Pp. 205-212 of the *Proceedings of LISA '02: Sixteenth Systems Administration Conference* (Berkeley, CA: USENIX Association, 2002).

HotSwap – Transparent Server Failover for Linux

Noel Burton-Krahn – HotSwap Network Solutions

ABSTRACT

HotSwap is a program that provides transparent failover for existing UNIX servers without modification or special hardware. HotSwap runs two instances of a server on independent machines in sync, so that if either machine fails, the other may assume control without breaking TCP connections or losing application state. Replication and failover is transparent to both clients and servers. Servers are not aware that a backup replica is maintaining state. Clients are unaware that a backup server has taken over from a failed master. This system is applicable to a wide variety of common servers including Java, Apache, and PostgreSQL and other servers that may have no other mechanisms for fault tolerance.

Introduction

Internet server applications must be scalable to many users, constantly available, and provide reliable service despite server failures, maintenance interruptions, and disasters. These features are critical in the long term, but are usually only considered after initial development. Most server applications today are developed with inexpensive components that do not support reliability, high availability or scalability, or any form of fault tolerance.

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 Apache, PHP, MySQL, etc. None of these applications has built-in fault tolerance. Current techniques for adding fault tolerance will allow clients to reconnect to a new server if one fails, but connections and state at the failed server will be lost. Fault-tolerant database and shared file servers are expensive. Implementing fault tolerance in a custom server is difficult.

HotSwap is a program that adds transparent fault tolerance to existing servers without modification. Application state is duplicated on two independent boxes that run in parallel. If one fails the other can continue without interrupting client connections. Replication is transparent to clients and servers, adding transparent failover to existing servers without modification.

There are several other techniques for fault tolerance [Coulouris01]. Each makes tradeoffs between degree of fault coverage, cost, and abilities. Scalability is the ability to increase performance by adding system components. Availability is the probability that a system is functioning properly at any moment in time. Reliability is the probability that a system will continue to function for a fixed period of time. Recoverability is the ability of a system to return to a functioning state without data loss after a failure. Total cost includes hardware, software, development, training, and maintenance. Disaster recovery is the ability to recover from a large-scale disaster like fire or earthquake that can damage a wide area. A single system image reduces maintenance costs when several components can be maintained as a single logical unit.

Different capabilities compete with each other. Adding system components to increase scalability and availability will increase cost and may decrease reliability. Reliability and recoverability require synchronized backups, which may decrease performance and scalability. Disaster recovery requires backups separated by long distances at reduced bandwidth, slowing system synchronization and performance.

Faults may come from software bugs, hardware failures, or disasters. Hardware failures should be masked by switching to a backup system. Disaster may strike a whole building or geographical area. Recovering data after a disaster is crucial, but restoring network connections may require new routing.

Different servers have different requirements for fault tolerance. Web servers use short transactions, which can usually be repeated if they fail. Databases and file servers that update client data must be careful to maintain consistency with their backups. Teleconferencing and gaming servers maintain real-time state in memory.

Let's quickly review current techniques for adding fault tolerance.[1]

Periodic Backup

Periodic backup is the easiest way to add disaster recovery. If a system crashes, a new one must be rebuilt from backup. Starting a new database server and restoring its backup may take some time. Changes to the system since the last backup are lost. Running applications may have to be shut down to prevent file

[1] The Aberdeen group has published an excellent comparison of current high-availability products for Linux at http://www.legato.com/resources/whitepapers/Aberdeen%20White%20Paper%20-%20Linux1.pdf.

updates during backup. Connections to the system will be lost if it fails, and will have to be restarted when the new system is restored.

There are backup schemes for whole processes, not just files. 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[2] [Zandy99], [Litzkow97] uses application checkpointing to move an application completely from one machine to another. Like backing up data files, checkpointing is not suitable for real-time synchronization between two running processes.

Server Clusters

A server cluster uses a front-end director to distribute client requests over several back-end servers. All back-end servers provide a consistent application interface. Consistency requires that all back-end servers use a shared database or file system. If a back-end server fails, the director will send new client requests to another server in the pool. The new server will retrieve the client's state from the shared database.

The Linux Virtual Server[3] project provides a director and a framework for back-end servers to use a shared Coda file system. F5Networks's BigIP[4] 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[5] is a similar product.

Directors are usually redundant. If one fails, the other assumes its IP address to accept new connections. Directors monitor the health of the back-end servers and remove failed servers from the pool. Directors may terminate client connections and attempt to repeat client requests if a back-end server fails. Directors do not address reliability or recoverability. That's up to the back-end servers and common database.

[2] http://www.cs.wisc.edu/condor/ .
[3] http://www.linuxvirtualserver.org/ .
[4] http://www.f5networks.com/ .
[5] http://www.cisco.com/warp/public/cc/pd/cxsr/400/index.shtml .

It is important to distinguish between availability and reliability in a server cluster. Clusters increase availability and scalability, but not reliability. If a server fails, another may be substituted for new connections, and thus availability is increased. However, once a client is connected to a back-end server, the connection's reliability is determined by the reliability of the chain of the director, back-end-server, and common database. The more components in the chain, the less reliable it is. Consider a server failure part way through a long file download. The replacement server will not know the state of the connection at the time of the failure, and the download must be restarted from the beginning.

Some directors like NetScaler[6] buffer the client transaction requests and will repeat them if the server fails. These are built to support short non-interactive transactions, like HTTP requests. Interactive or unbounded connections cannot be buffered and retried at the director level.

Server clusters are 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, and that is stored on a shared database. 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.

It is up to the back-end servers to synchronize shared state. Server machines may all connect to a central database or file server. If a middle server fails, all its state is lost and client connections are broken, but new client connections are directed to a new server. If a shared database fails, the whole cluster fails.

Fault Tolerant Applications

Some commercial database and file servers have built-in support for system synchronization and failover. Strict application consistency can seriously degrade performance. Lazy replication improves performance but relaxes consistency guarantees. Database

[6] http://www.netscaler.com/product/technology.html .

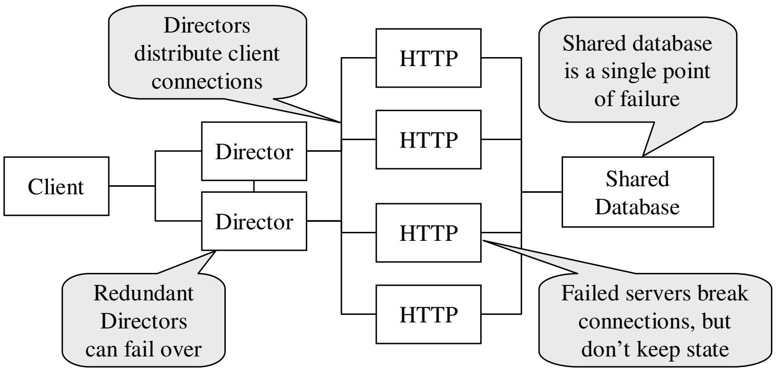

Figure 1: Schematic diagram of server cluster.

replication is still an active area of research [Patiño00, Wiesmann00]).

Commercial databases like Oracle[7] and Solid[8] provide good application-level fault tolerance. Fault tolerant file servers include NetApp Filer.[9]

Server clusters need a fault-tolerant shared database for reliability. However, most fault-tolerant applications are expensive. Rewriting an application to use a new database is not trivial. Converting an existing application usually requires considerable expense, training, and maintenance.

Fault-Tolerant Programming Frameworks

Developing a fault tolerant server is difficult. Another option is to write or rewrite a custom server from scratch using a development framework that supports fault tolerance like J2EE and Enterprise Java Beans (EJB[10])

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 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.

Fault-Tolerant 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[11] is quite reasonable.

Other systems use shared hardware like a shared SCSI disk to preserve state when a master fails. The shared disk itself may be fault-tolerant. However, the server's memory state and all client connections will be lost.

These are excellent solutions for avoiding hardware failures. They are often expensive though. Backup servers are connected by cables, so they are cannot be geographically separated. When disaster strikes the backup and master will both be vulnerable.

The different levels of RAID illustrate the tradeoffs between redundancy, performance, cost, and recovery. The simplest, RAID1, mirrors two disks. It is completely redundant, and does not increase performance. Higher RAID levels trade scalability for redundancy at increased cost.

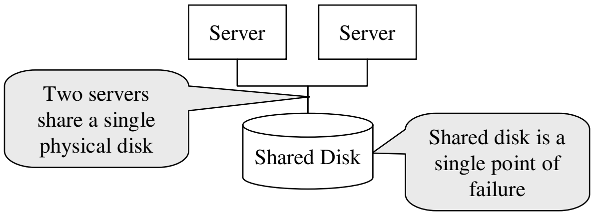

Figure 2: Shared hardware with a single point of failure.

TCP Connection Migration

All the servers mentioned so far break client connections on failure because they do not attempt to preserve TCP state. TCP connections are managed by the operating system itself. Even if an application synchronizes its internal state with a backup, the backup will not be able to reconstruct the sequence numbers, windows sizes, and timeouts of the master's TCP stack because that information is in the kernel, not the application.

