IPSJ SIG Technical Report

Vol.2009-ARC-183 No.21
Vol.2009-OS-111 No.21
2009/4/23

TCPmig: Migration of processes with TCP sockets in Single IP Address Cluster

Balazs Gerofi ,[†1] Hajime Fujita [†1]
and Yutaka Ishikawa [†1]

Single IP Address cluster offers a transparent view of a cluster of machines as if they were a single computer on the network. In such an environment, process migration can play a significant role in order to provide services seamlessly and to increase sustainability. We present TCPmig, a mechanism for migrating processes with TCP connections. Preliminary test results show that the transition is smooth even under high bandwidth network traffic, furthermore, the problem of preventing incoming packet loss during the migration is also addressed. TCPmig is implemented entirely in a kernel module for Linux 2.6, without any modifications to existing kernel code.

1. Introduction

As the Internet is becoming more and more the common scene of our everyday life, online service providers encounter the need of deploying infrastructures that not only ensure high availability and reliability but at the same time scale well with the increasing number of clients. Moreover, there is an emerging urge towards building and operating computing sites with sustainability in mind. Most importantly, reducing energy consumption as much as possible[1)].

A cluster of inexpensive commodity computers connected with high-speed interconnects can achieve comparable performance to special-purpose supercomputers while it offers much better cost efficiency. Thus, deploying cluster-based server systems is becoming common-place. This sort of systems are potentially easy to build and also to extend. However, efficiently addressing scalability, reliability, fault tolerance and sustainability is challenging from the software development point of view.

†1 The University of Tokyo

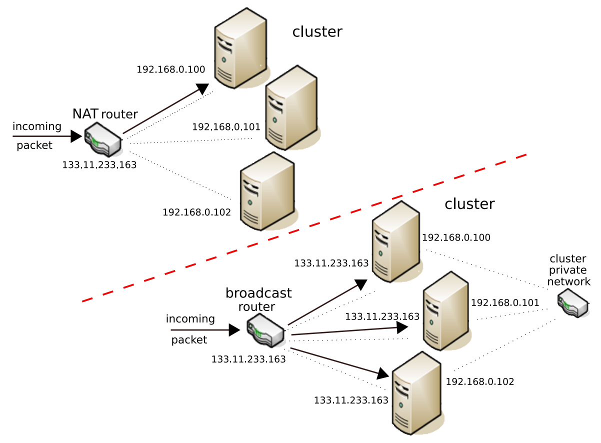

Fig. 1 Comparison between NAT and broadcast based Single IP Address Clusters

Single System Image (SSI) systems intend to address these problems on the operating system level and therefore letting application developers truly concentrate on business logic instead of dealing with system programming issues[2)].

Process migration is a mechanism which decouples an application from the physical machine that is executing it and allows the process to continue running on a separate computer, ideally, without any involved parties noticing the transition. Process migration can be used for load balancing to address scalability, it can increase fault tolerance and therefore makes the system more reliable and it also allows deallocation of computers which decreases the overall power consumption. Migrating applications that maintain TCP connections with their clients can cause difficulties due to the strong integration of a connection with its IP endpoints. Single IP Address Clusters help to overcome this problem by offering a transparent view of a whole cluster as if it was a single computer on the network. This makes it possible to migrate processes inside the cluster even with TCP connections, theoretically without the peer noticing it.

Providing a single IP address can be realized in several ways. The most common case is a cluster of machines with different local IP addresses and a router in front of them, which in turn translates the IP addresses appropriately. Linux

Virtual Server[3)4)], TCP Router[5)], and SAPS[6)] are based on this idea. In this case however, each time a connection is moved the router has to be updated which is extra administration, and besides, it can lead to incoming packet loss[7)].

Contrary, in a broadcast based cluster each node is equipped with a public and a local interface. The same IP address is assigned to public interfaces and the local ones are used for in-cluster communication while the router simply broadcasts each incoming packet to the whole cluster. Windows NLB[8)], ONE-IP[9)], Clone Cluster[10)], and Hive Server[11)] are examples of broadcast based Single IP Address clusters. **Fig. 1** depicts the difference between the two approaches. The broadcast property of this model makes it possible to migrate a connection without any extra effort on the router.

