Transparent Process Replication and Failover in Linux

Noel Burton-Krahn

Apr 20, 2002

1. Introduction

Internet server applications must support many simultaneous client connections at all times. They need to be scalable to many users, available at any time, and each connection must complete reliably. These features are critical in the long term, but are usually only considered after initial development. Most server applications are developed with inexpensive components that do not support high availability or scalability. After initial development, they must be altered to deal with hardware faults and high connection loads.

Adding fault tolerance to an existing system can be difficult and expensive, and may not be possible for some kinds of server applications. Many Internet server applications are developed using freely available tools like Linux, Apache, PHP, MySQL, etc. None of these applications has built-in fault tolerance. There are technologies available to use redundant non-fault tolerant servers, but they have limitations. They will allow clients to reconnect to a new server if one fails, but connections and state at the failed sever will be lost. These solutions often rely on client connections being short and repeatable, like Web connections. But they are not suitable for a real-time teleconferencing or gaming application. They are also not suitable for databases, since redundant database servers must maintain a consistent state.

This project adds transparent fault tolerance to existing servers. Application state is duplicated on two independent boxes that run in parallel. If one fails the other can continue without interrupting client connections.

2. Definitions

Client, Server – a server is a collection of processes on a single machine that accept and process connections from clients. This project's goal is to allow backup servers to complete clients' requests without interruption even if a master server fails. This project focuses on servers running on Linux, which accept TCP connections from clients over the Internet.

Master, Backup – The master is the primary server responsible for handing a client connection. The backup is another server that can take over the client connection if the master fails.

Failover - The ability for a client connection to be relocated from master to backup without interruption or loss of information. Failover should be transparent to clients. The client's connection should not be broken or need to be manually restarted. The difficult part or transparent failover is transferring the state from the failed master to the backup

Scalability – The ability to increase the maximum number of client connections by load balancing among several servers.

High Availability – Keeping an application available despite server failure. Typically, a high availability algorithm forwards all requests to a preferred server until that server becomes unavailable at which point load balancing occurs to locate another server that is still operational.

Cluster - A group of servers which work together to share the load of many client requests. Clusters are most effective for load balancing. Failed servers may be removed from the cluster while new client requests go to functioning servers. However, all client connections to a failed server are broken when the server fails.

Application State – As the client and server communicate, the server changes state. The server may advance a file pointer, update files on disk, or change its internal memory state. Somehow the backup must preserve all these aspects of the application state to support transparent failover.

Connection State – A backup server cannot simply open a socket and resume a TCP connection to a failed server. The backup has to resume all aspects of a TCP stream, including sequence numbers (SEQ) acknowledgements (ACK), and timers for timeouts and retransmits. The operating system is responsible for maintaining TCP state, not the server itself. Tracking TCP state is hard for user applications because the TCP state is held in kernel memory away from application control.

Encryption State – client-server communication may be encrypted. Encryption state consists of the initial session key and the current stream cipher state which changes with every byte read, like a checksum or shift register. Encryption is designed to prevent a third party from intercepting or assuming one side of a conversation. A backup server must cooperate with a master server so the backup maintains the same encryption state.

3. Common Fault Tolerance

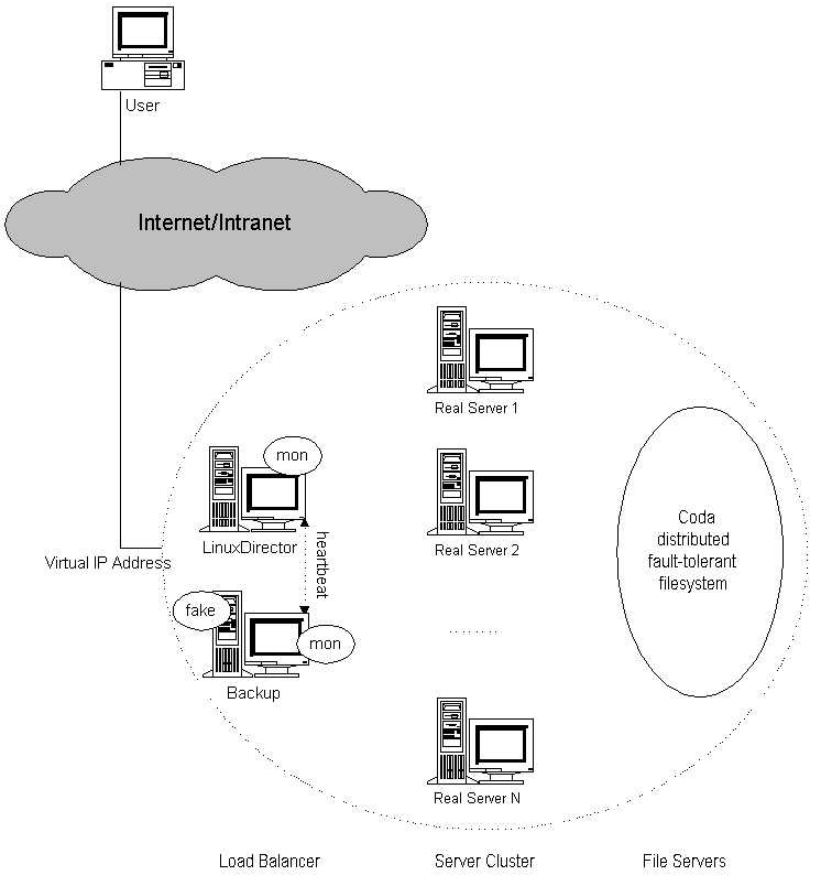
There are several strategies for fault tolerance in use today, but none of them adds transparent fault tolerance to existing applications. Systems that do add fault tolerance to existing systems always break client connections on failure, and may loose data.

3.1. Server Clusters

The most common strategy for fault tolerance and load balancing is to distribute connections over a pool of identical servers. This diagram is from the Linux Virtual Server (LVS[1]) project, and is a very common architecture:

---

[1] http://www.linuxvirtualserver.org/



High Availability of Linux Virtual Server

A pair of "Directors" accepts client connections and forward them on to "RealServers" The RealServers run identical software and share a common file system.

The Directors are redundant. If one fails, the other assumes its IP address to accept new connections. The Directors monitor the health of the RealServers and do not forward requests to unresponsive RealServers. Many commercial products provide the same functionality. f5Networks's BigIP[2] acts as a Director monitoring RealServer health and distributing requests. BigIP also works redundantly and preserves client connections and encryption state on failures. Cisco's LocalDirector[3] is a similar product.

---

[2] http://www.f5networks.com/

[3] http://www.cisco.com/warp/public/cc/pd/cxsr/400/index.shtml

RealServers in LVS use the Coda[4] file system. Coda allows redundant file servers uses robust caching. The result is good performance and the ability to survive server failures and client disconnections. Some RealServers in LVS may also function as Coda servers.

One common server missing from this diagram is a shared database. Web applications almost always have a database shared by web servers. Scripts on the web servers provide an HTML interface to the database, and usually store client session information there as well.

This diagram may be missing a shared database because databases are hard to replicate and are often a single point of failure in a web server application. Commercial redundant database solutions like Oracle[5] and Solid[6] are expensive. Many Web applications are made with freely available databases like MySQL[7] and PostgreSQL[8] that have excellent performance, but no built-in fault tolerance.

This server cluster architecture is ideal for web server applications. There are many simultaneous client connections, but they are each short and mostly read-only. The client's session (e.g a shopping cart) is the only information that changes frequently. None of the servers are expected to maintain state themselves. Client connections will be broken if a Director or RealServer fails. A broken connection is a minor inconvenience unless it's during a long download that must be repeated.

---

[4] http://www.coda.cs.cmu.edu/

[5] www.oracle.com

[6] http://www.solidtech.com/

[7] www.mysql.org

[8] www.postgresql.org

3.2. Periodic backup

This is a very common way of implementing backup for databases that do not support redundant servers. A snapshot of the database is taken every hour or so and saved. If the database crashes, a new one must be started with the backup. Starting a new database server and restoring its backup may take some time. Changes to the database since the last backup are lost. The database may have to be shut down during backup to prevent clients from updating the database while it is being dumped. The database may not support incremental backups, so the whole database may have to be copied. Connections to the database will be lost if it fails. Usually the client connects to a server that connects to a database. If the database fails, the server will fail and return an error to the client.

3.3. Application Fault Tolerance

It can be very difficult to maintain consistent application state between two servers, but some servers are built to do just that. Typically, applications that need fault tolerance the most are also the hardest to replicate. Strict application consistency can seriously degrade performance. Lazy replication improves performance but relaxes consistency guarantees. Database replication is still an active area of research ([Patiño00], [Wiesmann00]). A database may explicitly force all copies to be consistent before completing a transaction, or they may be lazy and defer consistency checks. Commercial databases like Oracle[9] and Solid[10] provide good application-level fault tolerance. Process checkpointing and hijacking [Skoglund00] is a technique of for preserving and replicating application state by serializing the entire state at some point and restoring it later. A process checkpoint consists of its data pages, executable path, and system descriptor like open files, file pointers, etc. Condor[11] ([Zandy99], [Litzkow97]) uses application checkpointing to move an application completely from one machine to

---

[9] www.oracle.com

[10] www.solidtech.com

[11] http://www.cs.wisc.edu/condor/

another. Like backing up data files, checkpointing is not suitable for real-time synchronization between two running processes.

3.4. Client Fault Tolerance

Some client-server protocols specify how a client should behave if a server is unavailable. The ubiquitous Simple Mail Transfer Protocol (SMTP[12]) is a good example. A mail transfer agent is required to search for a list of available mail servers and choose one at random, thus achieving load balancing. If a server is unavailable or fails, the client is required to follow a schedule of retries.

SMTP highlights how much easier it can be to implement simple load balancing and fault tolerance at the client side rather than at the server. It is not suitable for long two-way communications, but it is the standard for delivering mail on the Internet today.

3.5. Fault-Tolerant Programming Frameworks

Most off-the-shelf components for a server application are not fault tolerant, and adding fault tolerance later is not optimal. Another option is to write or rewrite a server application from scratch using a development framework that supports fault tolerance like J2EE and Enterprise Java Beans (EJB[13])

These frameworks work well for the domain they were designed for. EJB Is designed for writing Web-like applications where scripts provide an interface to a common database. EJB provides facilities for load balancing requests, maintaining client sessions, and fault tolerance.

Close examination of the EJB specification however reveals a limitation of EJB fault tolerance: transactions must be idempotent. If a transaction fails, the framework will

---

[12] http://www.faqs.org/rfcs/rfc2821.html

[13] http://java.sun.com/products/ejb/

automatically repeat it. Repeating a transaction must be equivalent to doing it exactly once. Fetching a record from a database is idempotent, but decrementing a balance it not. Applications that require non-idempotent operations have to implement their own fault tolerance.

3.6. Redundant Hardware

Fault tolerant hardware has good performance, but it has been traditionally very expensive. Lately though, hosts with all redundant hardware components have become available at competitive prices. The Stratus ftServer[14] is available for about US$5K. Of course, the price ceiling is unlimited.

These are excellent solutions for avoiding hardware failures. They are often expensive though. They also do not offer protection against catastrophic failures like fire.

4. Proposal

There is still no general way to provide transparent failover for off-the-shelf servers. This project's goal is to add transparent fault tolerance to existing servers without rewriting them. Any existing server should be able to fail over to a backup without breaking client connections. The master and backup server hardware should be totally isolated, except for a network connection. This system does not address load balancing. One machine provides a totally dedicated backup for another to achieve perfect transparency. This system has been implemented and does achieve those goals for a common type of server.

4.1. Scope

The perfect solution would provide completely transparent failover for any failure on any server. This project's goal is not so broad. Rather, it aims to provide transparent failover for some popular servers which do not support failover natively.

---

[14] http://www.stratus.com/products/nt/

4.1.1. Fault Coverage

This system is designed to protect against faults that cause a host to become unresponsive such as hardware failures, network failures, power failures, or natural disasters. It does not protect against software errors, or other errors where the system keeps functioning, but incorrectly.

The focus is on making a server highly available even though it runs on unreliable hardware. The hardware will eventually fail, but a backup will always be available to take over without interrupting current connections. Connections to a server should also be highly reliable by not breaking them if the server fails.

4.1.2. Replication and Determinism

This system replicates application state on a remote backup by running a remote copy of the server program and feeding the backup the same input as the master. This presumes that server programs are deterministic. The master and backup programs must maintain the same state if they are fed the same sequence of inputs from startup.

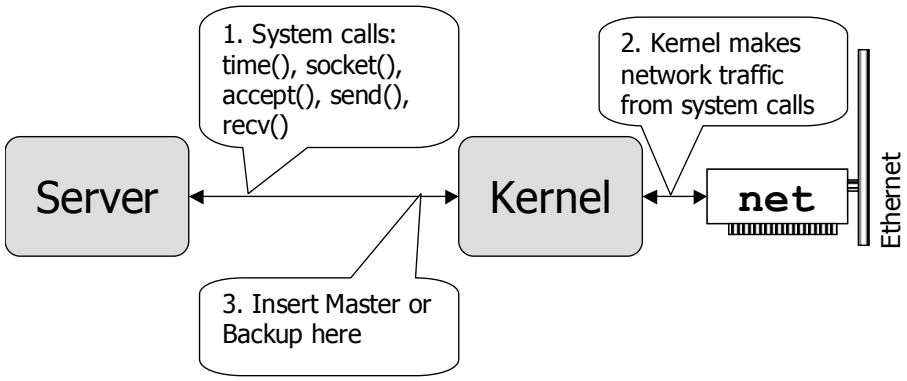
Programs should be deterministic by their nature. However, they will be run on different hardware with independent clocks and interrupts under operating system kernels, which are not synchronized. If a program depends on system state which cannot be synchronized, it is nondeterministic, and the master and backup states will diverge.

4.2. How it Works

The proposed system completely duplicates a master server on an independent backup machine. The backup server is fed the same input as the master and monitored to ensure that it would produce the same output. Both master and backup receive the same input from the client. The backup sends its output to the master. The master receives and verifies the backup's output and forwards it on to the client. While the backup produces

the same output the master would, it is indistinguishable to the client. That is, the backup will be able to replace the master at any time without the client's intervention or knowledge.

Transparency requires synchronizing both connection state and application state perfectly between master and backup. A network server is a collection of processes running on a single host. Server processes accept client connections, fork new processes, and may connect to other servers. Each process must be synchronized with its twin on the master machine.Each client connection must be verified to ensure the backup and master are in sync.



4.3. Synchronizing Application State

Synchronizing application state between two programs can be very difficult at the application level. Each application has its internal memory state, file pointers, etc. To synchronize state at the application level, an application has to record all changes to its state during a transaction, and verify that the backup performs the same state transitions. This project takes a different approach: if programs are deterministic then if they receive the same input, they will arrive at the same internal state.

Determinism is a requirement of this system. If a program starts in a known state and receives the same input, it must always produce the same output. Not all programs are deterministic, but we hope that our target servers are. If a master server is deterministic, then the backup will always produce the same output as the master as long as the backup receives the same input.

Programs by their mechanical nature should be deterministic. However, the types of programs we are interested in receive input from non-deterministic outside events:

- All programs run under multitasking operating systems which rely on hardware interrupts to schedule tasks. The order and duration each task gets the processor is not deterministic.
- The operating system may deliver asynchronous signals to a process at different points in execution. Two programs will not receive the same signal at the same stage of processing.
- Different scheduling and event handling can cause the operating system to process network traffic in different order. In particular:
    - New connections may be accepted in any order
    - Packets may be lost at one host but not on another
    - Outgoing packets will be assembled in different sized chunks due to buffering, timing, and retries.
- The clocks on two hosts can never be completely synchronized, and scheduling will never guarantee that two programs read the clock at the same moment.
- Operating systems supply arbitrary ids for system objects. For example, process IDs returned by fork(), wait(), and getpid(). The master and backup processes will have different process ids.
- Programs may access hardware-specific files such as
    - /dev/urandom the system hardware random device

    - /proc/*- a Linux filesystem which represents the kernel's view of processes by process ID.
- Some programs may depend on uninitialized memory for input (intentionally or not).

Note that many of these sources of nondeterminism come from the operating system itself through system calls like time(), fork(), getpids(), read(), etc. The proposed system reduces nondeterminism by carefully synchronizing network traffic and system calls.

The idea is to make sure the initial conditions are identical on the master and backup machines before starting the servers then verify that the servers proceed in sync. Providing a duplicate initial state is straightforward. All executables, configuration files, and data files must be the same on the master and backup systems. It suffices to copy files from master to backup before starting. As long as they are in sync, the master and backup will reproduce writes to local files, maintaining exact duplicates of data files. That means the backup will also provide a current backup of all data on disk without using a shared file server or another backup strategy.

It is easy to believe that simple programs are deterministic. However, commercial servers are not simple programs. A commercial server will accept new connections at any time, create subprocesses for tasks, use interprocess communication, and may use special local devices. For example, most encryption implementations on UNIX use the special /dev/urandom device to get a random seed. Every read from /dev/urandom yields different results, but synchronization requires that each server gets exactly the same results.

Here is an example simple server that just prints its process id and the current time. Even the simplest of servers is nondeterministic. Process ids like child_pid and getpid() are arbitrarily chosen by the kernel. The result of time() would always be unpredictable even on two machines with a synchronized clock due to scheduling randomness.

```
accept_sock = socket();
listen(accept_sock);
while(1) {
    // accept a new connection
    // and create a child to service it
    client_sock = accept(accept_sock);
    child_pid = fork();

    if( child_pid>0 ) {
        // the parent goes back to accepting
        close(client_sock);
        continue;
    }

    // the child does the request and exits
    print(client_sock, "pid=%u time=%lu\n",
```

All is not lost though. The answer is to intercept nondeterministic system calls at the master and return their results to the backup

4.4. Synchronizing System Calls

System calls must also be synchronized. Consider an encrypted connection. Encryption requires random session keys. The random keys can be taken from the system time, process id, or /dev/urandom, the Linux kernel random device. When the master and backup processes make these calls to their independent kernels, they will get different results, breaking synchronization. It is not enough for system calls to be close. The backup must get exactly the same results as the master. This is accomplished by intercepting system calls on the master and sending their results to the backup. We assume each process will execute a deterministic sequence of system calls. When the master makes a call, it sends the results to the backup. When the backup makes the same

call, it substitutes the master's results. If the backup looses its connection with the master, it assumes the master died and it becomes the master itself. If the backup starts producing different output, backup is assumed to have lost synchronization and it kills itself.

A system call is a function call that is processed ultimately by the OS kernel. On UNIX, all system calls are made available to C programs by libc.so, the shared system library. Libc preprocesses the arguments and then uses the architecture-dependent syscall() interface to trigger a context shift into the kernel which processes the request. System calls may be intercepted within the kernel, just before they get to the kernel, before they get to libc, or in the application space before libc. All these methods were evaluated for this project:

4.4.1. Kernel Level

It is possible to modify the kernel system call entry point. This was done for an earlier project by the author. MOSIX[15] [Barak99] also intercepts calls at the kernel level for process migration. The major drawback of this approach is hacking your kernel, and the lack of portability which that causes.

4.4.2. External Debugger

Debuggers are programs that attach to other programs so a developer can insert breakpoints and inspect memory while a program runs. GDB[16] is the most common UNIX debugger. Most modern operating systems including Linux and Windows provide an interface for writing a debugger. Under Linux, a debugger uses ptrace() to run a program until a system call. When the program calls the kernel, the kernel sends a signal to wake up the debugger. The debugger wakes up and uses read() and write() on /proc/$pid/mem to change memory in the client directly, then uses ptrace() to continue the target. Daniel and Choi [Daniel99] use this method.

---

[15] http://www.cnds.jhu.edu/mirrors/mosix/

[16] http://sources.redhat.com/gdb/

This approach has two drawbacks. First, the context switching between debugger and target eats a lot of time. Second, writing directly to process memory is very architecture dependent and non-portable.

4.4.3. Dynamic Code Patching

Yet another way of trapping system calls is to rewrite a process' core memory at execution time ([Tamches99], [Buck00], Detours[17]). There are two flavours.

The first method is the easiest. Every application has a table of function pointers that are resolved by the linker when the application is loaded. To trap a system call, just find that table and rewrite the entry.

One disadvantage with the function-call table approach is that it must be repeated for every dynamic library in a program's address space. The other approach is to write a jump instruction over the target function's body. Calls from anywhere in the process' memory space are guaranteed to be trapped.

The advantage of code patching is speed. There is no extra context switching because rewritten functions exist in the target application space. The disadvantage is the system dependence and lack of portability. Not to mention the risks of rewriting a program's memory space, which requires writing your own assembler/disassembler.

4.4.4. ld.so and LD_PRELOAD

Fortunately, there is an architecture-independent way of intercepting system calls inside a process' address space. The Linux kernel runs ld.so to load an executable and all its required dynamic libraries and resolve external function calls. Ld.so looks for libraries in

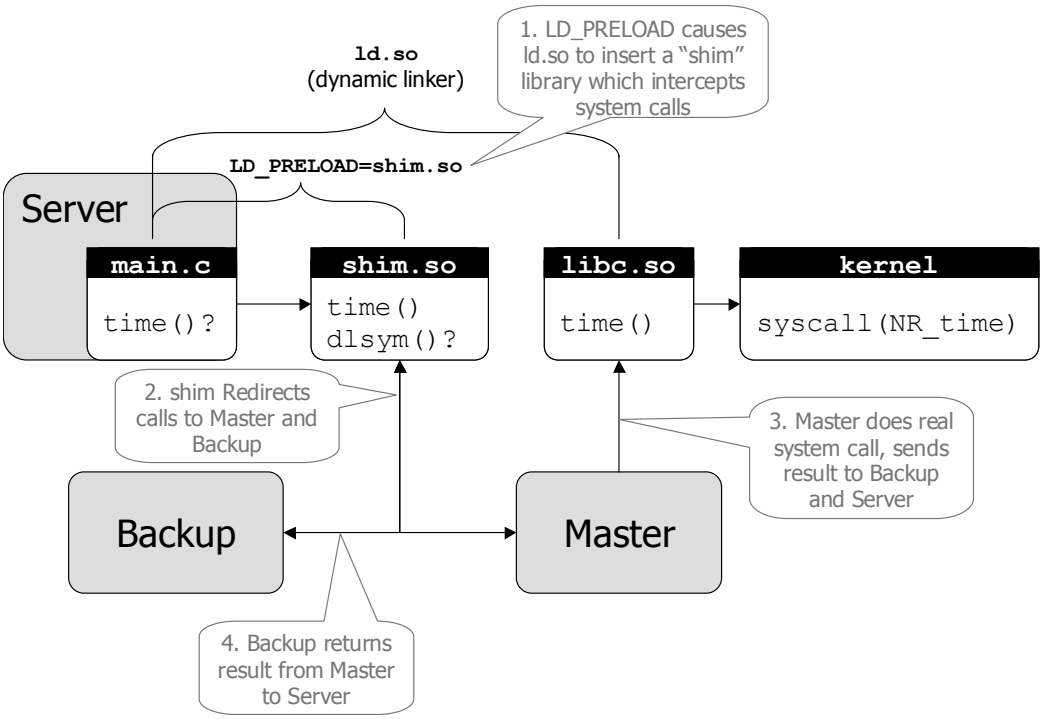
---

[17] http://research.microsoft.com/sn/detours/

a search path which includes the LD_LIBRARY_PATH environment variable. Ld.so also accepts the LD_PRELOAD environment variable which specifies a list of libraries to load and resolve symbols in preferred order.

Libraries loaded by LD_PRELOAD can redefine functions in other libraries like libc. For example, If an LD_PRELOAD library defines a function called time(), then that will override the time() defined in libc. Voila! A libc function can be overridden just by making a dynamic library which defines functions of the same name.

The redefined functions still have to call the real libc functions though. A call to dlsym(RTLD_NEXT, "time") will find the next definition of the time() function, usually in libc unless another LD_PRELOAD library has redefined it.



This approach works very well. The shim library intercepts system calls like time() , fork(), etc and passes the result from the master to the backup. In this way, the backup

gets exactly the same clock values (down to the nanosecond) as the master, and it sees the same process id space.There are ways to defeat this scheme:.

1. This only works with executables dynamically linked with libc.
2. Programs can still be non-deterministic. Their stacks and uninitialized memory will be different. Even though each program loads the same executables and libraries, they also load the shim library which behaves differently for master and backup. This can lead to a different stack and thus different uninitialized local variables.
3. Signals will also destroy determinism. Consider these programs. The program on the left is deterministic because it depends on time(). The one on the right uses an operating system signal which is non-deterministic. There is no way to control exactly when the alarm signal will be delivered to the process.

```
// run for 60 secs
i = 0;
t = time()
while(time()-t < 60) {
    i++;
}
// print i and exit
```

```
// alarm in 60 secs
int i;
void done() {
    // print i and exit
    printf("i=%d\n", i);
    exit(0);
}
```

4.4.5. Encrypted Connections

Any connection can be encrypted. Encryption is designed to prevent third parties from intercepting connections. The encryption key state starts randomly and changes with every byte, making it impossible to reconstruct. Still, a backup server must be able to continue with the same encryption state as its failed master.

The encryption state resides in application memory rather than in the operating system. So, if the application state is synchronized the encryption state should be too. SSL encryption starts with a server key and a random seed to generate a random session key. The random seed is usually read from the time, process id, and /dev/urandom. The shim

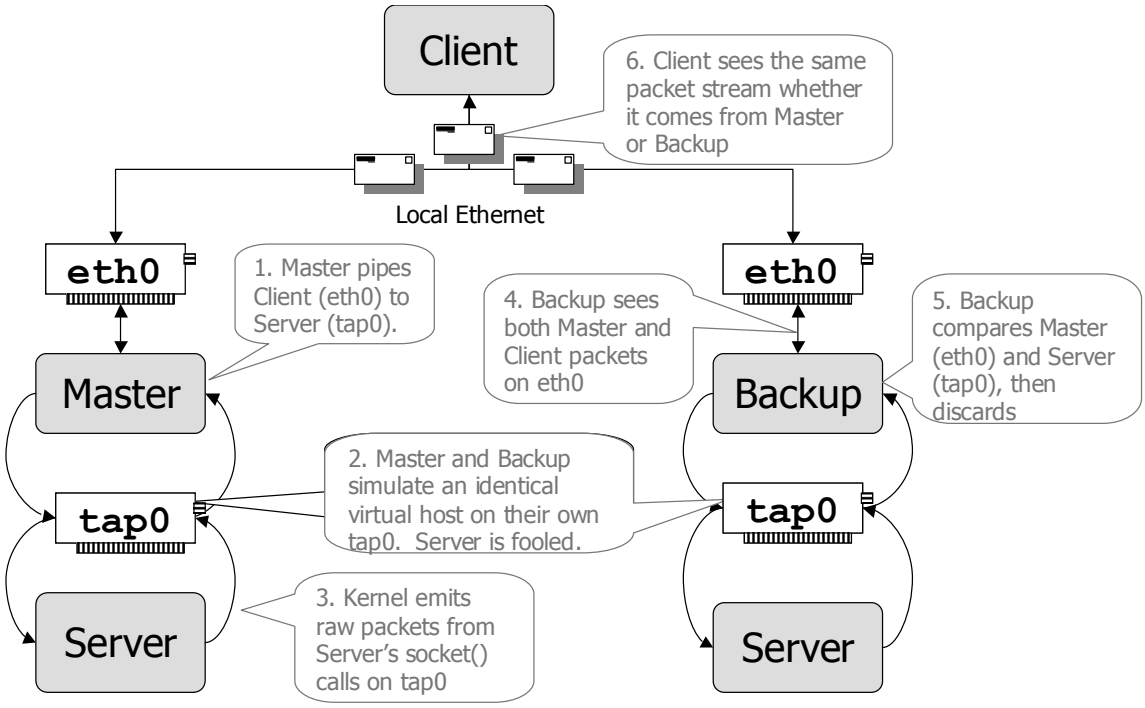
library synchronizes all these things. However, the encryption engine may also choose to use an uninitialized buffer for a random seed, and that cannot be guaranteed to be the same between master and server. Most UNIX encryption engines use the OpenSSL[18] library at their core, so a version of this could be tested and modified to use only synchronized sources of randomness such as synchronized getpid(), time() and reads from /dev/urandom.

4.5. Synchronizing Connection State

Synchronizing application state turned out to be fairly straightforward. Synchronizing connection state was much more complicated.

The idea is that both master and backup will share the same hardware and IP address and thus will both receive the same stream of packets from the client. The master will be responsible for sending packets back to the client. The backup stays in constant communication with the master and sends a checksum of the output it would produce. While the checksum agrees, the backup discards its output. If the checksum disagrees, the backup is assumed to have lost synchronization and it kills itself. . If the master fails to provide output, the backup takes over and sends its output to the client.

[18] www.openssl.org



A few tricks are required here:

1. The master and backup must present identical network interfaces -- that means the same IP and MAC addresses.
2. Server programs use the socket(), accept(), recv() and send() system calls. The kernel is responsible for generating network traffic. Somehow, the master and backup must trap and inspect all packets which come from the kernel from all server processes they control.
3. The master and backup may run at different rates, so they must do flow control to keep in sync

Items 1 and 2 are accomplished with Linux's ethertap device. That is a simulated network device that allows user-space programs to send and receive raw Ethernet packets. Writing and reading to /dev/tap0 will receive packets that would otherwise have been send out on the network. Solaris and BSD also support an ethertap device.

Master and backup sockets must be bound exclusively to an ethertap device so all their traffic is filtered by the master and backup monitors. This is accomplished by overriding the socket() system calls and using Linux's setsockopt(SO_BINDTODEVICE) system call. When each tap device is configured with the same IP and MAC addresses and all server sockets are bound to them, then all server traffic will have identical headers and be filter through the master or backup monitors.

The next step is to verify that each connection is identical. The master and backup track every client-server connection (there may be many several concurrent connections) and examining the packets that each side would emit. Initial testing confirmed that two independent servers with synchronized system calls would produce the same output. However, verifying that output by examining the network traffic proved to be tricky.

This project assumes all client-server communication is over TCP. TCP provides a reliable two-way stream of data between client and server. The TCP stack on each side (implemented in the kernel) must break the stream of data into packets, reassemble packets in order, and retransmit packets that get lost. Each TCP packet contains flags for initializing (SYN) and terminating (FIN) the connection, a sequence number (SEQ) for ordering bytes, and an acknowledgement (ACK) of the latest sequence number received. Here's a brief sketch of a TCP conversation:

Client	chooses initial sequence number Ic, sends SEQ=Ic, ACK=0, flags=SYN
Server	chooses initial sequence number Is, sends SEQ=Is, ACK=Ic+1, flags=SYN,AC
Client	sends data length=N1c, SEQ=Ic+1, ACK=Is+1, flags=ACK
Server	sends data length=N1s, SEQ=Is+1, ACK=Ic+1+N1c, flags=ACK
Client	sends data length=N1c, SEQ=Ic+1+N1c, ACK=Is+1+N1s, flags=ACK
…	
Server,	terminates by sending flags=FIN.
Client	

These are the crucial features of TCP for this project:

1. The initial sequence numbers (Ic and Is above) are randomly chosen by each kernel independently to avoid stale packets being mistakenly inserted into new connections.
2. The size of each packet (N1c, N1s) is arbitrary. The kernel uses timers and interrupts to buffer data written to a socket before assembling an TCP packet. It may delay transmission if there is not enough room on the other side (the TCP window) to receive them.
3. Sequence numbers are byte-oriented, not packet oriented.
4. Retransmitted packets may overlap. If a packet contains bytes 1-100, the next packet may contain 50-150.
5. Acknowledgement numbers do not have to match sequence numbers. Buffer restrictions may discard part of a packet.

The path to a correct solution had a few good mistakes, which I will not omit:

The first hurdle was random sequence numbers. The master and backup will both choose different sequence numbers for new TCP connections, and will not accept ACKs that do not match them. That means that all Client packets must be translated from the master's SEQ space to the backups, otherwise the backup would ignore the Client's ACK and the client would ignore the backup's SEQ. Luckily the difference between the master and backup SYN is a constant which can be observed by seeing the first packet from each side. So, every connection must be tracked individually and each packet fixed for the backup.

The next hurdle was different packet sizes. Buffering and timing caused the master and backup to produce packets of different sizes. The contents were the same, but the packet

sizes were different. So, each connection had to be tracked, the packet payload extracted (in order according to their SEQ) and the contents examined.

Flow control was the next problem. Often the backup would get out of step (different packet sizes) adjust its SEQ and get stuck waiting for an ACK which never arrived as the master and Client raced ahead. The answer was to make the master wait for the backup before sending its packets to the Client. The backup sends it traffic to the master. The master verifies the contents and sends as much as has been verified so far.

Finally one bug remained: the ordering of SYN packets. When there are multiple simultaneous connections, the master and backup may receive packets in different order or one side may lose a packet. That's not a problem for established connections; flow control takes care of it. However, the master and backup must accept new connections in the same order, otherwise the sockets returned by accept() will refer to different connections. That was easily fixed by making the backup delay accepting packets until the master did.

The final obstacle was to ensure that the backup could take over from a failed master. The backup and master ping each other regularly. If one fails, or a request times out, the other becomes the master.

5. The Solution

The final result of this project is the hotswap program. The user is responsible for making sure the initial conditions on the master and backup machines are identical: The user must configure the tap devices with the same addresses. Hotswap runs in master or backup mode. It opens a tap device, sets LD_PRELOAD to add the shim library to child processes and execs the server root process. The server is free to spawn as many children as it likes. Each child will get its own shim library via LD_PRELOAD and its own view of the world synchronized with its master process.

Here is a sample transcript:

1. The user starts the master and backup servers on two independent machines

User	synchronize file system with rsync
User	set duplicate IP and MAC addresses for tap devices on master and backup machines
User	run hotswap tap0 server arg0 arg1 … argn on master server
User	run hotswap tap0 –b tap0 <Master server IP> on backup server

2. Master and Backup each run their own copy of the server and synchronize system calls

Master	wait for connection from backup
Backup	connect to master
Master	send argv and envp to client
Backup, Master	set LD_PRELOAD=shim.so, exec(argv, envp)
Master Server	catch system call like time(). The shim sends the result to the backup
Backup Server	catch system call like time(). Wait for time() result from master and return that instead

3. Both machines accept connection from a client and verify output

Client	sends SYN to IP
Master	drop SYN on tap, send SYN address to backup
Backup	receive SYN, wait for master, drop SYN on tap.
Master Server	accept socket, fork() returns the new master pid to backup
Backup Server	accept socket, wait for master pid, then fork().
Master and Backup Servers	write() to socket
Master	read TCP packet from tap, wait for backup

Backup	read TCP packet from tap, send it to master
Master	compare TCP packet contents, send the smallest one
Client	Send ACK
Master and Backup	drop client packet on tap.

6. Testing

For testing, the backup and master both monitored a simple Web server written in Perl. Then 10 concurrent clients connected and downloaded several files in parallel. Both master and backup produced identical results, as expected, even under moderate load of many concurrent downloads. This test confirmed that

1. Synchronizing system calls and network traffic really does cause an independent program to produce identical results.
2. The system can handle many concurrent connections and moderate system load.
3. The System can work wit ha totally independent server. This one was written in Perl, so the shim was trapping system calls by the Perl interpreter.