There is recent research on moving TCP state to another machine. However, this research does not address how a client or server should transfer application state as well as TCP state. These systems provide facilities for moving TCP connections from a failed host to another. However, the server on the new host is responsible for ensuring that its state is consistent with the failed server. The server must be explicitly written to synchronize application state with a backup server.

[Snoeren01] describes extensions to the TCP protocol that allow a server or client to redirect a TCP connection to a new server without breaking the connection. This is handy for load balancing and avoiding failed servers. However, The TCP stack at both ends must be written to recognize these new TCP packet options. Replacement servers must implement their own synchronization for application state.

The reliable sockets system, Rocks [Zandy01], can preserve and reconnect TCP connections after link failures, address changes, and extended disconnection. Rocks does work without recompiling programs. However, servers must be rewritten to synchronize state. Rocks provides an API for managing and detecting resumed TCP connections.

Some redundant server cluster directors can synchronize both TCP and SSL state with their backups. If one director fails, the backup can continue the TCP connections. This only applies to the directors, not the back-end servers. A back-end server's TCP state is lost on failure.

[Alvisi00] proposes a system that allows a server to keep its TCP connections open until it restarts after

[7] http://www.oracle.com .
[8] http://www.solidtech.com .
[9] http://www.netapp.com/ .
[10] http://java.sun.com/products/ejb/ .
[11] http://www.stratus.com/products/nt/ .

failure. If the application is written to exploit this feature and it can reconstruct its state after failure, it can avoid closing client connections. However, the server must still be able to reconstruct its state after a crash.

[Aghdaie01] Presents a web server that preserves TCP connections and server state over failures. This is an example of a web server specially written to transfer both TCP and application state to a backup. This is not a general solution for all servers.

[Daniel99] presented a system similar to HotSwap. It used ptrace() to catch, redirect, and synchronize system calls between servers. However, that system used a central modified NFS server to provide a common file system. HotSwap does not use a shared file system; all servers are totally independent.

HotSwap

HotSwap maximizes availability and reliability by providing a hot backup server that maintains a complete independent copy of a master server's state. The backup is complete and dynamic, so it can take over all client connections when a server fails without interruption. It minimizes cost by adding this ability to almost any server without modification or special hardware. Both master and backup appear as a single computer on the network, and thus HotSwap provides a single system image. The tradeoff is a small amount of overhead to keep the master and backup servers in sync. HotSwap does not address scalability; backup servers do not share the load.

How it Works

HotSwap starts two identical instances of the same set of programs on two independent machines, a master and a backup. The programs are started from the same initial state, with duplicate file systems. As they run, HotSwap ensures that both copies are synchronized.

Synchronization means that both the master and backup programs see exactly the same input and produce exactly the same output. When a client connects, both servers receive the new connection. When a client sends data, both servers receive it. When the master server makes a system call, like requesting the current time, HotSwap ensures the backup gets the same value. In this way, both servers will go through the same sequence of state transitions and produce the same output. The master sends its output to the client. The backup verifies it would produce the same output as the master, and then discards its output.

If the master fails to produce output, or if it detects an internal error, the backup takes over. The backup can take over immediately since it is already in the same state as the master was before the master failed. The backup simply stops discarding its output. This is how HotSwap achieves transparent failover: the backup produces exactly the same output as the master would at any moment but discards its output until the master fails.

Both servers must start in the same initial state. To start the system, the two HotSwap processes synchronize their file systems then execute their server programs. After a failure, a new backup server can synchronize files without interrupting the surviving master. The operator can later choose when to restart the master and new backup to achieve full fault tolerance again.

Details of Synchronizing State

Synchronizing server state is critical. HotSwap requires that a server will produce the same output if it receives the same input from a particular set of sources. HotSwap synchronizes system calls like time(), getpid(), socket(), recv(), send(), etc. These are all the inputs used by our pilot servers, and they are enough to ensure that the servers we have tested are

