2010 IEEE International Conference on Cluster Computing

# An Efficient Process Live Migration Mechanism for Load Balanced Distributed Virtual Environments

Balazs Gerofi*, Hajime Fujita† and Yutaka Ishikawa*†
* *Graduate School of Information Science and Technology*
The University of Tokyo
Tokyo, JAPAN
† *Information Technology Center*
The University of Tokyo
Tokyo, JAPAN
{bgerofi@il.is.s, hfujita@cc, ishikawa@is.s}.u-tokyo.ac.jp

***Abstract*—Distributed virtual environments (DVE), such as multi-player online games and distributed simulations may involve a massive amount of concurrent clients. Deploying distributed server architectures is currently the most prevalent way of providing such large-scale services, where typically the virtual space is divided into several distinct regions requiring each server to handle only part of the virtual world. Inequalities in client distribution may, however, cause certain servers to become overloaded, which potentially degrades the interactivity of the environment and thus renders the load balancing problem a crucial issue. Prior research has shown several approaches for avoiding uneven workload, nevertheless, addressing the problem mainly at the application layer.**

**In this paper we focus on solving the DVE load balancing problem at the operating system level. We propose an efficient process live migration mechanism, which is optimized for processes maintaining a massive amount of network connections. Building on top of it, we have implemented a decentralized middleware that instruments process migration among the cluster nodes, attempting to equalize loads on all machines. We demonstrate the performance of the live migration mechanism on a real-world multiplayer game server and show the behavior of the load balancing engine through a realistic DVE simulation.**

## I. Introduction

In distributed virtual environments (DVE), for example, massively multi-player online games (MMPOG), networked virtual environments (NVE) [1] and distributed simulations such as the High-Level Architecture (HLA) [2] thousands, or even hundreds of thousands of concurrent entities, i.e. clients, may interact with each other. To support such large-scale virtual environments, typically, a distributed server infrastructure is used, where the virtual space is partitioned into several distinct zones requiring each server to manage only a portion of the virtual world [3].

However, neither the interaction nor the movement of clients in the virtual space is predictable, which from time to time leads to high concentration of clients in certain zones. This causes imbalanced load distribution among the servers, adversely affecting the response time and damaging the interactivity of the virtual environment.

Previous studies have yielded several approaches to balance the workload among DVE servers, although addressing the problem always at the application layer [3], [4]. Load balancing algorithms implemented at this abstraction level are suffering from the following restrictions: Client migrations are heavy, because client state has to be subtracted and transferred between the zones and clients have to reconnect to the new server; the load of a particular server maintaining a certain zone can be directly migrated only to a server handling a neighboring zone in the virtual space, imposing severe constraints on the physical machines that may participate in the load balancing procedure at the given period [5].

In this paper we investigate solving the load balancing problem of a cluster distributed DVE server at the operating system level. Our architecture is based on a single IP address cluster, where the router broadcasts incoming packets to every node [6], [7]. We are aiming at operating zone servers as the migratable units of the system, thus requiring two important conditions to be satisfied. On one hand, zone servers may hold a massive amount of network connections that have to be sustained transparently on the destination node, on the other hand, they exhibit highly interactive behavior that must not degrade during the migration.

We propose a process live migration mechanism that allows continued execution of applications during most of the migration procedure and is optimized for processes that maintain a massive amount of network connections. Our mechanism tracks socket changes incrementally and transfers them in an aggregated fashion, which we call *incremental collective socket migration*. Both remote (client) and local (in-cluster) connections are migratable, with the transition being fully transparent from the peers' point of view. Our technique offers *short process freeze time*, the time while the application stays unresponsive during the migration, therefore making it feasible for highly interactive applications.

Due to their diverse utilization in DVE communications, we support migrating both UDP and TCP sockets, where TCP sockets can stay either in established or in listening states. Prior research has reported that in-cluster socket migration may cause incoming packet loss during the migration [8]. We address the problem of *preventing incoming packet loss* by exploiting the broadcast property of the network configuration.

978-0-7695-4220-1/10 $26.00 © 2010 IEEE
DOI 10.1109/CLUSTER.2010.25

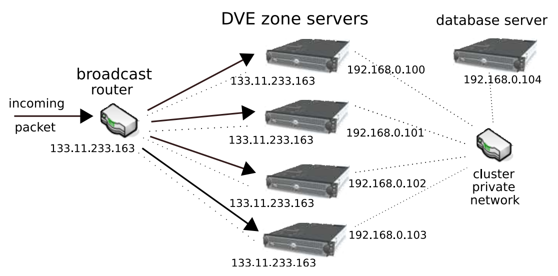


Fig. 1: DVE cluster server architecture.

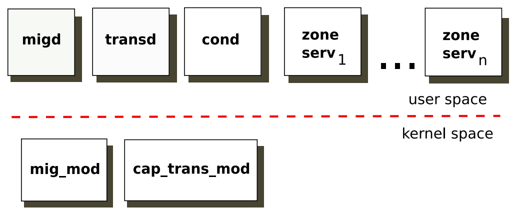


Fig. 2: Software components of a server node.

Building upon process live migration, we have developed a completely decentralized middleware layer, which monitors the load of each node in the cluster and instruments process migrations among the computers attempting to equalize the loads on all machines.

We present performance measurements of the live migration mechanism on *OpenArena*, an open source first-person shooter (FPS) multi-player online game (which is an extension to the Quake III game engine) [9]. Moreover, we demonstrate experimental results on dynamically load balancing a realistic DVE simulation, in which zone server processes maintain a massive amount of client, as well as, local connections such as MySQL database sessions. Characteristics of the simulation, e.g., network communication properties, are based on attributes of real-world MMPOGs.

The kernel level components of our prototype implementation are entirely comprised of kernel modules for Linux 2.6, without any modifications to existing kernel code.

This paper makes the following contributions: an efficient process live migration mechanism is designed and optimized for processes maintaining a massive amount of network connections; connections can be both client and in-cluster based on TCP or UDP sockets; incoming packet loss during the migration is prevented by exploiting the broadcast configuration of the router; and the migration mechanism is integrated into a load balancing middleware providing operating system level support for load balanced distributed virtual environments.

We begin with an architectural overview of the system in Section II. Process live migration, incoming packet loss prevention and socket migration are described in Section III. The dynamic load balancing middleware is explained in Section IV, implementation details are provided in Section V and evaluation is given in Section VI. Section VII surveys related work and Section VIII concludes the paper.

## II. System Architecture and Assumptions

### A. Cluster Configuration

Figure 1 summarizes the single IP address server architecture, which consists of DVE and database servers. Each DVE server node in the cluster is equipped with a public and a local network interface. The same IP address is assigned to the public interfaces and the local ones are used for in-cluster communication. The router simply broadcasts each incoming packet to all DVE server nodes [10]. Database servers may store persistent state information, which is in turn accessed by the zone server processes.

Unlike network address translation (NAT) based single IP address clusters [11], where each time a connection is migrated inside the cluster the router's IP to MAC address mapping needs to be updated [8], the benefit of the this configuration is the combination of the single public IP address and the broadcast property that allows migrating connections inside the cluster without any extra effort on the router. For further details on socket migration refer to Section III-C.

An issue immediately raised by the singe public IP configuration is how new connections are accepted. In our proposed architecture DVE processes are identified by separate port numbers, instead of separate IP addresses, ensuring that a given port number is taken care by exactly one server node at any time. Furthermore, in this paper we assume that files used by DVE server processes are available on all server nodes, either by having them replicated or if files are shared, through a distributed file system.

### B. Software Components

Figure 2 depicts the software components present on each DVE server node. The components and their role in the system are as follows:

*mig_mod:* the process migration kernel module is an extension of the BLCR [12] open-source checkpoint-restart implementation, incorporating our changes for supporting process live migration and TCP/UDP socket migration;

*cap_trans_mod:* a kernel module handling network packet capturing and address translation, that are part of the incoming packet loss prevention and in-cluster socket migration mechanisms, which will be explained in detail in Section III-B and Section III-C, respectively.

*transd:* the translation daemon is a user-level daemon process, that receives address translation requests and consults the kernel for installing the appropriate filters. It is present on all nodes inside the cluster that may be involved in a local socket migration. The exact mechanism will be described in Section III;

*cond:* the conductor daemon is another user-level daemon process that monitors resource consumption of zone server processes, communicates to other nodes' conductors and instruments process migration for balancing load among the server nodes. It is responsible for decisions such as initiating or accepting a process migration request.

*migd:* the process migration daemon is responsible for actually carrying out migration requests, i.e. consulting the kernel for executing a process checkpoint/restart operation through the network. It works in tight co-operation with the conductor process.

*zone_serv$_i$:* each DVE zone server manages a partition of the virtual space. Maintains multiple client connections and may have connections with neighboring zone servers in the virtual space or connections with other local services such as database servers.

## III. Process and Socket Migration

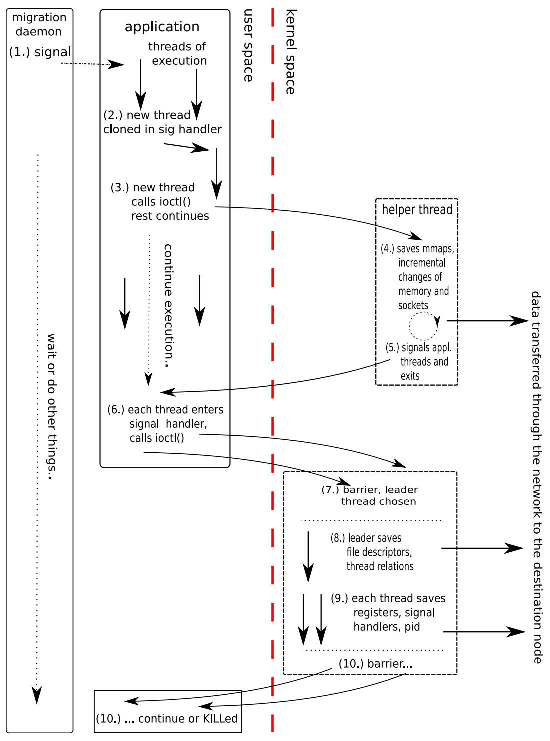
### A. Process Live Migration

Process live migration is the act of transferring a process from a source node to a destination node while allowing the program execution to proceed during most of the migration procedure. One of the possible approaches is the so-called precopy strategy [13], where the overall migration mechanism is divided into two phases. The precopy phase 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. Finally, in the freeze phase, the actual execution context is moved to the destination node, during which only, the process stays unresponsive.

We have extended the Berkeley Lab Checkpoint/restart library [12] for supporting process live migration. Our mechanism is based on incremental checkpointing, i.e. 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 incremental checkpointing.

First, the application receives the signal of a live checkpointing request. It clones a new thread and all the application threads simply return from the signal handler (i.e. continue their execution). The new thread enters the kernel via an ioctl() call, transfers memory mappings to the destination node and enters a loop of tracking address space changes and incrementally checkpointing sockets. In each subsequent iteration the loop timeout is decreased. When it reaches a threshold of the timeout (which is currently 20 milliseconds) it signals the application threads for final checkpointing. Executing the signal handler, each application thread enters the kernel, they synchronize on a barrier and a leader thread is chosen based on setting an atomic variable. The leader transfers open file table (note, that the contents of the files are not transferred), file descriptors, where a final incremental step of socket checkpointing is also performed, and thread relations. Each thread then transfers registers, signal handlers and its process/thread ID. They all enter a final barrier before returning to userspace where they finish up the signal handler and continue or finish their execution, according to the option specified.

As for the destination node, the restarting steps performed are the same with the regular BLCR mechanism [12], except that the leader thread of the restarting process receives and applies incremental updates on the process address space, before the actual execution context gets migrated. Further implementation specific aspects of the precopy strategy are discussed in Section V-A.


migration daemon
(1.) signal
wait or do other things..
application
threads of execution
(2.) new thread cloned in sig handler
(3.) new thread calls ioctl() rest continues
continue execution..
(6.) each thread enters signal handler, calls ioctl()
user space
kernel space
helper thread
(4.) saves mmaps, incremental changes of memory and sockets
(5.) signals appl. threads and exits
(7.) barrier, leader thread chosen
(8.) leader saves file descriptors, thread relations
(9.) each thread saves registers, signal handlers, pid
(10.) barrier...
(10.) ... continue or KILLed
data transferred through the network to the destination node
Fig. 3: Process live migration mechanism.

Checkpointing can also be initiated directly from the kernel, without notifying the application [14]. However, the signal based approach offers a convenient property, such as that even if a thread executes a system call and therefore may lock important kernel structures, it will abandon the call and return to userspace first. For further details on how socket migration benefits from this, refer to Section V-C.

### B. Prevention of Incoming Packet Loss

As we shall detail in Section III-C, restoring sockets on the destination node is performed in the freeze-phase of the process migration. Between disabling the socket on the source node and restoring it on the destination, there is clearly a certain period of time while the socket is not processing incoming packets. These packets may get lost, requiring the transport layer protocol to retransmit the data, thus adversely affecting the interactivity of the application [8].

A mechanism for preventing incoming packet loss is realized by capturing network packets, which match the migrating socket, on the destination node while the connection is unresponsive. In case of a TCP connection, before disabling the socket on the source machine, remote IP, remote port and local port number are transferred to the target node.

After the TCP socket is successfully migrated, each packet on the queue gets processed by the socket. For implementation specific issues refer to Section V-B.

### C. *Incremental Collective Socket Migration*

In this section we propose a socket migration mechanism, which is optimized for processes that maintain a massive amount of network connections.

The application's file descriptor table is iterated during the freeze-phase of the process migration procedure, where for each open file the appropriate migration steps are performed. BLCR supports regular files, which are re-opened on the destination node, but sockets are simply omitted in the original implementation.

We have extended BLCR for supporting socket migration. Both UDP and TCP sockets are migratable, TCP sockets possibly residing in established or in listening states. Socket migration essentially, both in case of UDP and TCP sockets, means disabling the socket (i.e. unhashing from lookup tables and clearing its timers), subtracting state information, dumping not yet processed incoming and outgoing packet queues, transferring all data through the network and restoring it on the destination node. Technical details of the socket migration implementation are discussed in Section V-C.

Initially, we followed the natural way of iterating the file descriptor table and migrating each socket one-by-one [15]. However, there are two reasons why this approach imposes excessive performance overhead. On one hand, the network bandwidth is not fully utilized because short periods of computation (subtracting socket data) and network transmissions are repeated numerous times. On the other hand, as we have described earlier, socket migration is preceded by incoming packet loss prevention on the destination node which introduces further synchronizations between the source and the destination machines.

As an improvement for this problem we have scattered the file descriptor table iteration into three phases. In the first phase, capturing details of all TCP connections are collected and transferred to the destination node. When the capturing is successfully enabled for all sockets, the second phase subtracts state information and buffer queues of all connections into one unified buffer that is transferred in one go to the destination machine. Finally, BLCR's regular file descriptor table iteration is executed, but excluding the already processed network connections.

Later, we noticed that subtracting socket changes incrementally could also be performed during the precopy phase of the migration, which significantly decreases the number of bytes necessary to be transferred, because most of the socket structures do not change when the loop timeout becomes short enough during precopy. We maintain tracking structures for connections and transfer only the changes in each subsequent loop, including the final process freeze phase. In Section VI we demonstrate the efficiency of the various improvements.

*In-cluster Connection Migration:* Local (in-cluster) connections demand special attention, because the local IP address of the migrating socket changes after the migration. This issue is overcome by enabling a translation filter on the peer's host and is demonstrated through the following example scenario.

Let us consider the situation, where process $P$ on host $\text{IP}_1$ is migrated to the node $\text{IP}_2$, but it maintains a TCP connection with a process running on host $\text{IP}_3$. Steps are performed as follows. First, $\text{IP}_1$ contacts $\text{IP}_2$ to request packet capturing from $\text{IP}_3$. Then, $\text{IP}_3$ enables a translation filter (an in-kernel mechanism on a lower level than the socket processing, which is described in detail in Section V-D) for rewriting the target IP address of packets that are transmitted to $\text{IP}_1$ (the original address of the connection) so that they will be sent to $\text{IP}_2$ instead. For incoming packets, the filter exhibits the opposite behavior, i.e. packets that arrive from $\text{IP}_2$ will be modified and the source IP address is rewritten to $\text{IP}_1$. The socket is then migrated to $\text{IP}_2$, the captured packets are reinjected and it continues communicating to $\text{IP}_3$, without $\text{IP}_3$ noticing the transition.

## IV. Dynamic Load-Balancing

We developed a decentralized middleware, that monitors resource consumption of the server nodes and instruments process migration in order to load balance the cluster. Machines may join and leave at any time.

When a DVE node is initialized, the conductor daemon process scans the local (in-cluster) network looking for other computers that play the same role in the system. At the same time it answers discovery messages from other hosts that may be also searching for DVE server nodes.

Important roles of the conductor daemon are monitoring the node's condition and updating the information received from other machines. For the monitoring purpose the conductor retrieves load information via the *atop* utility [16]. Each node also keeps track of the load status of other nodes based on the latest information they sent, practically maintaining an approximation on the overall load of the whole cluster.

Our load balancing algorithm is sender initiated, meaning that overloaded nodes initiate process migrations to lightly loaded machines. Typically, a dynamic load balancing algorithm can be further specified by four important properties: a transfer policy, a location policy, a selection policy and an information policy [17]. In the next subsections we describe our load balancing mechanism in the context of these terms.

### A. *Transfer Policy*

The transfer policy determines whether a node is in a suitable state to participate in a migration.

On the sender side (i.e. the node from which the process is migrated) our transfer policy is threshold driven, which means that a node enters a migration initiator state when the local load is over a critical threshold or when the difference between the node's load and the approximated overall cluster load exceeds a certain value.

The receiver side (i.e. the node to where a process is migrated) enters the migrating state based on a two-phase commit protocol with the sender, allowing to receive only one migration at a given time. Note, that the actual receiver is chosen based on the location policy and the transfer policy only determines whether or not the chosen receiver is participating in the migration. After the migration is successfully finished, both nodes enter a calm-down period for stabilizing the indicators of their resource consumption.

### *B. Location Policy*

The location policy's responsibility is to find a suitable node where a process can be migrated.

Our location policy takes two attributes into account, the current condition of the local node and the approximation of the overall cluster load. It attempts to find a node in its local peer database that has a load index which is on the opposite side of the cluster average. The key objective of this step is to find a node that is nearly as much lighter as the local node is heavier compared to the overall average, which results in both nodes' load converging to the cluster wide average after the migration. Once a feasible node is found, the migration is negotiated with the destination.

### *C. Selection Policy*

The selection policy decides which process has to be migrated when a node is overloaded.

Our selection policy is also based on two attributes. It tries to find a process that consumes approximately as much CPU time as the difference between the local node and the approximated average of the cluster. Once again, this approach is driven by the idea of bringing both the sender and the receiver nodes' load closer to the approximated cluster average.

### *D. Information Policy*

The information policy decides when information about other nodes in the system is collected.

We follow a periodic policy, having each node broadcasting its load information periodically to all nodes in the cluster, which also serves as a heartbeat message denoting the node's presence in the system. Note that mechanisms for scalable broadcasting, such as utilizing spanning-trees, have been proposed [18], and are out of the scope of this paper.

## V. IMPLEMENTATION

We next provide implementation details on process live migration, incoming packet loss prevention, socket migration and local address translation.

### *A. Process Live Migration*

Process live migration following the precopy strategy 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 [19].

We opted for the approach of using the dirty bit and relaxing the swap facility, which is reportedly not a major restriction in environments with highly interactive applications, because swapping is not used anyway [19]. Utilizing directly the dirty bit allows us having the dirty page tracking mechanism entirely implemented in a kernel module, without any changes to existing kernel code.

However, besides tracking dirty pages that are already part of the process address space, another memory management issue that arises is the changes in the address space itself. Namely, insertions (i.e. memory allocations), modifications or removals (i.e. freeing) of continuous memory areas that are mapped in to the process address space. The Linux memory management system maintains mapped in memory areas as a linked list of *vm_area_struct* structures.

In order to track and reflect memory area changes we maintain a linked list of our own tracking structures that store the memory area properties of the last incremental loop. In each subsequent loop this list and the actual *vm_area_struct* list are compared and the tracking list is updated accordingly.

### *B. Prevention of Incoming Packet Loss*

As we described earlier, before a TCP socket is migrated the destination node enables a packet capturing feature in order to prevent losing packets that might arrive while the socket is unresponsive.

The actual capturing on the destination node is performed by a Linux Netfilter hook [20]. Netfilter 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 that are delivered to the localhost appear.

The hook function collects packets that match the corresponding socket's remote IP, remote port and local port numbers. It also checks TCP sequence numbers and stores duplicated packets only once.

After the TCP socket is successfully migrated, the reinjection phase iterates the capture queue and submits each packet to the network stack by calling the netfilter's *okfn()* (which in case of IPv4 is the *ip_rcv_finish()* function).

### *C. Socket Migration*

*1) TCP:* The initial step of migrating a TCP socket is unhashing it from both the *ehash* and *bhash* kernel hashtables and clearing the retransmission timer of the write queue.

Besides the main socket data structures, an important issue is in-flight packets. 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 queues which have to be taken into account. The backlog queue, that holds packets while a TCP socket is locked and the prequeue, that enables the Linux fast-path receiving mechanism [21].

The benefit of signal based checkpointing notification is the fact, that the process neither locks the socket nor performs fast-path receiving, due to its return to userspace. This ensures that both the backlog and the prequeue are empty during the process freeze phase and therefore copying the write queue, the receive queue and the out-of-order queue is sufficient. On the other hand, the socket tracking mechanism during the precopy phase simply omits sockets that are locked or being used for fast-path receiving, leaving the checkpoint either for the subsequent loop or the final process freeze phase.

Each time a connection is checkpointed and transferred to the destination, the target node simply stores the data. During the freeze phase it only allocates a new socket structure and applies the changes to the relevant fields of the socket representation based on the latest update. It allocates receive, write and out-of-order queues, updates packets and re-inserts them. Finally, the socket is rehashed for both the *ehash* and *bhash* hashtables, the retransmission timer is restarted and the socket is attached to the right file descriptor of the process.

Adjustment of TCP 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 in 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.

*2) UDP:* Migrating UDP sockets is considerably easier than TCP. Besides the main UDP socket data structure, we only track and transfer the socket buffers residing on the receive queue.

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.

### D. Local Address Translation

Local address translation, which is enabled on the (in-cluster) peer's endpoint of a migrated connection also utilizes the Linux Netfilter facility [20]. There are two hook functions registered for this purpose, one resides on *NF_INET_LOCAL_IN*, for translating incoming packets, the other is attached to *NF_INET_LOCAL_OUT*. Incoming packets' source address and outgoing packets' destination address are rewritten, respectively.

We encountered two technical issues that are worth mentioning. There is a Linux IP destination cache entry assigned to each outgoing packet, that is inherited from the socket the packet originates from [21]. Since we do not modify the socket details on the peer's host, this structure holds the old destination IP address, resulting in the packet's transmission to the old destination. This has been overcome by creating an accurate destination cache entry and besides rewriting the IP header's destination address, the cache entry is also replaced.

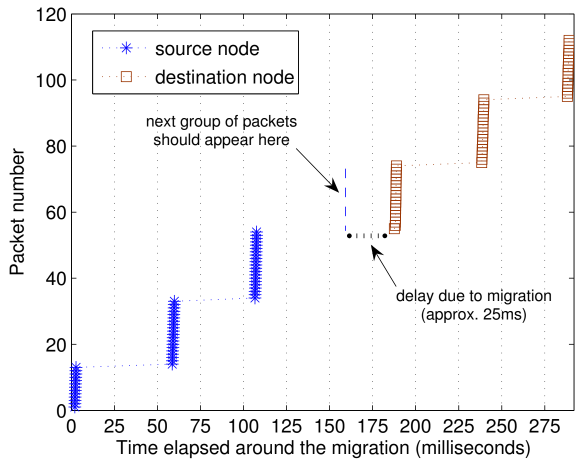


Fig. 4: Packet delay due to migration (OpenArena server, 24 clients).

Another issue was the TCP checksum, that needs to be updated in order to reflect the modified IP header.

## VI. Evaluation

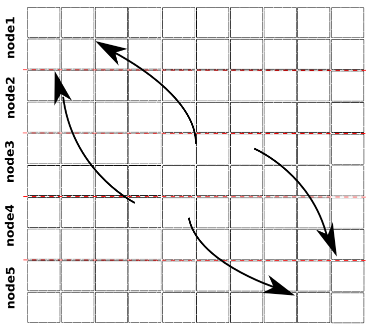
### A. Experimental Framework

Experiments were conducted on a dedicated single IP address cluster with five DVE server nodes and a MySQL database server. Each node is equipped with a 2.4GHz Dual-Core AMD Opteron processor and two gigabytes of RAM. All machines are interconnected by a Gigabit Ethernet network for in-cluster communication and each DVE server has a Gigabit Ethernet public interface to which the router broadcasts incoming packets.

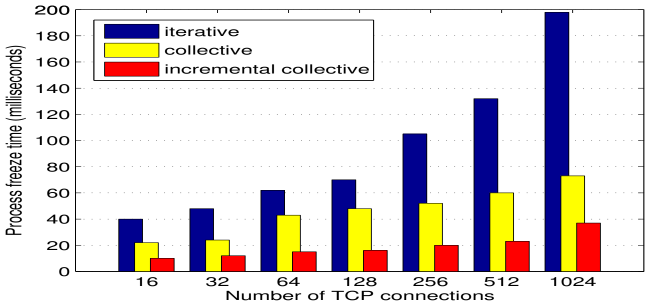
### B. OpenArena

OpenArena is an open source first-person shooter (FPS) multi-player online game based on the Quake III engine [9]. We evaluate our migration mechanism through live migrating an OpenArena server with 24 clients being involved. OpenArena uses UDP protocol for server-client communication and the default update frequency is 20 messages per second.

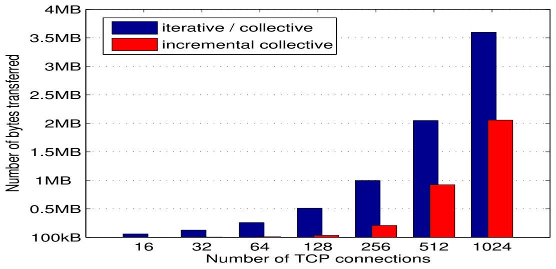
We experience a 20 milliseconds downtime of the server process execution during the migration and assess the imposed delay at the network packet level by capturing server packets with *tcpdump*. Figure 4 shows the time difference between the last packet of the source and the first packet of the destination nodes. One of the key observations is the characteristic of the regular execution, which updates the clients in every 50 milliseconds (i.e., 20 updates per sec). Focusing on the migration, the imposed delay compared to the expected packet



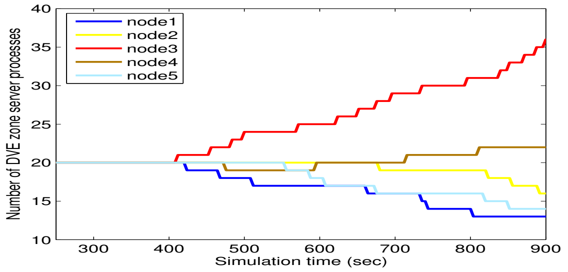
(a) Initial virtual space partitioning and the main directions of the clients' movement during the simulation.



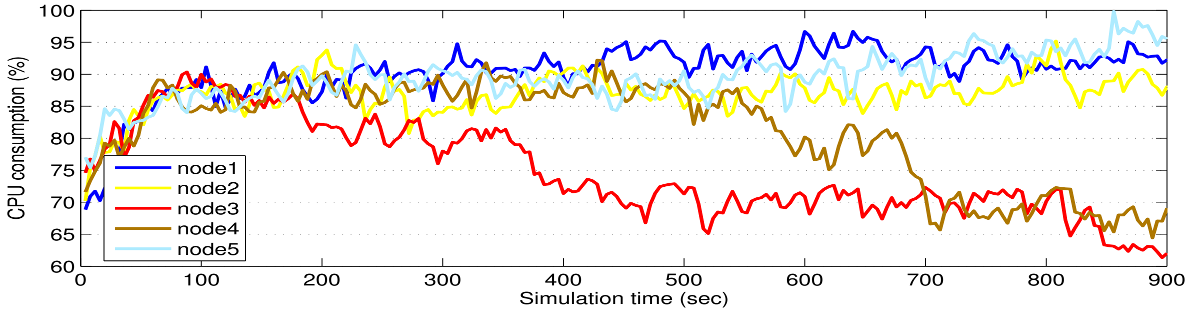
(b) Worst case process freeze time with iterative, collective, and incremental collective socket migration according to the number of network connections.



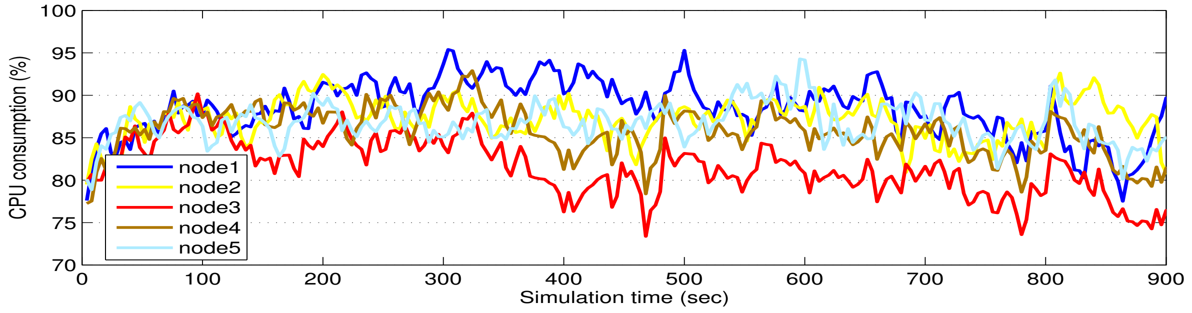
(c) Socket data transferred during freeze phase of migration.



(d) Zone server process distribution among nodes with load-balancing enabled.



(e) CPU consumption per node without load-balancing.



(f) CPU consumption per node with load-balancing enabled.

Fig. 5: Process freeze time, socket data transferred, zone server processes and load distribution among DVE server nodes.

transmission is approximately 25 milliseconds, which remains completely transparent from the clients' point of view.

### C. DVE Simulation Environment

We have implemented a DVE simulation with characteristics resembling real-world multi-player online games. The server process updates clients approximately 20 times per second (which is the default setting of the Quake III engine) with a message size of 256 bytes, an average value for MMPOGs as reported [22], [23].

The virtual space is partitioned into one hundred zones following a ten times ten grid shape. Each DVE server node is initially assigned to 20 zones, i.e. there are 20 zone server processes running on each of them. Figure 5a depicts the zone arrangement and the initial assignment to server nodes. Zone servers (processes) all maintain a MySQL session with the local database server, where properties of the virtual world are repeatedly updated. They perform the so-called *real-time loop* [24], which continuously processes events from clients, governs interactions among them and responds state updates. CPU consumption of a zone server process grows proportionally with the number of clients present in the given zone. We currently do not maintain direct connections among the zone servers. With careful synchronization among the hosts involved, local socket migration could be performed for such connections as well, which we intend to address as part of our future work.

There are 10,000 clients participating in our DVE simulation initially following a uniform distribution among the zones. The overall experiment takes approximately 15 minutes, in which clients from the middle regions of the virtual space are instructed to gradually move into the directions of the up-left and down-right corners. Figure 5a also shows the direction of the high-level movement of clients. Reportedly, this sort of clustering of entities in large-scale environments is very common [24]. We log and analyze the load distribution of the DVE servers, both with and without load balancing.

### D. DVE Experimental Results

Measurements were conducted to assess live migration process freeze time of zone servers used in our DVE simulation. We show how collective and incremental socket migration (described in Section III-C) improves live migration performance. We migrate zone server processes with different number of client connections, where the number of clients varies from 16 up to 1024. Each server also maintains a local MySQL session.

Figure 5b shows experimental results on live migration worst case process freeze time in case of iterative, collective and incremental collective socket migration. Furthermore, Figure 5c depicts the worst case (largest) amount of bytes transferred during the process freeze phase according to the number of maintained connections.

