Portable library of migratable sockets

Marian Bubak[a,b,*], Dariusz Żbik[a,d],
Dick van Albada[c], Kamil Iskra[c] and Peter Sloot[c]
[a]*Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland*
[b]*Academic Computer Centre – CYFRONET, Nawojki 11, 30-950 Kraków, Poland*
[c]*Informatics Institute, Universiteit van Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*
[d]*School of Banking and Manageament in Cracow, Kijowska 14, 30-079 Kraków, Poland*

Efficient load balancing is essential for parallel distributed computing. Many parallel computing environments use `TCP` or `UDP` through the socket interface as a communication mechanism. This paper presents the design and development of a prototype implementation of a network interface that can preserve communication between processes during process migration. This new communication library is a substitution for the well-known socket interface. It is implemented in user – space; it is portable, and no modifications of user applications are required. `TCP/IP` is applied for internal communication, which guarantees relatively high performance and portability.

Keywords: Distributed computing, load balancing, process migration, `Dynamite`, sockets

1. Introduction

In order to efficiently use distributed computing power it is essential to enable process migration from one host to another. In this way, load can be moved from a heavily loaded node to another, less loaded one. It is also desirable to do this in a way that allows the original host to be serviced (i.e. rebooted) without a need to break computations. As a rule, any parallel computation requires some communication and synchronization, so advanced distributed computing environments should handle the communication between processes in spite of the migration. The most popular environments like PVM and MPI do not offer this kind of functionality. These communication libraries have to be rewritten to support task migration.

One of the systems developed to support dynamic load balancing is `Dynamite` [6,9], which attempts to maintain an optimal task mapping in a dynamically changing environment. `Dynamite` balances the load within the system by migrating individual tasks. `Dynamite` comprises monitoring, scheduling, and migration subsystems [9]. The problem is that migrating processes can not use pipes, shared memory, kernel supported threads, and sockets. Support for open files is limited to files that are available through the same pathname before and after the migration [6].

Many parallel computing environments use `TCP` or `UDP` through the socket interface as a communication mechanism. This paper presents the concept and first implementation of a library that, besides offering the same functionality as the system socket library for the `TCP/IP` protocol family, allows migration of the process without interrupting communication with other processes. The new library, called *msocket*, can be a substitution for the standard socket library so that no changes in the application program will be required. All necessary modifications are handled at the library level so that no changes in the kernel are needed, either.

The *msocket* library handles both `TCP` and `UDP` sockets. Finding a solution for the stream oriented `TCP` sockets is much more complicated and that is why this paper focuses mainly on `TCP`.

The *msocket* library was developed as an extension to the `Dynamite`.

2. Environments enabling open socket migration

One of the environments for migrating processes is Hijacking [10]. This system does not require any changes in the process code; changes are done dynamically after a process starts. The Hijacking system uses DynInst [3], an architecture independent API for modifying the running program. The *mutator* process attaches to the process (application) that is to be mi-

*Corresponding author: Marian Bubak, Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland. Tel.: +48 12 617 39 64; Fax: +48 12 633 80 54; E-mail: bubak@uci.agh.edu.pl.

ISSN 1058-9244 / $8.00 © 2001, IOS Press. All rights reserved

grated. The process is stopped, and then a child process, named *shadow*, is created. The *shadow* inherits all resources used by the parent. After the migration, processes use resources through the *shadow*. This solution is transparent but rather expensive because of the additional communication delays incurred for all system calls routed through the *shadow*. The principal disadvantage of the *shadow* is that it uses the host where the process was initiated.

Mosix [1] is software to support cluster computing and it is implemented on the operating system level. Each Unix version requires a different implementation of Mosix, and recently the seventh implementation of Mosix was developed for Linux using the `x86` based processors. The Mosix migration mechanism is called *Preemptive Process Migration*. