IPSJ Transactions on Advanced Computing Systems Vol. 3 No. 1 1–12 (Mar. 2010)

*Regular Paper*

# Live Migration of Processes Maintaining Multiple Network Connections

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 for providing services seamlessly and for increasing sustainability. In this paper we propose a live migration mechanism which is capable of moving processes that maintain a massive amount of network connections, supporting both TCP and UDP sockets. Incoming packet loss during socket migration is prevented by exploiting the broadcast property of the Single IP Address cluster, while process live migration minimizes the execution freeze time during the actual migration of the process context. Performance evaluation on machines equipped with a 2.4 GHz CPU and Gigabit Ethernet interconnect shows that migrating a process of 1 GB image size and over 1000 established network connections results in less than 200 ms process freeze time, rendering the transition fully transparent and responsive from the clients' point of view. The implementation is comprised entirely of a kernel module for Linux 2.6, without any changes to the existing kernel code.

## 1. Introduction

Several network based applications, such as massively multi-player online games (MMOG) [1], networked virtual environments (NVE) [2] and distributed simulations like the High Level Architecture (HLA) [3] maintain persistent network connections. While providing these services using clusters of inexpensive commodity computers has become commonplace, addressing scalability, reliability and sustainability is still challenging in such environments.

Process migration is a mechanism which decouples an application from the physical machine that is executing it, from the *source node* and allows the process to continue running on a separate computer, on the *destination node*. It can be used for load balancing to address scalability, it can be utilized as a foundation of fault tolerance making the system more reliable and it also allows the deallocation of computers which can decrease the overall power consumption [4].

Migrating applications that maintain TCP/IP or UDP/IP 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 were a single computer on the network. This makes it possible to migrate processes inside the cluster even with network connections, practically without the client noticing it.

Providing a single IP address can be realized in two ways, **Fig. 1** depicts the difference between these approaches. 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 [5),6)]. However, the problem in this case is that each time a connection is moved, the router has to be updated in order to reflect the new IP to MAC address mapping. This extra administration reportedly leads to a noticeable decrease in the process's responsiveness, besides it can potentially cause incoming packet loss since the migration is not an anticipated event on the client side [7].

To the contrary, in a broadcast based cluster each node is equipped with a pub-

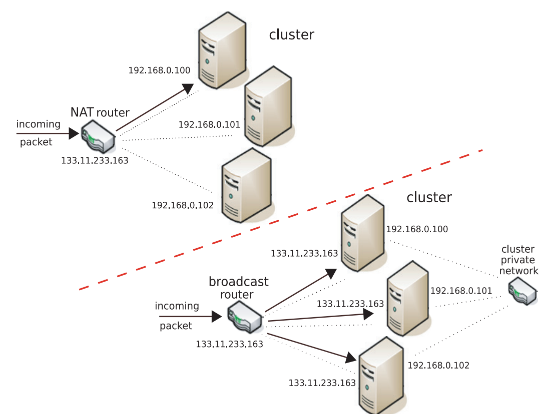



**Fig. 1** Comparison between NAT and broadcast based Single IP Address Clusters.





© 2010 Information Processing Society of Japan

lic and a local interface. The same IP address is assigned to public interfaces and the local ones are used for in-cluster communication. The router simply broadcasts each incoming packet to the whole cluster [8),9)]. The broadcast property of this model makes it possible to migrate a connection without any extra effort on the router and proper co-operation between the source and the destination nodes allows capturing packets that might arrive during the migration.

The main difference between the two cluster setups lies in the way new connections are accepted. In case of broadcast based clusters, a cluster-coordinated method can be realized at the operating system level, without posing any restrictions on the type of deployable services [10)].

Further focusing on responsiveness, process migration can be categorized in two cases considering the process freeze time required, *frozen migration* and *live migration* [4)]. In case of *frozen migration*, the process gets suspended, its memory image is transferred to the destination node where the execution is then restarted. The foundation of this mechanism is called *checkpoint-restarting*. On the other hand, *live migration* lets the application proceed with its execution while it asynchronously transfers most of the process image. Subsequently, it tracks and sends incremental updates of the data changed in memory until a predefined condition is met, when the execution context is moved to the destination node. This solution is based on *incremental checkpoint-restarting*.

This paper makes the following contributions: a process migration technique is designed that is capable of moving processes with *massive amount of network connections*; *UDP and TCP sockets* in established or listening states are supported; *incoming packet loss is prevented* by exploiting the broadcast based cluster and therefore rendering the transition fully transparent on the peer's side. Furthermore, we show how process *live migration* enhances responsiveness at the socket level.

The rest of the paper is organized as follows, Section 2 presents basic background on process migration and introduces the Berkeley Checkpoint-Restart Library (BLCR) [11)], which we have modified to support live migration of processes with TCP/UDP connections. Section 3 describes the design and Section 4 details the implementation of the socket migration mechanism. The process live migration technique and dirty page tracking are explained in Section 5. Performance evaluation is given in Section 6 and related work is discussed in Section 7. Finally, Section 8 concludes the paper.

## 2. Background

In this section the BLCR [11)] checkpoint-restart library is introduced first, providing an overview of the main steps of the checkpoint and restart procedures. The act of process migration, based on checkpoint-restarting is then described.

### 2.1 Berkeley Checkpoint-Restart library

The Berkeley Checkpoint-Restart library (BLCR) [11)] 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 for instance by executing the application via a special tool provided by the BLCR package.

#### 2.1.1 Checkpointing

The BLCR checkpoint library installs a dedicated signal handler to make an application checkpointable. **Figure 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 to the libraries character special file to enter the kernel-space.
(3) The threads are synchronized and one is chosen as the leader.
(4) The leader dumps thread relations, memory mappings and file descriptors.
(5) Each thread writes its registers, signal handlers and its 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. 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.

#### 2.1.2 Restarting

Restarting an application is mainly the mirror procedure of checkpointing. The restart utility opens the context file, which can be practically establishing a connection via a socket, and executes the following steps. It forks a new process for the application being restarted, which clones itself to have as many threads



© 2010 Information Processing Society of Japan

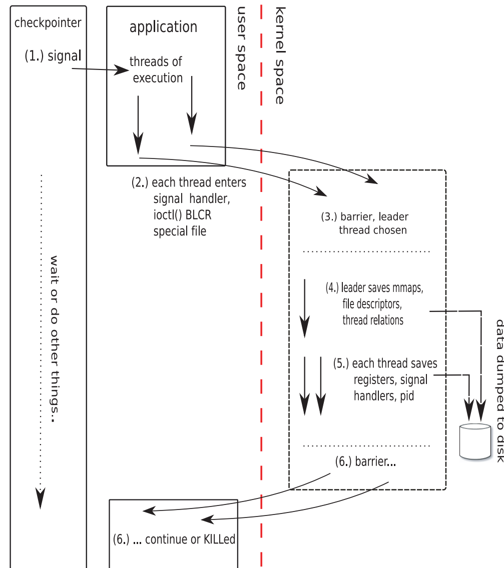



**Fig. 2** BLCR checkpoint mechanism.

of execution as the checkpointed application contained. Each thread issues an ioctl() call to the library's character special file to enter kernel-space and one thread is chosen as the leader. The leader thread restores process-wide resources: open files, memory maps and thread relations. Each thread then restores its pid, the signal handlers and the registers. The restarter process finally gets notified and all threads return to user-space where they eventually resume their original execution.

### 2.2 Process Migration

Process migration is realized as a synchronized checkpoint/restart event between the source and the destination nodes inside the cluster.

The source node establishes a network connection to the destination node and checkpoints the target process. However, instead of writing the process resources into a regular file, it transfers them through the connection directly to the destination node. The destination node in turn executes the restart procedure, but, instead of reading data from a checkpoint file, it receives all process resources through the network connection.

Process live migration involves further complexity on the way checkpointing is executed and will be described in Section 5.

## 3. Socket Migration Design

Socket migration is tightly integrated into the process migration mechanism, therefore it also has two phases, checkpointing and restarting a connection.

A general socket migration mechanism is executed in both TCP and UDP cases, which is further specialized according to the underlying protocol, differing mainly on the data structures the kernel is using for their representations.

For a certain period of time during the migration the socket is not available for processing incoming traffic, which can potentially cause losing some of the receiving packets. A mechanism for preventing incoming packet loss is realized by capturing the packets on the destination node while the migration is carried out. For further details on how packet capturing is implemented refer to Section 4.2.3. UDP sockets are not meant to provide a reliable data transfer, thus prevention of incoming packet loss is only enabled in the case of TCP connections.

### 3.1 Checkpointing

The actual socket checkpointing is performed while the leader thread iterates the file descriptor table and dumps the open files of the process. The main execution steps are the following.

The connection's remote IP address, remote port and local port are collected and a capturing request for preventing incoming packet loss is sent to the destination node. A status response is read from the (context) socket whether packet capturing has been enabled successfully or not. The socket is deactivated (so that packets cannot be delivered to it), relevant fields of the socket structures are copied, queues are iterated, buffers and dumped, all data are sent to the destination node and finally the socket is closed.

Notice that the checkpointing and restarting procedures are synchronized during the activation of the packet capturing, which happens directly before the socket gets unhashed, thus, ensuring that the number of packets captured on the destination node is small.


© 2010 Information Processing Society of Japan

### 3.2 Restarting

Similarly to the connection checkpointing phase, connection restarting is part of the process restarting and it is performed while the leader thread iterates the file descriptor table for restoring the process' open files.

First, the packet capturing request is read from the context socket, installed and a status message is sent to the source node. Socket structures and queue data are then read, a new socket is created. The relevant fields are restored, as well as the buffer queues. The socket gets rehashed and timers are reactivated. Packet capturing is still active at this point, thus no incoming packets are delivered to the socket. The captured packets are then re-injected and capturing gets disabled.

## 4. Socket Migration Implementation

In order to describe socket migration in technical details, we provide an overview of the Linux socket infrastructure. The main kernel structures and the relations among them are introduced first. Linux specific aspects of the TCP/UDP migration are then discussed along with the implementation of the packet capturing feature.

### 4.1 Linux Socket Infrastructure

There are several data structures used in the Linux kernel for representing and maintaining TCP/UDP connections. Each open file of a process is referred to as a *file* struct from the process's file descriptor table. The directory cache entry field of a *file* 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 specialized 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 various 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, the TCP state, fields for RTT measurement, it controls the slow start mechanism and so on.

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

UDP sockets are, on the other hand, represented by a *udp_sock* structure which is built on top of *inet_sock*.

### 4.2 TCP Migration

The Linux based implementation of the TCP socket migration is explained in this section. We show which data structures have to be transferred and detail how checkpointing and restarting are performed.

#### 4.2.1 Checkpointing

The Linux kernel maintains several socket buffer queues for representing TCP connections. 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 have to be taken into account. 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 ensure that both the backlog and the prequeue are empty during the migration and therefore, copying the write queue, the receive queue and the out-of-order queue is sufficient.

**Backlog queue:** 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



© 2010 Information Processing Society of Japan

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. If the socket is not locked, incoming segments are processed and placed on the receive queue. On one hand, the socket is surely not locked by any user process, because each thread of the application returns to userspace during the execution of the signal handler (i.e., releases the socket even if it was processing a read()/write() system call). On the other hand, removing the socket from both *ehash* and *bhash* before locking it, guarantees that the backlog queue is empty, because each packet that does not have a matching socket hashed gets discarded.

**Prequeue:** There is a so called *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 number of the incoming packet is matching the criteria of receiving, the packet size is not longer than the user process's request and the current process is the one registered for *fast-path* processing, the actual processing of the packet is put off into the thread's process context. This mechanism decreases the amount of time the kernel spends in *NET_RX_SOFTIRQ* bottom-half, which in turn increases the kernel's overall responsiveness. Since the migration is initiated by a signal, it ensures that even if a thread was waiting in a read() system call (and therefore registered itself for processing the prequeue), the system call is abandoned and prequeue processing gets disabled before returning to user-space for executing the signal handler.

Checkpointing a Linux TCP socket is performed as follows. First the socket is unhashed from both the *ehash* and *bhash* hashtables. The retransmission timer of the write queue is then cleared. Relevant fields of the socket representation structures, such as the *tcp_sock*, *inet_connection_sock* and *inet_sock* are extracted. The receive, write and out-of-order queues are iterated and the *sk_buff* structures are linearized. All data are transferred to the destination node and finally, the socket queues are purged and the socket is closed.

#### 4.2.2 Restarting

The following steps are executed for restarting a TCP socket. First, a new socket is allocated and data are received from the source node. The relevant fields of the socket representation structures are updated. *sk_buff* structures of the receive, write and out-of-order queues are allocated, updated and re-inserted. An IP route destination entry is created for the socket. The socket gets rehashed for both the *ehash* and *bhash* hashtables. The retransmission timer is then restarted. Finally, the socket is attached to the right file descriptor of the process.

##### 4.2.2.1 TCP Timestamps

Adjustment of timestamps on the destination node is inevitable 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 approximately every 10 milliseconds. Different nodes can have different jiffies.

Timestamps are recorded during packet transmission and reception and they 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. We overcome 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 affected structure accordingly.

#### 4.2.3 Preventing Incoming Packet Loss

As it was described in Section 3 a solution for preventing incoming packet loss while checkpoint-restarting TCP connections is proposed. The mechanism 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 [12] provides a facility to attach arbitrary functions to certain phases of the network stack processing. Kernel modules can register to listen at five different hooks, of which two are of interest for incoming traffic. *NF_IP_PRE_ROUTING* is responsible for packets that have passed the basic sanity checks (i.e., not truncated, IP checksum OK, etc), whereas packets that are to be delivered to the localhost are passed to the *NF_IP_LOCAL_IN* hook.

Hook functions receive packets and return a decision for the netfilter framework.



© 2010 Information Processing Society of Japan

Some of the possible decisions are *NF_ACCEPT*, that lets the packet continue traversal, *NF_DROP* drops the packet and frees its resources, while *NF_STOLEN* takes over the packet. Both *NF_DROP* and *NF_STOLEN* prevent the packet from being passed to upper layers.

The capturing feature takes place on the *NF_INET_LOCAL_IN* hook. Packets that match the corresponding remote IP, remote port and local port of the connection being migrated are simply stored on a buffer queue, i.e., the decision *NF_STOLEN* is made. Duplicated packets (based on sequence numbers) are dropped, *NF_DROP* is returned, while packets that are not related to the connection are simply accepted.

After the TCP socket is successfully migrated, the re-injection phase iterates the capture queue and submits 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 re-injection and the *NET_RX_SOFTIRQ* bottom-half on the capture queue. For preventing inconsistency, the queue is protected with a spinlock and bottom-halfs are disabled while the lock is held.

The netfilter is eventually cleared which enables the standard packet receiving mechanism on the migrated socket.

### 4.3 UDP Migration

Migrating UDP sockets is considerably easier than TCP. Besides the main UDP socket data structure, we only transfer the socket buffers residing on the receive queue and the *inet_options* structure if the structure is present.

There is an issue, however, that is worth noting with respect to UDP server sockets, that are bound to a local port. Each UDP server socket has to be unhashed before the migration takes place and consequently it has to be rehashed on the destination node.

## 5. Process Live Migration

We have extended BLCR for supporting live migration. Our mechanism is based on incrementally dumping address space changes in a helper thread, while letting the application proceed with its original execution. **Figure 3** illustrates the main steps of performing live checkpointing, which are the following:

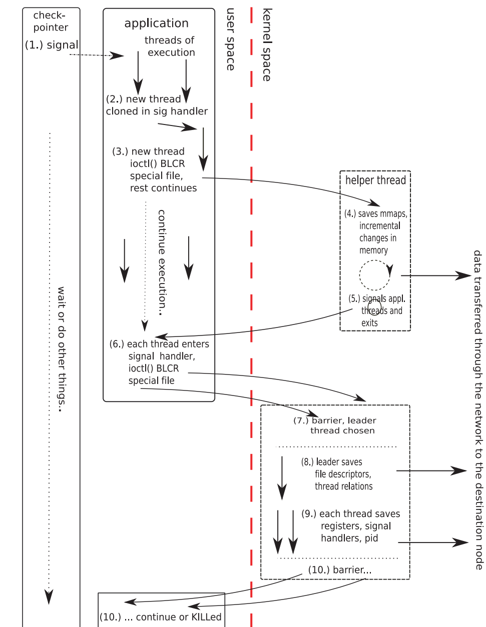



**Fig. 3** Live migration mechanism.

( 1 ) The application receives the signal of a live checkpointing request.
( 2 ) It clones a new thread and all the application threads simply return from the signal handler (i.e., continue their execution).
( 3 ) The new thread enters the kernel via an ioctl() call.
( 4 ) It transfers memory mappings to the destination node and enters a loop of tracking address space changes, where in each subsequent iteration the loop timeout is decreased and the number of the transferred bytes is logged.
( 5 ) When it reaches a limit on the number of bytes transferred it signals the application threads for final checkpointing.
( 6 ) Executing the signal handler, each application thread enters the kernel through an ioctl() call.
( 7 ) All threads enter a barrier and a leader thread is chosen.
( 8 ) The leader transfers the file table, the file descriptors (this is where the


© 2010 Information Processing Society of Japan

socket migration takes place) and thread relations.

(9) Each thread transfers registers, signal handlers and its pid.

(10) They enter a final barrier, all of them return to userspace where they finish up the signal handler and continue their execution or are killed depending on the options specified.

As for the destination node the restarting procedure is very similar to the one in case of frozen migration. After the leader thread is chosen among the threads being restarted, it receives incremental updates from the source node and applies these to the process's address space. When the incremental phase is finished the restarting procedure follows the same steps as in the frozen migration.

Incremental checkpointing can also be initiated directly from the kernel, without notifying the application [13]. However, an advantage of the signal based approach is that even if a thread executes a system call, and therefore may lock important kernel structures, it will abandon it and return to userspace. For further details on how socket migration benefits from this, please refer to Section 4.2.1.

### 5.1 Tracking Dirty Pages

Carrying out incremental checkpointing is built upon the mechanism of tracking dirty pages between subsequent updates. Currently, two main approaches exist, one manipulates the write bit of the page-table entries (PTE), while the other utilizes the dirty bit [14].

We opted for the approach of using the dirty bit and relaxing the swap facility, which is reportedly not a major restriction in HPC environments [14]. Using directly the dirty bit allows us having the incremental checkpoint mechanism entirely implemented in a kernel module, without any changes to existing kernel code.

