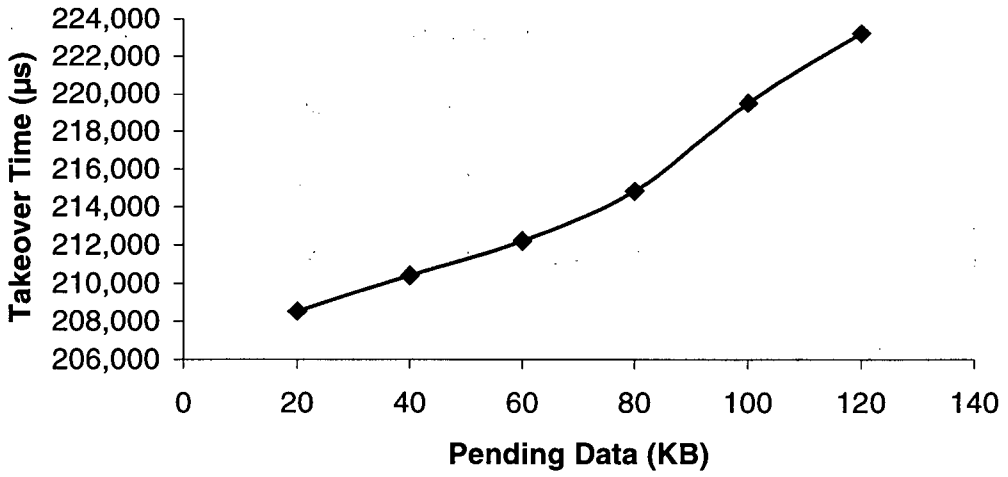


Figure 4-2 Takeover Time on Different Data Sizes

7. Returning to the user space.

After this, the daemon application executes the corresponding application, and the application recovers the service from the log.

The first step takes about 1 second for the system to detect the failure of a primary server. We measure the overall time from step 2 to step 6. In the simplified system, there is only one primary server, and there is only one connection from the client to the primary server to be replicated. It is shown in Figure 4-2 that the takeover time increases with the sizes of data to be copied, but not in a large scale.

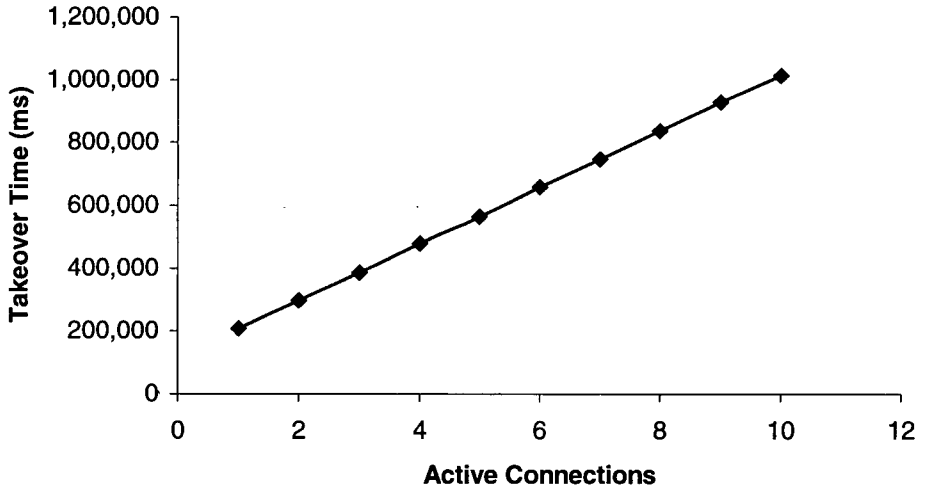


Figure 4-3 Takeover Time on Different Connection Numbers

Figure 4-3 shows the relationship between takeover time and the number of active connections. The results are sampled on a single client with different numbers of connections with the server. The backup server takes over when there are 10K bytes of pending data on each connection. The takeover time increases linearly with the number of connections.

Comparing Figure 4-2 and Figure 4-3 we can see that the influence of the number of active connections is much bigger than that of the pending data size. When there are many active connections, the course of takeover may take a rather long time, which in turn may cause the client to give up after trying to communicate with the server for several times.

4.4. Overall

The results from these experiments show that the system achieves our design goals with acceptable overhead. Although the connection establishing time is not so satisfying when the backlog is set to 0, there is no server application using such configuration in practice. Through the experiments we also found that when it sends out data asynchronously without waiting for a response from the server, the client may achieve approximately the same throughput as that in a non-replication environment. However, the scalability of the system is constrained by the takeover time. And as discussed in section 4.2, using synchronous communication will degrade the system performance. Checkpointing, which we did not use when we tested the system, will also introduce overhead.

Chapter 5

Conclusions and Future Work

5.1. Conclusions

The thesis has evaluated the idea of implementing replication in the TCP layer. By implementing replication in the TCP layer, the services provided by the primary server migrates to the backup server without breaking the TCP connections when a primary server fails.

In general, the system has met all the initial goals stated in Chapter 3. First, the system is transparent to clients, which will not notice the failure of the primary server and do not need to reestablish TCP connections. Second, the overhead caused by replication is relatively low and there is no big influence on the system performance. Third, the server application is only involved to checkpoint from time to time to use this service in normal operation. And last, the system provides an easy way to checkpoint and take over. Using this system, an application can easily achieve fault tolerance without managing replication itself. The system provides a general service for the applications with the requirement of fault tolerance.

Most existing fault tolerant systems are implemented in the application layer, so the failure of a server cannot be transparent to the clients. Every system must manage the

replication itself. Some other systems [AAPPS99] [APB96] [AYI97] [DCHKW97] [PABSDZN98] provide the mechanism of redirecting client requests to another server, but they are not designed for the problem that our system addresses, and the redirection in these system only happens in the connection establishment stage.

The success of our system shows that implementing replication in the TCP layer is a feasible way to achieve fault tolerance. By separating replication and checkpointing, we can use the system as a more general-purpose tool for other applications. Many techniques used in the system are proved to be useful and can be utilized in other situations as well. For example, the technique of migrating TCP connections can used in a content-based switch [AAPPS99] to reduce the overhead of changing TCP sequence numbers.

5.2. Future Work

This thesis presents the idea of implementing replication in the TCP layer. Although the results are promising, there is still room for future research.

First, the system is far from optimized. The overhead of setting up the backup environment can be reduced by using some more efficient communication method instead of UDP. In present implementation, the backup buffers are allocated from a submap inside the kernel, which has a limit of about 19 MB in our experimental environment*. This constrains the maximum number of sockets that can be replicated on the backup server. Moreover, because all the backup buffers are allocated inside the kernel, and the malloc() routine inside the kernel allocates unpaged physical memory [MBKQ96], the number of sockets that can be replicated is limited by the physical

* This information be obtained by using "vmstat –m".

memory available on the backup server. The system is just a prototype to test the idea of implementing replication in the TCP layer; many details required by a stable system, such as exception handling, are missing from our implementation.

Second, we only used some simple programs to test the system performance. More comprehensive applications need to be implemented to evaluate the system and the application interface.

The system is built on Myrinet, which is a Gigabit-per-second network widely used in research environments but not in the business world. More efforts are needed to port the system to commercial high-speed networks such as Gigabit Ethernet before it can be exploited in the real world.

Bibliography

[AAPPS99] G. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, D. Saha, "Design, Implementation and Performance of a Content-Based Switch".

[AB91] D. Agrawal and A.J. Bernstein. "A Nonblocking Quorum Consensus Protocol for Replicated Data". IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No. 2, p.p.171-179, April 1991.

[AM98] L. Alvisi and K. Marzullo, "Message Logging: Pessimistic, Optimistic, Causal, and Optimal". IEEE Transactions on Software Engineering, Vol. 24, No. 2, p.p. 149-159, February 1998.

[APB96] E. Anderson, D. Patterson and E. Brewer. "The Magicrouter, and Application of Fast Packet Interposing". OSDI, 1996.

[AYI97] D. Andresen, T. Yang and O.H. Ibarra. "Towards a Scalable Distributed WWW Server on Workstation Clusters". Journal of Parallel and Distributed Computing, 1997.

[Barlett87] J. Barlett, J. Gray, and B. Horst. "Fault Tolerance in Tandem Computer systems". The Evolution of Fault-Tolerant Computing, Vol.1, pp. 55-76, New York, 1987.

[BG83] K.P. Bernstein and N. Goodman. "The Failure and Recovery Problem for Replicated Databases". In Processing of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, p.p.114-122, August 1983.

[BFF95] T. Berners-Lee, R. Fielding, and H. Frystyk. "Hypertext Transfer Protocol – HTTP/1.0". IETF RFC 1945, October 1995.

[BJ87] K.P. Birman and T.A. Joseph, "Reliable Communication in the Presence of Failures". ACM Transactions on Computer Systems, Vol. 5, No. 1, pp.47-76, February 1987.

[BMST92] N. Budhiraja, K. Marzullo, F.B. Schneider and S. Toueg. "Primary-Backup Protocols: Lower Bounds and Optimal Implementations".

[CISCO] Cisco Systems Inc., "LocalDirector". http://www.cisco.com

[CM84] J. Chang, and N.F. Maxemchuk, "Reliable Broadcast Protocols". ACM Transactions on Computer Systems. Vol.2, No.3, pp.251-273, August 1984.

[DCHKW97] O.P. Damani, P.E. Chung, Y. Huang, C. Kintala and Y.M. Wang. "ONE-IP: Techniques for Hosting a Service on a Cluster of Machines". Computer Networks and ISDN Systems, 29:1019-1027, 1997.

[DG96] O.P. Damani and V.K. Garg, "How to Recover Efficiently and Asynchronously when Optimism Fails". In Proceedings of the 16th International Conference on Distributed Computing System, p.p. 108-115, 1996.