One of the key observations is that process freeze time in case of iterative socket migration grows proportionally with the number of bytes transferred, however this is not case when collective and incremental socket migration is enabled. Migrating sockets in an aggregated fashion helps better utilizing the network bandwidth, while incrementally tracking changes significantly decreases the number of bytes representing the connections. As it is shown migrating over 1000 connections results in less than 40 milliseconds downtime, which we believe is short enough even for a highly interactive application.

Resource consumption during the simulation has been logged to assess the efficiency of the load balancing middleware. Figure 5e depicts the load distribution of nodes during the simulation with load balancing disabled. It shows that $\text{node}_1$ and $\text{node}_5$, the ones responsible for the upper and lower regions of the virtual space suffer significant load concentration as the simulation progresses, eventually reaching a phase of constantly consuming over 95% of their CPUs. $\text{Node}_3$ and $\text{node}_4$, on the other hand, gradually become less and less loaded, eventually falling below 65% of CPU utilization.

As Figure 5f demonstrates, with load balancing enabled the system automatically reassigns (i.e. live migrates) zone server from the nodes responsible for the upper and lower regions to servers originally maintaining the middle regions of the virtual space resulting in a much lighter imbalance in their resource consumption.

An interesting aspect, changes in the number of zones each particular node maintains, is shown on Figure 5d. As it depicts part of the server processes ran on $\text{node}_1$ and $\text{node}_5$ were relocated, i.e. the number of processes decreased, to nodes such as $\text{node}_3$ and $\text{node}_4$, where in turn the number of processes increased.

## VII. Related Work

### A. Connection Migration

Connection migration has been subject of many prior studies. NEC corp. proposed transferring TCP sessions between nodes for a distributed Web Server architecture, assigning each TCP session to a virtual IP address which is reported to cause loss of incoming packets [8].

SockMi [25] offers TCP migration with IP layer forwarding between the source and the target node, therefore it is not feasible for decoupling a process entirely from its source machine. Furthermore, it requires application specific support for explicitly exporting and importing connections. Tcpcp [26] 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 [27] and MSOCKS [28].

TCP Migrate option [29] 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 both ends of the connection need the extension to the protocol, which forms a strict limitation on the supported client machines.

Reliable sockets (ROCKS) and reliable packets (RACKS) [30] 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.

### B. Process Migration

Process migration has been a hot topic in system research and several distributed operating systems offer the capability of migrating processes. V-System [13], Amoeba [31], Mach [32], Sprite [33], and MOSIX [34] 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 [35] 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 either connection migration or incrementally dumping address space changes.

Zap [36] 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 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 [14], [19]. 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.

### C. Virtual Machine Migration

Virtual machine (VM) migration is an actively researched topic in recent years. Solutions based on Xen [37], KVM [38] and VMware's VMotion [39] provide also with 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 comparing to migration on the process level [40]. 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.

It has also been shown that VM live migration keeping network connections alive gives comparable service downtime to process level live migration [37]. However, no results are disclosed for the case where massive amount of connections are involved.

On the other hand, taking zone server as the migratable unit of the system, the disadvantage of VM based solutions lies in the fact that each server would have to reside in its own VM allocating extra resources.

### D. Load balancing Distributed Virtual Environments

Prior studies have yielded several approaches to load balancing server architectures in Distributed Virtual Environments [3], [4], [5], [24]. These techniques, however, are all concerned with application layer solutions and none of them deals with support on operating system level.

To the best of our knowledge, only MOSIX [34] has been considered so far for managing cluster resources at the OS level in the domain of game server hosting [41]. However, involvement of the MOSIX *home-node* in network communication even after process migration and the lack of support for live migrating multithreaded applications makes MOSIX' application cumbersome for interactive applications such as DVE servers.

## VIII. Conclusion and Future Work

In this paper, a novel operating system level approach to automatically load balancing distributed virtual environments has been proposed.

We have presented a process live migration mechanism that is optimized for applications maintaining a massive amount of network connections. No modificatios are required either to the TCP protocol or to the client side network stack. Incremental collective socket migration, a technique that tracks socket changes and transfers them in an aggregated fashion, offers acceptable process freeze time even for highly interactive applications. A mechanism for preventing incoming packet loss during the migration has also been proposed. Experimental results on OpenArena, an FPS multiplayer online game server showed that the transition remains completely transparent from the clients' point of view. Moreover, we demonstrated through processes that resemble realistic DVE communication properties, that migrating over 1000 TCP connections can be performed with keeping the process freeze time less than 40ms.

Exploiting process live migration, we have developed a decentralized middleware that instruments process migration among the cluster nodes at the operating system level. We have shown that it succeeds in equalizing imbalances in the load of a set of machines participating in a DVE simulation, leaving the load balancing problem entirely transparent for application developers. Our prototype implementation is based on Linux 2.6 with kernel level components entirely implemented in kernel modules, making their deployment process easy.

Process live migration that keeps network connections alive could be utilized in several other scenarios, such as addressing fault tolerance or power management. In the future we intend to investigate further use cases. Multimedia streaming, among others, is one of our main future perspectives.

## Acknowledgment

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

## References

[1] S. Singhal and M. Zyda, *Networked Virtual Environments: Design and Implementation*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999.

[2] "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules," *IEEE Std 1516-2000*, pp. i–22, Sep 2000.

[3] D. Lee, M. Lim, S. Han, and K. Lee, "ATLAS: A Scalable Network Framework for Distributed Virtual Environments," *Presence: Teleoper. Virtual Environ.*, vol. 16, no. 2, pp. 125–156, 2007.

[4] R. Chertov and S. Fahmy, "Optimistic load balancing in a distributed virtual environment," in *NOSSDAV '06: Proceedings of the 2006 international workshop on Network and operating systems support for digital audio and video*. New York, NY, USA: ACM, 2006, pp. 1–6.