## 6. Experimental Results

We evaluate our live migration mechanism to the assess packet delay at the socket level depending on the process image size; the process freeze time in case of frozen and live migration and the process freeze time according to the number of network connections maintained. The test environment is a broadcast based single IP address cluster with two nodes, each node is equipped with a 2.4 GHz 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.

### 6.1 Packet Delay and TCP Window Size Changes

We migrate processes with different image sizes under a high bandwidth network traffic while the migration delay, i.e., the time difference between the last packet of the source node and the first packet of the destination node, as well as changes in the TCP window size, were measured on the peer's side. Each experiment is evaluated three times and the average result is presented. We apply both frozen and live migration in order to show how the socket migration is affected by the technique utilized at the process level. The process being migrated is generating a traffic of 120 MB/s.

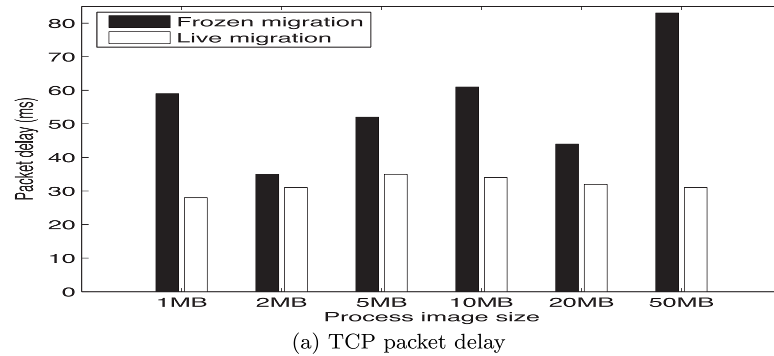
**Figure 4** (a) shows the results of the migration delay according to the process image size and the migration technique applied. As it demonstrates, the delay between the last packet on the source node and the first packet on the destination node changes between 50 and 80 milliseconds in case of frozen migration. To the contrary, live migration keeps the packet delay constant 30 milliseconds regardless of the process image size.

During normal execution the peek TCP window size oscillates between 24000 and 25000 in our test environment. As shown in Fig. 4 (b), there is a significant window size degradation for processes over 10 MB image size. In the case of frozen migration, the whole process address space is transferred before the file descriptor table is iterated, which prolongs the process freeze time. The long freeze time in turn increases the period during which 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 by decreasing the window size.

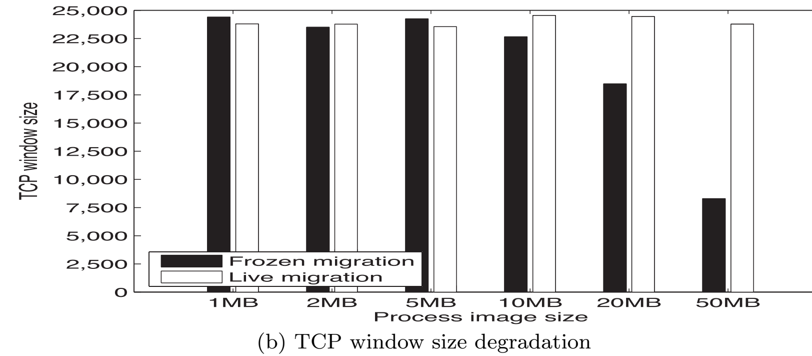
Live migration lends itself naturally as a remedy for this problem, it transfers most of the process image asynchronously, leaving only a small portion of data to be transferred during the actual context transition. Figure 4 (b) reflects this by showing that there is no window size degradation even with bigger process image sizes.



© 2010 Information Processing Society of Japan




(a) TCP packet delay




(b) TCP window size degradation

**Fig. 4** Packet delay and TCP window size degradation.

### 6.2 Multiple Connections

We evaluate our migration technique by applying it to processes that maintain several network connections. In each case we are aiming at simulating an interactive server. The server process executes a loop of receiving 1 kB data from all of its clients, performing some computations and sending the data back. Moreover, in each iteration a random amount of pages are dirtied. Client processes are simply sending and receiving data in an infinite loop.

Since we are not aware of the typical characteristics of MMOG or NVE server processes in terms of memory need due to their proprietary nature, we present measurements for a wide scale of attributes. Memory footprint size scales from 5 MB to 50 MB in the case of frozen migration, whereas in the case of live migration results are presented up to 1 GB of process image size. The number of connections scales in both cases from 2 up to 1024.

**Figure 5** (a) shows the overall duration of frozen migration with different process image sizes and number of network connections, where the process image size grows up to 50 MB. In contrast, overall live migration duration is shown on Fig. 5 (b) for processes with 32 MB, 64 MB and 128 MB memory footprint size, while Fig. 5 (c) demonstrates the same experiment for processes with 256 MB, 512 MB and 1 GB of image size.