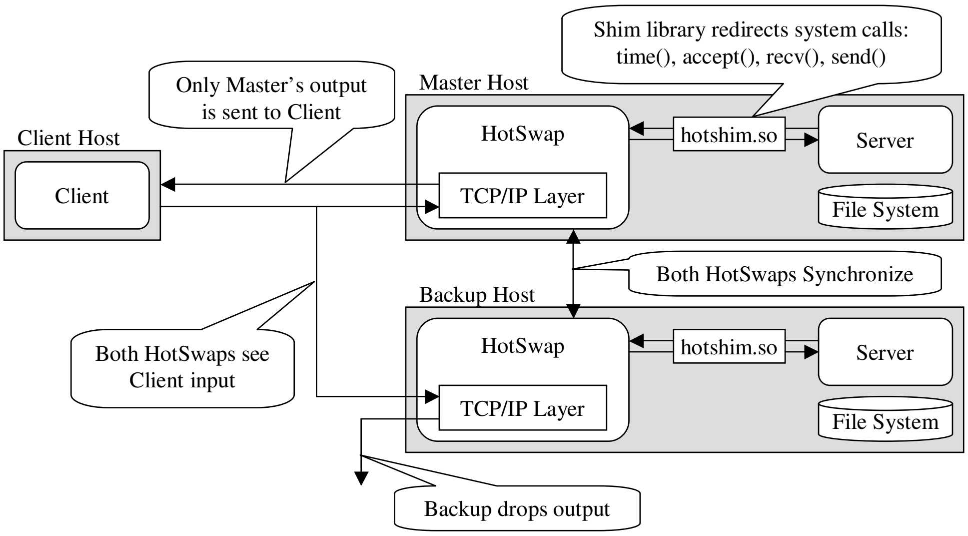

Figure 3: How HotSwap works.

synchronized. Some servers may use other sources of input like inode numbers or direct access to hardware that HotSwap cannot catch or synchronize. HotSwap may not be able to synchronize these servers, but it is likely that the servers themselves could be modified slightly to work with HotSwap to synchronize.

HotSwap synchronizes system calls by relinking programs before they run. In UNIX and Windows, applications are dynamically linked against libraries that provide system calls. HotSwap inserts a shim library that redefines system calls to transfer control to the running HotSwap program. HotSwap also has to synchronize network traffic at the TCP/IP level. Clients and servers usually communicate over TCP, a network protocol that breaks a stream of data into packets that are reassembled in sequence and acknowledged. The operating system (OS) is responsible for managing TCP connections. When a program calls send(), the OS adds that to the outgoing TCP buffer, sends a TCP packet and changes the TCP connection state. A TCP connection has many state variables, including packet sequence numbers, timeouts, buffered packets, and acknowledgements. HotSwap must synchronize these state variables, so it uses its own TCP/IP network stack.

HotSwap constantly intercepts and synchronizes a subset of system calls, just enough to ensure synchronization between processes. It also runs in a file-system-only mode where is just synchronizes changes to the local file system. This allows a backup to maintain an active backup of the master's file system without incurring the extra overhead of full process synchronization. This mode is used for disaster with a geographically remote backup.

Limitations

HotSwap relies on each server receiving input only from the set of system calls that HotSwap monitors and synchronizes. HotSwap synchronizes all system calls to sockets, time, file stats, process ids, semaphores, and the /dev/random device. HotSwap cannot synchronize state if a server receives input from hardware devices or from the timing of asynchronous signals.

The master and backup systems must start at the same time with identical file systems to ensure they receive the same input from local files. HotSwap runs chroot()'ed in a server directory to minimize the amount of files required to synchronize. The chroot() has the extra advantage of improving security by limiting the server's access to the file system.

After a server fails, the survivor will continue running by itself as a solo master. A new backup system will continue to synchronize files with the running master, but will not synchronize applications until the running master restarts. If the running master fails, the backup will restart with a synchronized file system, but existing connections and transactions will be lost.

Results

HotSwap has been tested with Perl, Java, Python, Apache, OpenSSL, OpenSSH, and PostgreSQL under Linux.

The first test was replicating a simple web server written in Perl to serve video files. The master was disconnected in the middle of a video, and the client continued displaying the video from the backup without