We have developed TCPmig, a mechanism for migrating processes with TCP connections. TCPmig moves the entire TCP state machine including buffer queues and preserves seamless data transfer by adjusting timestamps on the destination node appropriately. Experimental results show that TCPmig provides a smooth transition even under high bandwidth network traffic condition. Unlike other existing solutions, TCPmig addresses the problem of preventing incoming packet loss by exploiting the broadcast property of the network configuration.

The rest of the paper is organized as follows, Section 2 briefly introduces the Berkeley Checkpoint-Restart Library (BLCR)[12)] which we have modified in order to support migration of processes with TCP connections. Section 3 explains the Linux network stack focusing on structures which are relevant for the migration while Section 4 details the actual migration mechanism. Performance evaluation is given in Section 5 and related work is discussed in Section 6. Finally, Section 7 concludes the paper.

2. Berkeley Checkpoint-Restart library

The Berkeley Checkpoint-Restart library (BLCR)[12)] is an open source system-level checkpointer designed with High Performance Computing (HPC) applications in mind. In order to make an application checkpointable there is no need to modify its source code. Basic support for BLCR can be enabled by executing the application via a special tool provided by the BLCR package, linking the application statically with the BLCR checkpoint library or forcing the BLCR dynamic library to be loaded during the application's startup. Furthermore, BLCR offers a convenient property on the way how it interacts with its context file. All process resources are written sequentially (i.e. no file repositioning is issued) which ensures that even a socket, fifo or a pipe can be used for this purpose.

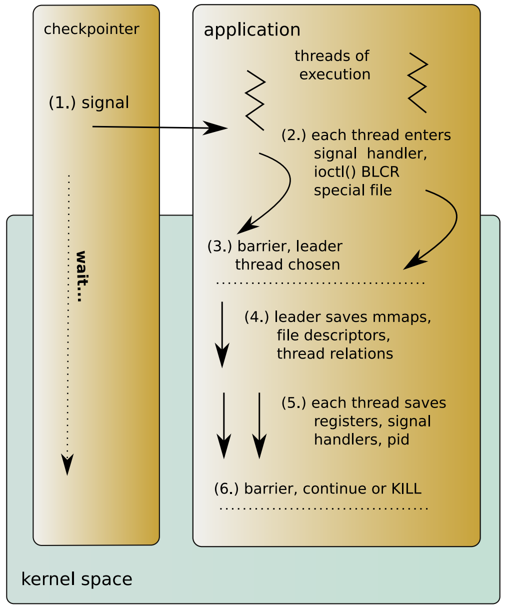
checkpointer
application
threads of execution
(1.) signal
(2.) each thread enters signal handler, ioctl() BLCR special file
wait...
(3.) barrier, leader thread chosen
(4.) leader saves mmaps, file descriptors, thread relations
(5.) each thread saves registers, signal handlers, pid
(6.) barrier, continue or KILL
kernel space

Fig. 2 BLCR checkpoint mechanism

2.1 Checkpointing

The BLCR checkpoint library installs a dedicated signal handler in order to make an application checkpointable. **Fig. 2** demonstrates the main execution steps during the checkpoint procedure, which are the following:

(1) The target process is notified via a signal that checkpointing is requested
(2) Each thread of the process executes the BLCR signal-handler, which issues an ioctl() call on the libraries character special file in order to enter kernel-space.
(3) The threads are synchronized and one is chosen as leader

(4) Leader dumps thread relations, memory mappings and file descriptors
(5) Each thread writes its registers, signal handlers and it's pid
(6) All threads are synchronized again and the program either continues running or gets killed according to the options specified

There are several restrictions on the checkpointable open files however. For example, sockets are entirely not supported and regular files are assumed to be available under the exact same path and are reopened during restart.

2.2 Restarting

Restarting an application is basically the mirror procedure of checkpointing. The restart utility opens the context file (which can be practically establishing a connection via a socket) and forks a new process. **Fig. 3** illustrates the main execution steps of the new process, which are the following:

(1) An ioctl() call is issued on the BLCR character special file in order to enter kernel-space
(2) The process clones itself as many times as many threads the checkpointed application contained.
(3) One thread is chosen as leader
(4) Leader restores process-wide resources, open files, memory maps and thread relations
(5) Each thread restores its pid, signal handlers and registers
(6) All threads return to user-space where they finish up the BLCR signal handler and eventually resume their regular execution

3. Linux network stack

In order to describe TCP migration in details it is necessary to provide an overview of the Linux socket infrastructure. The main kernel structures and the relations among them are introduced first. **Fig. 4** gives an overview of the most important structures with respect to migration highlighting some of their most relevant fields.