Notice that although frozen migration takes shorter time, the migration duration is equal to the process freeze time in this case. Reaching over 600 ms even with only 50 MB image size is not acceptable for an interactive service.

Figure 5 (d) depicts the actual process freeze time in the live migration case. As it shows process freeze time is less than 200 ms for 1 GB process image size and over 1000 network connections, while maintaining approximately 500 connections the actual transition results in less than 100 ms. Consequently, live migration process freeze time is significantly shorter than a frozen migration process freeze time. Process freeze time in this range ensures full transparency and a high responsiveness even for an interactive server.

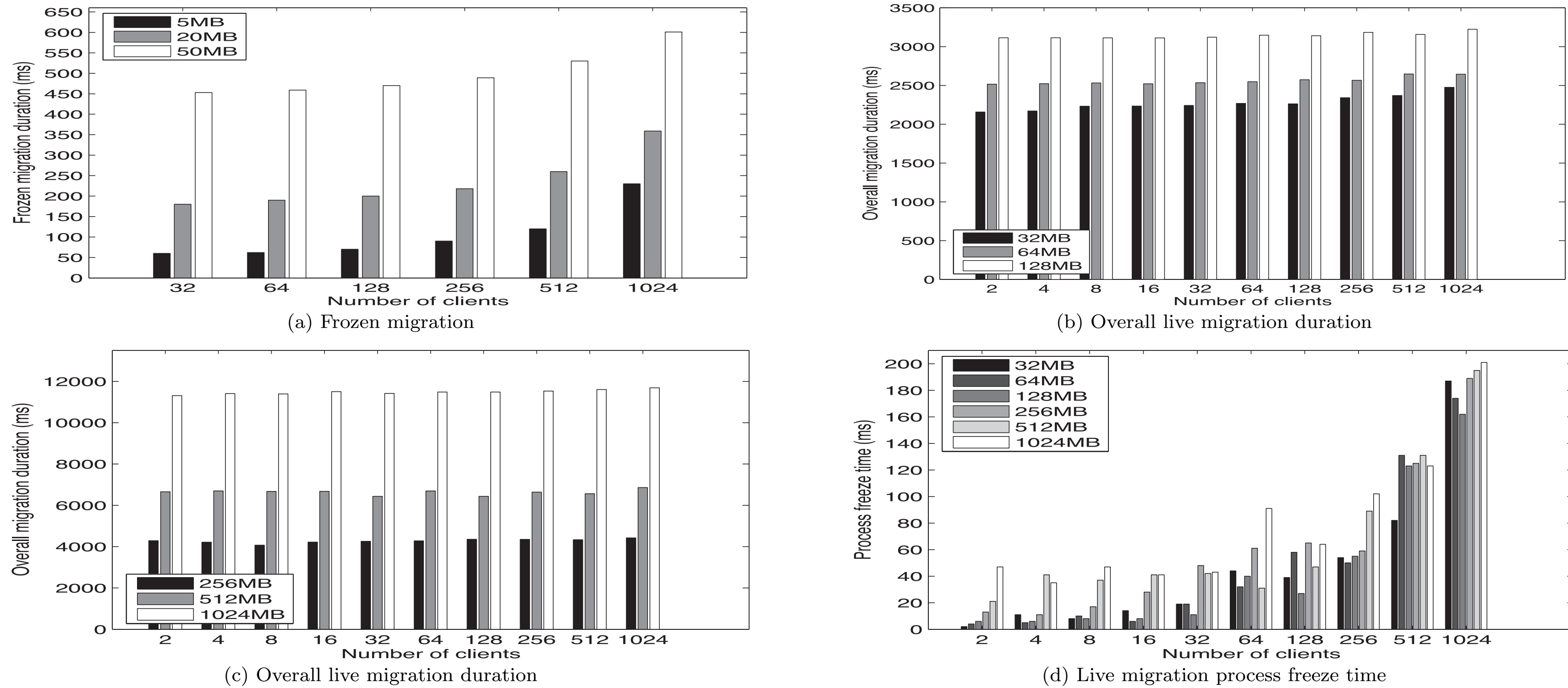
## 7. Related Work

### 7.1 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 during the migration.

SockMi [15] 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 [16] provides similar capabilities to SockMi, where the source node establishes an IP layer forwarding mechanism to the destination after the migration takes place. However,

 © 2010 Information Processing Society of Japan




(a) Frozen migration

(b) Overall live migration duration

(c) Overall live migration duration

(d) Live migration process freeze time

**Fig. 5** Comparison between frozen and live migration.

Tcpcp is implemented as a kernel patch. Earlier forwarding based solutions were also proposed in MobileIP [17] and MSOCKS [18].