7. Future Work

More testing needs to be done:
This should be tested with real production servers like Apache, MySQL and PostgreSQL, and servers written in Java.
This should be tested with OpenSSL

The backup should take over from the master on non-fatal errors such as reading from a failed disk drive

The backup and master are only connected by a TCP connection, which means they could be separated by a long distance to provide geographic failure protection. Maintaining a Client connection with a remote backup could be difficult. However, the master could forward Client packets to a remote backup. In that case, the backup would provide a continuously up-to-date backup in case the master is destroyed. The only question is how much overhead would that add.

8. Related Work

[Daniel99] presented a similar system which used ptrace() to attatch to redundant processes. That system used a modified NFS server to provide a common file system. The ESS1 system used a similar technique called supplication, see Bell System Tech Journal, 1964—1965.

[Aghdaie01] Presents a web server that preserves TCP connections and server state over failures. This is not a general solution for applications. It is a web server rewritten to support failover.

There is other work to maintain TCP connections. [Snoeren01] describes extensions to the TCP protocol that allow a client to redirect a TCP connection to a new server. This requires extensions to the TCP stack at both ends. The reliable sockets system [Zandy01] also masks TCP connection failures.

[Alvisi00] proposes a system that allows a server to keep its TCP connections open until it restarts after failure. If the application is written to exploit this feature and it can reconstruct its state after failure, it can avoid closing client connections.

9. Conclusion

This project demonstrates that it is possible to synchronize two server processes at the network and system call level. The two processes maintain identical memory states and produce the same network traffic. One can take over from the other transparently without breaking client TCP connections or loosing application state.

This system assumes processes are deterministic. It does work with independently developed servers, and it does synchronize system calls. Some programs may rely on input that cannot be synchronized between independent computers like asynchronous signals or kernel scheduling. This system needs further testing to determine the set of current commercial servers that it can support.

10.References

[Aghdaie01] Navid Aghdaie, Yuval Tamir. Client-Transparent Fault-Tolerant Web Service; 20th IEEE International Performance, Computing, and Communications Conference, pp 209-216, 2001, citeseer.nj.nec.com/aghdaie01clienttransparent.html

[Alvisi00] L. Alvisi, T.C. Bressoud, A. El-Khashab, K. Marzullo, D. Zagorodnov. Wrapping server-Side TCP to Mask Connection Failures. Technical Report, Department of Computer Sciences, The University of Texas at Austin, July 2000. http://citeseer.nj.nec.com/alvisi01wrapping.html

[Barak99] Barak A., La'adan O. and Shiloh A., Scalable Cluster Computing with MOSIX for LINUX , Proc. Linux Expo '99, pp. 95-100, Raleigh, N.C., May 1999, http://www.mosix.cs.huji.ac.il/ftps/mosix4linux.ps.gz

[Buck00] B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," Journal of Supercomputing Applications (to appear), 2000. 14 http://citeseer.nj.nec.com/buck00api.html

[Daniel99] Eric Daniel and Gwan S. Choi, TMR for Off-the-Shelf Unix Systems, The 29th International Symposium on Fault-Tolerant Computing
Madison, Wisconsin, USA, June 15-18, 1999 http://www.crhc.uiuc.edu/FTCS-29/pdfs/daniele2.pdf

[Litzkow97] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny, "Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System", University of Wisconsin-Madison Computer Sciences Technical Report #1346, April 1997. http://www.cs.wisc.edu/condor/doc/ckpt97.ps

[Patiño00] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, G. Alonso: Scalable Replication in Database Cluster In: 14th International Symposium on Distributed Computing (DISC), Toledo, Spain, October 2000. http://www.inf.ethz.ch/department/IS/iks/publications/pjka00.html

[Skoglund00] E. Skoglund, C. Ceelen, J. Liedtke, Transparent Orthogonal Checkpointing Through User-Level Pagers, In 9th International Workshop on Persistent Object Systems (POS9), Lillehammer, Norway, September 2000, http://www.l4ka.org/publications/files/l4-checkpointing.pdf

[Snoeren01] Alex C. Snoeren and David G. Andersen and Hari Balakrishnan, Fine-Grained Failover Using Connection Migration, Proc. of 3rd USENIX Symposium on Internet Technologies and Systems (USITS), 2001, citeseer.nj.nec.com/snoeren01finegrained.html

[Tamches99] Tamches, Ariel and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99), pp. 117-130. New Orleans, LA, February 1999. USENIX. http://citeseer.nj.nec.com/tamches01finegrained.html

[Wiesmann00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, G. Alonso: Understanding Replication in Databases and Distributed Systems. In: 20th International Conference on Distributed Computing Systems (ICDCS), Taipei, Taiwan, Republic of China, April 2000. http://www.inf.ethz.ch/department/IS/iks/publications/wpska00.html

[Zandy99] Victor C. Zandy, Barton P. Miller, and Miron Livny. Process Hijacking. The Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC'99), Redondo Beach, California, August 1999, pp. 177-184. http://www.cs.wisc.edu/paradyn/papers/#hijack

[Zandy01] V. Zandy and B.P. Miller, "Reliable Sockets", Computer Sciences Technical Report, University of Wisconsin, June 2001. ftp://grilled.cs.wisc.edu/technical_papers/rocks.pdf