[DGP90] A.R. Downing, I.B. Greenberg, and J.M. Peha, "OSCAR: An Architecture for Weak-Consistency Replication". In Proceedings of IEEE PARBASE-90 International Conference on Databases, Parallel Architectures, and Their Applications, p.p. 350-358, 1990.

[Dimmer85] C.I. Dimmer, "The Tandem Non-Stop System".

[EZ92] E.N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output Commit". IEEE Transactions on Computers, Vol. 41, No. 5, p.p. 526-531, May 1992.

[FYT97] D. Funato, K. Yasuda, and Hi. Tokuda, "TCP-R: TCP Mobility Support for Continuous Operations". IEEE 97.

[Gifford79] D.K. Gifford, "Weighted Voting for Replicated Data". ACM SIGOPS, pp.150-162, December 1979.

[Herlihy87] M. Herlihy. "Concurrency versus Availability: Atomicity Mechanisms for Replicated Data". ACM Transactions on Computer Systems, Vol. 5, No. 3, p.p. 249-274, August 1987.

[IBM] IBM Corporation. "IBM Interactive Network Dispatcher". http://www.ics.raleigh.ibm.com/ics/isslearn.htm.

[JB86] T.A. Joseph and K.P. Birman, "Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems". ACM Transactions on Computer Systems, Vol. 4, No. 1, pp.54-70, February 1986.

[KG94] H. Kopetz and G. Grunsteidl, "TTP – A Protocol for Fault-Tolerant Real-time Systems". In IEEE Computer, Vol. 27, p.p. 14-23, January 1994.

[KR81] H.T. Kung and J.T. Robinson, "On optimistic Methods for Concurrency Control", ACM Transactions on Database Systems, Vol.6, pp.213-226, June 1981.

[LLG92] R. Ladin, B. Liskov and S. Ghemawat. "Providing High Availability Using Lazy Replication". ACM Transactions on Computer Systems. Vol.10, No.4, pp.360-391, November 1992.

[MRJ97] A. Mehra, J. Rexford, F. Jahanian, "Design and Evaluation of a Window-Consistent Replication Service". IEEE Transactions on Computers, Vol. 48, No. 9, Septermber 1997.

[OL88] B. Oki and B. Liskov. "Viewstamped Replication: A New Primary Copy Method to Support Highly Available Distributed Systems". 7th ACM Symposium on Principles of Distributed Computing, pp.8-17, August 1988. ACM SIGOPS-SIGACT.

[PABSDZN98] V.S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, E. Nahum. "Locality-Aware Request Distribution in Cluster-based Network Servers". ACM ASPLOS VIII, p.p. 205-216, October, 1998.

[PL91] C. Pu and A. Leff, "Replica Control in Distributed Systems: An Asynchronous Approach". In Proceedings of ACM AIGMOD, p.p. 377-386, May 1991.

[PR85] J. Postel and J. Reynolds, "File Transfer Protocol (FTP)". IETF RFC 959, October 1985.

[PST97] K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, and A.J. Demers, "Flexible Update Propagation for Weakly Consistent Replication". ACM SOSP-16, p.p. 288-301, October 1997.

[RAV98] S. Rao, L. Alvisi, and H.M. Vin, "The Cost of Recovery in Message Logging Protocols".

[RMDJ94] J. Rexford, A. Mehra, J. Dolter and F. Jahanian. "Window-Consistent Replication for Real-Time Applications". Proc. Workshop Real-Time Operating Systems and Software, pp.107-111, May 1994.

[Schneider83] F.B. Schneider, "Fail-Stop Processors", Digest of Papers from Spring CompCon '83 26th IEEE Computer Society International Conference, pp.66-70, March 1983.

[Schneider90] F.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.

[SL95] X. Song, J.W.S. Liu, "Maintaining Temporal Consistency: Pessimistic vs. Optimistic Concurrency Control", IEEE Transactions on Knowledge and Data Engineering, Vol. 7, No. 5, p.p. 786-795, October 1995.

[Stevens94] W.R. Stevens, "TCP/IP Illustrated, Volume1: The Protocols". Addison-Wesley, January 1994.

[Trian92] P. Triantafillou. "High Availability is not Enough", IEEE 92.

[TT95] P. Triantafillou and D.J. Talylor, "The Location-Based Paradigm for Replication: Achieving Efficiency and Availability in Distributed System", IEEE Transactions on Software Engineering, Vol.21, No.1, January 1995.

[WB84] G.T.J. Wuu and A.J. Bernstein. "Efficient Solutions to the Replicated Log and Dictionary Problems". In ACM Proc. Of the Third Annual Symposium on Principles of Distributed Computing". pp.233-242, August 1984.

[WS95] G.W. Wright and W.R. Stevens, "TCP/IP Illustrated, Volume 2: The Implementation". Addison-Wesley, January 1995.

[XSSRT96] M. Xiong, R. Sivasankaran, J. Stankovic, K. Ramamritham, and D. Towsley, "Scheduling Transaction with Temporal Constrains: Exploiting Data Semantics". In Proceedings IEEE Real-Time Systems Symposium, December 1996.