TCP Migrate option [19] 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) [20] 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 require the extended socket library on each side of the connection.

### 7.2 Process Migration

Process migration has been researched actively and several distributed operating systems offer the capability of migrating processes. V-System [21], Amoeba [22], Mach [23], Sprite [24),25], MOSIX [26] or OpenSSI [27] 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 [28] instead of TCP/IP.



© 2010 Information Processing Society of Japan

BLCR [11] is an open source checkpoint-restart library for Linux, which can be used for migrating processes. BLCR currently does not support connection migration, neither incrementally dumping address space changes.

Zap [29] implements a thin virtualization layer on top of the operating system which provides the facility of migrating a group of processes, called pods. Zap's VNAT [30] mechanism for virtualizing network resources supports connection migration. Its main drawback is that it requires the Zap VNAT mechanism to be present also on the client side in order to map the virtual address to the new remote physical address after the migration.

Incremental checkpoint/restart has been proposed by several recent studies [13),14),31)]. While they all offer the benefit of process live migration, none of them deals with sockets, therefore lacking the ability of migrating processes that maintain network connections.

NEC reports [7] the integration of their TCP migration mechanism with process frozen migration, however they do not deal with multiple connections. Besides, no details are revealed regarding the process migration itself.

### 7.3 Virtual Machine Migration

Virtual machine (VM) migration is an actively researched topic in recent years. Solutions based on Xen [32], KVM [33] and VMware's VMotion [34] also provide the ability of live migrating VM instances.

Due to its clear separation of the OS from the underlying hardware VM migration naturally eliminates the problem of "residual dependencies", which is an advantage compared to migration at the process level [4]. While several Single System Image (SSI) systems leave residual dependencies on the source node after a process is migrated, such as network connections are routed through, or certain system calls are still forwarded back to the source node, our proposed solution transfers all the dependencies of the process, allowing the source node to be possibly switched off after the migration has taken place.

It has also been showed that conducting VM live migration while keeping network connections alive gives comparable service downtime to process level live migration [32]. However, no results are provided for the case where a massive amount of connections are involved.

MMOG and NVE services often divide the virtual space (referred as zoning) and let each zone be handled by a separate server process [35]. Aiming at having each zone-server as a migratable unit of the system, a disadvantage of VM based solutions lies in the fact that each server needs to reside in its own VM, leading to unnecessary allocation of resources.

## 8. Conclusion and Future Perspectives

We have proposed a process live migration technique which is capable of migrating processes that maintain massive amount of network connections. Both TCP and UDP connections are supported.

Test results showed that the migration is efficient enough for moving real-world applications. At the socket level, keeping the packet delay around 30 milliseconds, allows us to move connections with a high bandwidth network traffic while rendering the transition fully transparent on the client's side. At the process level, incrementally transferring address space changes enables a smooth migration even if multiple network connections are maintained. We showed that live migrating a process with a gigabyte size address space and over 1000 network connections results in less than 200 milliseconds process freeze time.

In the future we intend to further improve the service downtime and to apply our migration mechanism to applications, such as massively multi-player online game servers, networked virtual environments, distributed simulations or media streaming solutions.

MMOG and NVE servers often maintain in-cluster connections, for instance with database servers. Since we have control over all in-cluster machines, we are planning to investigate how these connections could be kept alive. Our envisioned solution is based on applying IP level filters on both ends of the connection in order to translate the new local IP address properly.

**Acknowledgments** This work has been supported by the CREST project of the Japan Science and Technology Agency (JST).

## References

1) Martin, D., Moorsel, A.v. and Morgan, G.: Efficient Resource Management for Game Server Hosting, *ISORC '08: Proc. 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, Washington, DC, USA, pp.593–596,



© 2010 Information Processing Society of Japan

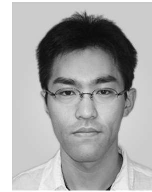
IEEE Computer Society (2008).
2) Singhal, S. and Zyda, M.: *Networked virtual environments: Design and implementation*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1999).
3) Cai, W., Yuan, Z., Low, M.Y.H. and Turner, S.J.: Federate migration in HLA-based simulation, *Future Gener. Comput. Syst.*, Vol.21, No.1, pp.87–95 (2005).
4) Milojičić, D.S., Douglis, F., Paindaveine, Y., Wheeler, R. and Zhou, S.: Process migration, *ACM Comput. Surv.*, Vol.32, No.3, pp.241–299 (2000).
5) Zhang, W.: Linux Virtual Servers for Scalable Network Services, *Linux Symposium* (2000).
6) O'Rourke, P. and Keefe, M.: Performance Evaluation of Linux Virtual Server, *LISA 2001 15th Systems Administration Conference* (2001).
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, pp.1019–1027, Elsevier Science Publishers, Ltd. (1997).
10) Fujita, H., Matsuba, H. and Ishikawa, Y.: TCP Connection Scheduler in Single IP Address Cluster, Washington, DC, USA, *IEEE Computer Society*, pp.366–375 (2008).
11) Duell, J.: The design and implementation of Berkeley Lab Linux Checkpoint/restart, Technical report, Lawrence Berkeley National Laboratory (2000).
12) Linux Netfilter: Firewall, NAT, and packet mangling for Linux. http://www.netfilter.org
13) Gioiosa, R., Sancho, J.C., Jiang, S. and Petrini, F.: Transparent, Incremental Checkpointing at Kernel Level: A Foundation for Fault Tolerance for Parallel Computers, *SC '05: Proc. 2005 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, p.9, IEEE Computer Society (2005).
14) Wang, C., Mueller, F., Engelmann, C. and Scott, S.L.: Proactive process-level live migration in HPC environments, *SC '08: Proc. 2008 ACM/IEEE conference on Supercomputing*, Piscataway, NJ, USA, pp.1–12, IEEE Press (2008).
15) 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).
16) Almesberger, W.: TCP Connection Passing (2004). Ottawa Linux Symposium.
17) 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).
18) Maltz, D.A. and Bhagwat, P.: MSOCKS: An Architecture for Transport Layer Mobility, *Proc. IEEE INFOCOM*, pp.1037–1045 (1998).
19) Snoeren, A. and Balakrishnan, H.: An End-to-End Approach to Host Mobility, *Proc. MobiCom '00*, pp.155–166 (2000).
20) Zandy, V.C. and Miller, B.P.: Reliable Network Connections, *Proc. 8th International Conference on Mobile Computing and Networking*, pp.95–106 (2002).
21) 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).
22) 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).
23) 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, *Proc. Summer 1986 USENIX*, pp.93–112 (1996).
24) 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).
25) 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).
26) Barak, A. and Wheeler, R.: MOSIX: An Integrated Multiprocessor UNIX, *USENIX Winter Technical Conference*, pp.101–112 (1989).
27) OpenSSI: Open Single System Image Clustering Project. http://ssic-linux.sourceforge.net
28) Steketee, C.: Process Migration and Load Balancing in Amoeba (1999).
29) Osman, S., Subhraveti, D., Su, G. and Nieh, J.: The design and implementation of Zap: A system for migrating computing environments, *Proc. Fifth Symposium on Operating Systems Design and Implementation*, pp.361–376 (2002).
30) Su, G. and Nieh, J.: Mobile Communication with Virtual Network Address Translation, Technical report (2002).
31) Mehnert-Spahn, J., Feller, E. and Shoettner, M.: Incremental Checkpointing for Grids, *Linux Symposium* (2009).
32) Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I. and Warfield, A.: Live migration of virtual machines, *NSDI'05: Proc. 2nd conference on Symposium on Networked Systems Design & Implementation*, Berkeley, CA, USA, pp.273–286, USENIX Association (2005).
33) Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A.: Kvm: The Linux virtual machine monitor, *Ottawa Linux Symposium*, pp.225–230 (2007).
34) Nelson, M., Lim, B.H. and Hutchins, G.: Fast transparent migration for virtual


© 2010 Information Processing Society of Japan

machines, *ATEC '05: Proc. annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, p.25, USENIX Association (2005).
35) Glinka, F., Ploss, A., Gorlatch, S. and Müller-Iden, J.: High-level development of multiserver online games, *Int. J. Comput. Games Technol.*, Vol.2008, pp.1–16 (2008).

(Received July 24, 2009)
(Accepted December 7, 2009)

**Balazs Gerofi** is a Ph.D. candidate of the Computer Science Department at the University of Tokyo as of April 1, 2009. He received his M.Sc. degree of Computer Science (cum laude) from the Vrije Universiteit Amsterdam in October, 2006. His research focuses on operating systems, distributed computing, and dependable systems.



**Hajime Fujita** is a Project Research Associate at Information Technology Center of the University of Tokyo. He received his Master of Computer Science degree in 2008 from the University of Tokyo. His research interests include systems software for computer clusters and dependable systems.

**Yutaka Ishikawa** is a professor of the University of Tokyo, Japan. Ishikawa received the B.S., M.S., and Ph.D. degrees in electrical engineering from Keio University. From 1987 to 2001, he was a member of AIST (former Electrotechnical Laboratory), METI. From 1993 to 2001, he was the chief of Parallel and Distributed System Software Laboratory at Real World Computing Partnership. His interests include the next generation supercomputer, cluster technologies, dependable parallel and distributed systems.



© 2010 Information Processing Society of Japan