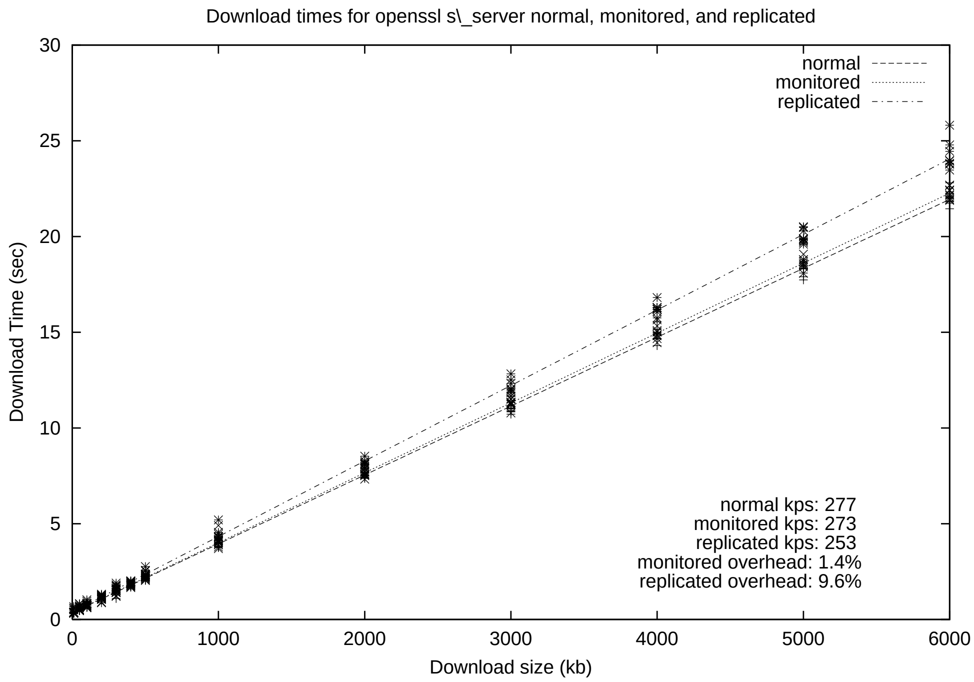

Figure 4: Download performance.

interruption. The same test was then performed using Java, Python, and Apache with good results.

The OpenSSL s_server program was tested to evaluate an encrypted web server. Encrypted connections are usually difficult to replicate, since each side must maintain identical encryption state, and that changes with every byte processed on an encrypted stream. However, as long as the master and backup use the same seed values to generate their initial session keys, they should maintain the same state. Unfortunately, our initial test with OpenSSL failed! Close examination of the OpenSSL source revealed that OpenSSL used an uninitialized buffer plus bytes from /dev/urandom for a seed. This highlights a limitation of HotSwap; processes must use only input from synchronized system calls. Fortunately, we easily patched the OpenSSL server to initialize its buffer and use only /dev/urandom, and the test succeeded. We tested s_server to measure the overhead for just intercepting and monitoring system calls and full replication. We measured the time required to download various sizes of files to see how the overall bandwidth of the server was impacted by replication, using commodity hardware.

The results show that intercepting and monitoring system calls only introduces a 1.4% overhead, and full replication to another box reduced bandwidth by only 9.6%.

The OpenSSH tests demonstrated that HotSwap really does provides a single system image where master and backup appear as one computer. We used ssh to log in and edit files and scp to upload files. All these actions were replicated on both the master and backup simultaneously and transparently.

Replicating PostgreSQL demonstrated that HotSwap can immediately add transparent failover to a database without modifying the database itself.

Conclusion

HotSwap has unique properties. It adds transparent failure and a single system image to servers without any shared components. Backup servers maintain identical file-system and internal memory states with the master. Client connections are never lost or broken on server failure. Servers do not have to be modified, with few exceptions. The price of fully transparent replication is a small amount of overhead.

Availability

HotSwap will be available in server kits that include a complete tested server and the minimal root file system required to support them. We expect HotSwap server kits to be available for download from http://www.hotswap.net in October, 2002.

About The Author

Noel Burton-Krahn received his M.Sc. in Comp. Sci. from the University of Victoria in 2002, and his B.Sc. in 1994. He spent the last six years designing and implementing Internet server applications, with emphasis on network protocols, security, and process migration. He has recently founded HotSwap Network Solutions, where he may be reached at noel@hotswap.net .

References

[Aghdaie01] Aghdaie, Navid and Yuval Tamir, "Client-Transparent Fault-Tolerant Web Service," *20th IEEE International Performance, Computing, and Communications Conference*, pp. 209-216, http://citeseer.nj.nec.com/aghdaie01 clienttransparent.html, 2001.

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

[Coulouris01] Coulouris, George, Jean Dollimore and Tim Kindberg; *Distributed Systems: Concepts and Design, Third Ed.,* Addison-Wesley, ISBN 0201-619-180, http://www.cdk3.net/ .

[Daniel99] Daniel, Eric 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, http://www.crhc.uiuc.edu/FTCS-29/pdfs/daniele2.pdf, 1999.

[Litzkow97] Litzkow, Michael, 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, http://www.cs.wisc.edu/condor/doc/ckpt97.ps, April 1997.

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

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

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

[Wiesmann00] Wiesmann, M., F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding Replication in Databases and Distributed Systems," *20th International Conference on Distributed Computing*

Systems (ICDCS), Taipei, Taiwan, Republic of China, http://www.inf.ethz.ch/department/IS/iks/publications/wpska00.html, April 2000.

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

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