[5] T. N. B. Duong and S. Zhou, "A dynamic load sharing algorithm for massively multiplayer online games," in *Networks, 2003. ICON2003. The 11th IEEE International Conference on*, Sept.-1 Oct. 2003, pp. 131–136.

[6] Microsoft, "Network Load Balancing Technical Overview," http://www.microsoft.com/technet/prodtechnol/windows2000serv/deploy/confeat/nlbovw.mspx.

[7] H. Fujita, H. Matsuba, and Y. Ishikawa, "TCP Connection Scheduler in Single IP Address Cluster." Washington, DC, USA: IEEE Computer Society, 2008, pp. 366–375.

[8] M. Takahashi, A. Kohiga, T. Sugawara, and A. Tanaka, "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, 0-0 2006.

[9] "OpenArena," http://openarena.ws/smfnews.php, 2010.

[10] O. P. Damani, P. E. Chung, Y. Huang, C. Kintala, and Y.-M. Wang, "ONE-IP: techniques for hosting a service on a cluster of machines," in *Selected papers from the sixth international conference on World Wide Web*. Essex, UK: Elsevier Science Publishers, Ltd., 1997, pp. 1019–1027.

[11] W. Zhang, "Linux Virtual Servers for Scalable Network Services," *Linux Symposium*, 2000.

[12] J. Duell, "The design and implementation of Berkeley Lab Linux Checkpoint/restart," Lawrence Berkeley National Laboratory, Technical Report, 2000.

[13] M. M. Theimer, K. A. Lantz, and D. R. Cheriton, "Preemptable remote execution facilities for the V-system," in *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 1985, pp. 2–12.

[14] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 9.

[15] B. Gerofi, H. Fujita, and Y. Ishikawa, "Live Migration of Processes Maintaining Multiple Network Connections," *IPSJ Transactions on Advanced Computing Systems (ACS 29)*, vol. 3, no. 1, 2010.

[16] "ATOP System & Process Monitor," http://www.atcomputing.nl/Tools/atop, 2010.

[17] N. G. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems," *Computer*, vol. 25, no. 12, pp. 33–44, 1992.

[18] S. Chatterjee and M. Bassiouni, "Scalable and efficient broadcasting algorithms for very large internetworks," vol. 3, jun 1996, pp. 1642 –1647 vol.3.

[19] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration in HPC environments," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[20] Linux Netfilter, "Firewall, NAT, and packet mangling for Linux," http://www.netfilter.org.

[21] Seth, Sameer and Venkatesulu, M. Ajaykumar, *TCP/IP Architecture, Design and Implementation in Linux*. Wiley-IEEE Computer Society Pr, 2008.

[22] J. Kim, J. Choi, D. Chang, T. Kwon, Y. Choi, and E. Yuk, "Traffic characteristics of a massively multi-player online role playing game," in *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*. New York, NY, USA: ACM, 2005, pp. 1–8.

[23] A. Bharambe, J. Pang, and S. Seshan, "Colyseus: a distributed architecture for online multiplayer games," in *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 12–12.

[24] F. Glinka, A. Ploss, S. Gorlatch, and J. Müller-Iden, "High-level development of multiserver online games," *Int. J. Comput. Games Technol.*, vol. 2008, pp. 1–16, 2008.

[25] M. Bernaschi, F. Casadei, and P. Tassotti, "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, Feb. 2007.

[26] W. Almesberger, "TCP Connection Passing," 2004, Ottawa Linux Symposium.

[27] P. Bhagwat, C. Perkins, and S. Tripathi, "Network layer mobility: an architecture and survey," *IEEE Personal Communication, vol. 3, no. 3*, pp. 54–64, 1996.

[28] D. A. Maltz and P. Bhagwat, "Msocks: An architecture for transport layer mobility," *Proceedings of the IEEE INFOCOM*, pp. 1037–1045, 1998.

[29] A. Snoeren and H. Balakrishnan, "An end-to-end approach to host mobility," *Proc. MobiCom '00*, pp. 155–166, 2000.

[30] V. C. Zandy and B. P. Miller, "Reliable network connections," *Proceedings of the 8th International Conference on Mobile Computing and Networking*, pp. 95–106, 2002.

[31] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba – a distributed operating system for the 1990s," *IEEE Computer, vol. 23, no. 5*, pp. 44–53, 1990.

[32] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for unix development," *Proceedings of the Summer 1986 USENIX*, pp. 93–112, 1996.

[33] F. Douglis and J. Ousterhout, "Transparent process migration: Design alternatives and the sprite implementation," *Software - Practice and Experience, vol. 21, no. 8*, pp. 757–785, 1991.

[34] A. Barak and R. Wheeler, "Mosix: An integrated multiprocessor unix," *USENIX Winter Technical Conference*, pp. 101–112, 1989.

[35] C. Steketee, "Process Migration and Load Balancing in Amoeba," 1999.

[36] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of Zap: A system for migrating computing environments," in *In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002, pp. 361–376.

[37] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286.

[38] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Ottawa Linux Symposium*, July 2007, pp. 225–230. [Online]. Available: http://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf

[39] M. Nelson, B. H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, p. 25. [Online]. Available: http://portal.acm.org/citation.cfm?id=1247360.1247385

[40] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Comput. Surv.*, vol. 32, no. 3, pp. 241–299, 2000.

[41] D. Martin, A. v. Moorsel, and G. Morgan, "Efficient Resource Management for Game Server Hosting," in *ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 593–596.