3.1 Linux TCP socket infrastructure

There are several data structures used in the kernel for representing and maintaining TCP connections. Each open file of a process is referred as a *file* struct from the process' file descriptor table. The directory cache entry field of a *file*

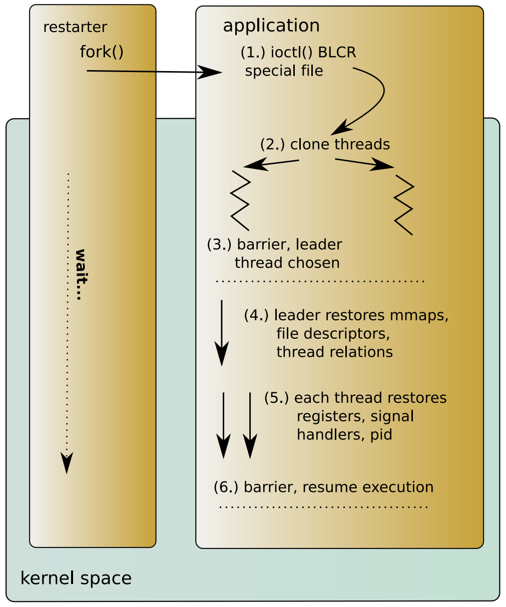
restarter
fork()
wait...
application
(1.) ioctl() BLCR special file
(2.) clone threads
(3.) barrier, leader thread chosen
(4.) leader restores mmaps, file descriptors, thread relations
(5.) each thread restores registers, signal handlers, pid
(6.) barrier, resume execution
kernel space

Fig. 3 BLCR restart mechanism

struct associates the file with the underlying *inode* structure.

In case the file is a socket the main *socket* structure is accessible through the *inode*. The *socket* struct represents a general BSD style socket, holds high level state information, function pointers to protocol specific methods and a reference to the *sock* structure, which is the actual network layer representation of the connection.

A rather central notion of the Linux network stack is the *sk_buff* socket buffer structure. Socket buffers are used for representing both incoming and outgoing packets on the network. The *sock* structure maintains buffer queues (write, receive and backlog) which are linked lists of socket buffers. It also holds timestamp values of last received and sent packets which are expressed in terms of kernel jiffies.

In an object-oriented fashion the connection representation is further spe-

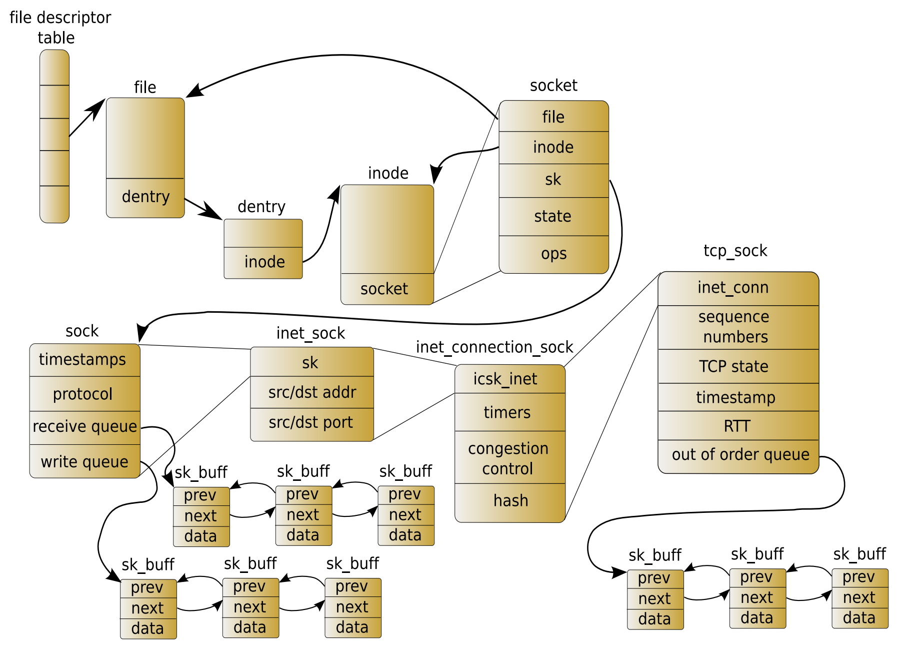

Fig. 4 Relevant structures for migration in the Linux TCP architecture

cialized according to the actual protocol used. Internet connections are represented in an *inet_sock* structure, which contains both local and remote IP addresses and also port numbers. Connected sockets are maintained through the *inet_connection_sock* structure which holds different timers, congestion control parameters and the hash bucket for the kernel hash table which is used for determining which socket is responsible for an incoming packet.

Finally, the *tcp_sock* structure keeps track of the TCP state machine. It stores sequence numbers, TCP state, fields for RTT measurement, it controls slow start mechanism and so on.

4. TCP migration

Similarly to process migration, TCP migration has also two phases, checkpointing and restarting a connection. Checkpointing a TCP connection involves copying necessary state information and dumping not yet processed packets from the relevant buffer queues.

The Linux kernel maintains several socket buffer queues for different purposes. The three most important ones are the write queue for outgoing packets, the receive queue for incoming packets and the out-of-order queue for packets that arrived with sequence numbers which do not fit into the expected sequence window. However, there are two other ones which are worth mentioning, namely the backlog queue (which is part of the general *sock* structure) and the prequeue (which is TCP specific and therefore maintained in *tcp_sock*). We will show first that copying the write queue, the receive queue and the out-of-order queue is sufficient.

Every incoming packet is pushed upwards on the network stack by the *NET_RX_SOFTIRQ* bottom-half. The backlog queue plays an important role when a socket is locked (for instance by a user application) but there are packets arriving from the network. In this case *NET_RX_SOFTIRQ* will place the packets on the backlog queue in order to prevent bringing the receive queue into an inconsistent state.

On the other hand, there is a *fast-path* receiving mechanism in the Linux network stack which is based on the prequeue. When a user application is waiting on a socket for incoming packets (i.e. it is suspended on a read() system call) it installs itself as a potential thread for performing *fast-path* processing. If the sequence numbers of the incoming packets are matching the criteria of receiving, the actual processing of the packets are put off into the thread's process context, therefore decreasing the amount of time the kernel spends in *NET_RX_SOFTIRQ* bottom-half, which in turn increases the kernel's overall responsiveness. This mechanism serves the purpose of increasing the networking performance of the kernel.

Every socket in the kernel, associated with a connection, resides on two hash tables. The so called *ehash* is responsible for keeping track established connections, while *bhash* contains all sockets that are bound to a local port.

TCPmig ensures that both the backlog and the prequeue are empty during the migration. Firstly, removing the socket from both *ehash* and *bhash* before locking it guarantees that the backlog queue is empty, because every packet gets discarded which doesn't have a matching socket hashed. Secondly, since the migration is initiated by a signal, even if a thread was waiting in a read() system call (and therefore registered itself for processing the prequeue), the system call is abandoned due to the signal and prequeue processing gets disabled before returning to user-space for executing the signal handler.

TCPmig provides a solution for preventing incoming packet loss during checkpoint-restart of a TCP connection, which takes advantage of the broadcast based Single IP Address configuration. The packet capturing feature which is activated on the destination node right before the socket is unhashed on the source node is implemented as a *netfilter* hook in the kernel. Netfilter[13)] provides a facility to attach arbitrary functions to certain phases of the network stack processing. The capturing feature takes place on the *NF_INET_LOCAL_IN* hook, where packets which are to be delivered to the local machine appear. Packets that match with the corresponding remote IP, remote port and local port of the connection being migrated are simply stored on a buffer queue. Note, that duplicated packets (based on sequence numbers) are stored only once.

Re-injection of packets that were captured during migration is discussed below, along with the steps of restarting.

4.1 Checkpointing

Since we have tightly integrated the TCP checkpointing into BLCR, the actual checkpointing is performed while the leader thread dumps the file descriptor table of the process during process checkpointing. Please note, that the packet capture-reinject feature is only activated in case the checkpointing is performed over a socket and not a regular context file. In which case, the main execution steps are the following:

(1) The connection's remote IP address, remote port and local port are collected and a capturing request is sent to the destination node
(2) A status response is read from the (context) socket whether the capturing has been enabled successfully or not
(3) Socket is unhashed and transmission timers are disabled
(4) Relevant fields of the socket structures are copied, queues are iterated, socket buffers serialized and dumped
(5) Data sent to destination node and socket is closed

It is worth noting that second step ensures that the checkpointing and the restarting procedures are synchronized at this point, i.e. the number of packets captured on the destination node is going to be likely small.

4.2 Restarting

Similarly to the checkpoint phase, connection restarting is also part of the BLCR process restarting and it is performed while the leader thread restores the file descriptor table. It happens as follows:

(1) Capture request is read from the context socket, the netfilter is enabled and a status response sent back
(2) Socket structures and queue data are read
(3) A new socket is created and attached to the right file descriptor
(4) Relevant fields are restored as well as buffer queues
(5) IP destination cache entry is recreated
(6) Socket rehashed and timers reactivated, netfilter is still active at this point, thus no incoming packets get delivered to the socket
(7) A tasklet for re-injecting captured packets and disabling netfilter is scheduled

The tasklet scheduled in the last step iterates the capture queue and re-injects each packet to the network stack by calling the netfilter's *okfn()* (in case of IPv4 this is the *ip_rcv_finish()* function).

Please note that there is a race condition between the tasklet and the *NET_RX_SOFTIRQ* bottom-half on the capture queue. For preventing inconsistency, the queue is protected with a spinlock and the tasklet holds it while bottom-halfs are disabled.

The tasklet eventually clears the netfilter and therefore enables the standard packet receiving mechanism on the migrated socket.

4.3 Timestamps

TCPmig addresses the problem of adjusting timestamps on the destination node in order to preserve data transfer seamlessly even after the migration. The Linux TCP implementation uses kernel jiffies for timestamps which is a counter increased approximetaly in every 10 milliseconds. Different nodes can have different jiffies obviously.

Timestamps are recorded during packet transmission and reception and they

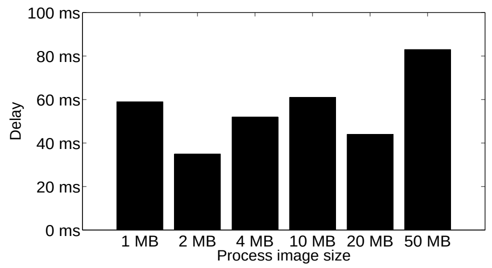

Fig. 5 TCP migration delay according to process image size

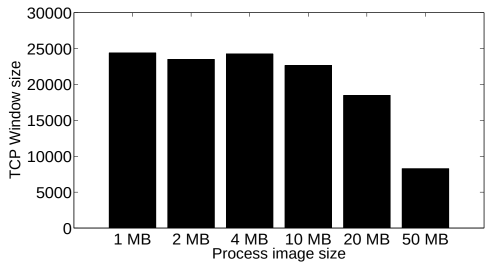

Fig. 6 TCP Window size degradation according to process image size

also form the basis of several TCP related algorithms. Round-trip time measurement or congestion window size adjustment are some of the examples. In order to keep these algorithms working appropriately after the migration occurs, timestamps of the socket structures and buffers have to be updated on the destination node. TCPmig overcomes this problem by recording the jiffies of the source node during the checkpoint, computing the difference on the destination node and adjusting the timestamps of each migrated structure accordingly.

5. Experimental results

We evaluate TCPmig to demonstrate its performance. The test environment is a broadcast based single IP address cluster with two nodes, each node equipped with a 2.2GHz Dual-Core AMD Opteron processor and two gigabytes of RAM. The nodes are connected with a Gigabit Ethernet network for in-cluster communication and they both have a Gigabit Ethernet public interface.

We have migrated processes with different image sizes under high bandwidth network traffic while the migration delay, i.e. the time difference between last packet of the source node and first packet of the destination node has been measured on the peer's side. Changes in the TCP window size has been logged as well. The process being migrated is generating a traffic of 120MB/s.

Figure **Fig. 5** shows the results of the migration delay. As it demonstrates, the delay between the last packet on the source node and the first packet on the destination node changes between 50 and 80 milliseconds regardless of the process' image size. TCPmig ensures that the checkpointing and the restarting processes are synchronized before a socket is migrated, which keeps the packet delay constant.

During normal execution the peek TCP window size scales between 24000 and 25000 in our test environment. As shown in Figure **Fig. 6**, there is no significant window size change during migration for processes with less than 10MB image size.

BLCR transfers the whole process address space before it iterates the file descriptor table, which enlarges the process freeze time in case of a large process image. The long freeze time in turn increases the period while the socket's buffer queues remain unprocessed even though the peer assumes the same communication conditions and sends data with the same bandwidth, for which the Linux TCP reacts with decreasing the window size. Processes over 20MB suffer from this phenomenon significantly.

6. Related work

6.1 Connection migration

TCP migration has been implemented before. NEC corp. proposed transferring TCP sessions between nodes for a distributed Web Server architecture under Linux kernel version 2.4[7]. Their environment assigns each TCP session a virtual IP address which is reported to cause incoming packet loss due to the need of updating the ARP table on the router each time a connection is migrated.

SockMi[14] offers TCP migration with IP layer forwarding between the source and the target node, therefore it is not feasible for addressing fault tolerance in a cluster environment. Furthermore, their implementation requires application specific support for exporting and importing connections. Tcpcp[15] provides similar capabilities to SockMi, where the source node establishes an IP layer forwarding mechanism to the destination after the migration takes place. However, Tcpcp is implemented as a kernel patch. Earlier forwarding based solutions were also proposed in MobileIP[16] and MSOCKS[17].

TCP Migrate option[18] is an extension to the TCP protocol in order to support session migration. The transfer can be initiated by sending a special migrate SYN packet with a previously arranged token in order to reestablish the connection. A major drawback of this solution is that the peer must also support the protocol extension.

Reliable sockets (ROCKS) and reliable packets (RACKS)[19] both offer transparent network connection mobility using only user-level mechanisms. They can detect a connection failure, preserve the endpoint of a failed connection in a suspended state and automatically reconnect. However, they both need the presence of the extended socket library on each side of the connection.

6.2 Process migration

Process migration has been researched actively and several distributed operating systems offer the capability of migrating processes. V-System[20], Amoeba[21], Mach[22], Sprite[23][24], MOSIX[25] or OpenSSI[26] are some of the examples, although connection migration is supported in a very limited way. Amoeba provides connection migration, but it restricts the implementation for dealing explicitly with RPC communications, which are layered on the lower level FLIP protocol[27] instead of TCP/IP.

BLCR[12] is an open source checkpoint-restart library for Linux, which can be used for migrating processes. BLCR currently does not support connection migration and reportedly does not intend to include this feature in its forthcoming releases.

Zap[28] implements a thin virtualization layer on top of the operating system which provides the facility of migrating a group of processes. Zap's VNAT[29] mechanism for virtualizing network resources supports connection migration. Its main drawback is that it requires the Zap module to be loaded on the client side as well.

NEC reports[7] the integration of their TCP migration mechanism with process migration, however they do not provide detailed information about the process migration itself.

7. Conclusion and future perspectives

We have developed TCPmig, a mechanism for migrating processes with TCP connections in a Single IP Address cluster. Preliminary test results show that the migration is smooth (the packet delay caused by the migration is independent from the process' image size) even under high bandwidth network traffic. TCPmig's performance is proven to be good enough for applying it in real world scenarios.

In the future we intend to further evaluate the performance of TCPmig, especially by applying it to real world applications. Building load balanced and fault tolerant distributed Web or streaming servers, reducing overall cluster energy consumption and providing a distributed file system service built upon a broadcast based Single IP Address cluster are some of our future perspectives.

A high-level scheduling protocol is to be designed and developed for supervising decisions regarding which processes should migrate to which node. Decisions can be drawn with respect to different policies, according to the goal of the given system.

Acknowledgements

This work has been supported by the CREST project of the Japan Science and

Technology Agency (JST).

References

1) Feng, W.-C. and Cameron, K.: The Green500 List: Encouraging Sustainable Supercomputing, *Computer*, Vol.40, No.12, pp.50–55 (2007).
2) Buyya, R., Cortes, T. and Jin, H.: Single System Image, *Int. J. High Perform. Comput. Appl.*, Vol.15, No.2, pp.124–135 (2001).
3) Zhang, W.: Linux Virtual Servers for Scalable Network Services, *Linux Symposium* (2000).
4) O'Rourke, P. and Keefe, M.: Performance Evaluation of Linux Virtual Server, *LISA 2001 15th Systems Administration Conference* (2001).
5) Dias, D. M., Kish, W., Mukherjee, R. and Tewari, R.: A scalable and highly available web server, *COMPCON '96: Proceedings of the 41st IEEE International Computer Conference*, Washington, DC, USA, IEEE Computer Society, pp.85–92 (1996).
6) Matsuba, H. and Ishikawa, Y.: Single IP address cluster for internet servers, *Proceedings of 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS2007)* (2007).
7) Takahashi, M., Kohiga, A., Sugawara, T. and Tanaka, A.: TCP-Migration with Application-Layer Dispatching: A New HTTP Request Distribution Architecture in Locally Distributed Web Server Systems, *Communication System Software and Middleware, 2006. Comsware 2006. First International Conference on*, pp. 1–10 (2006).
8) Microsoft: Network Load Balancing Technical Overview, `http://www.microsoft.com/technet/prodtechnol/windows2000serv/deploy/confeat/nlbovw.mspx`.
9) Damani, O.P., Chung, P.E., Huang, Y., Kintala, C. and Wang, Y.-M.: ONE-IP: techniques for hosting a service on a cluster of machines, *Selected papers from the sixth international conference on World Wide Web*, Essex, UK, Elsevier Science Publishers, Ltd., pp.1019–1027 (1997).
10) Vaidya, S. and Christensen, K.J.: A Single System Image Server Cluster using Duplicated MAC and IP Addresses, *Proceedings of the 26th Annual IEEE Conference on Local Computer Networks*, pp.206–214 (2001).
11) Takigahira, T.: Hive server: high reliable cluster Web server based on request multicasting, *Proceedings of The Third International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'02)*, pp. 289–294 (2002).
12) Duell, J.: The design and implementation of Berkeley Lab Linux Checkpoint/restart, Technical report, Lawrence Berkeley National Laboratory (2000).
13) Linux Netfilter: Firewall, NAT, and packet mangling for Linux, `http://www.netfilter.org`.
14) Bernaschi, M., Casadei, F. and Tassotti, P.: SockMi: a solution for migrating TCP/IP connections, *Parallel, Distributed and Network-Based Processing, 2007. PDP '07. 15th EUROMICRO International Conference on*, pp.221–228 (2007).
15) Almesberger, W.: TCP Connection Passing (2004). Ottawa Linux Symposium.
16) Bhagwat, P., Perkins, C. and Tripathi, S.: Network Layer Mobility: an Architecture and Survey, *IEEE Personal Communication, vol. 3, no. 3*, pp.54–64 (1996).
17) Maltz, D.A. and Bhagwat, P.: MSOCKS: An Architecture for Transport Layer Mobility, *Proceedings of the IEEE INFOCOM*, pp.1037–1045 (1998).
18) Snoeren, A. and Balakrishnan, H.: An End-to-End Approach to Host Mobility, *Proc. MobiCom '00*, pp.155–166 (2000).
19) Zandy, V.C. and Miller, B.P.: Reliable Network Connections, *Proceedings of the 8th International Conference on Mobile Computing and Networking*, pp.95–106 (2002).
20) Theimer, M., Lantz, K.A. and Cheriton, D.R.: Preemptable remote execution facilities for the V-System, *ACM SIGOPS Operating Systems Review*, pp.2–12 (1985).
21) Mullender, S.J., van Rossum, G., Tanenbaum, A.S., van Renesse, R. and van Staveren, H.: Amoeba – A Distributed Operating System for the 1990s, *IEEE Computer, vol. 23, no. 5*, pp.44–53 (1990).
22) Accetta, M.J., Baron, R.V., Bolosky, W.J., Golub, D.B., Rashid, R.F., Tevanian, A. and Young, M.: Mach: A New Kernel Foundation for UNIX Development, *Proceedings of the Summer 1986 USENIX*, pp.93–112 (1996).
23) Ousterhout, J.K., Cherenson, A.R., Douglis, F., Nelson, M.N. and Welch, B.B.: The Sprite Network Operating System, *IEEE Computer vol.21, no. 2*, pp. 23–36 (1988).
24) Douglis, F. and Ousterhout, J.: Transparent Process Migration: Design Alternatives and the Sprite Implementation, *Software - Practice and Experience, vol. 21, no. 8*, pp.757–785 (1991).
25) Barak, A. and Wheeler, R.: MOSIX: An Integrated Multiprocessor UNIX, *USENIX Winter Technical Conference*, pp.101–112 (1989).
26) OpenSSI: Open Single System Image Clustering Project, `http://ssic-linux.sourceforge.net`.
27) Steketee, C.: Process Migration and Load Balancing in Amoeba (1999).
28) Osman, S., Subhraveti, D., Su, G. and Nieh, J.: The design and implementation of Zap: A system for migrating computing environments, *In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pp.361–376 (2002).
29) Su, G. and Nieh, J.: Mobile Communication with Virtual Network Address Translation, Technical report (2002).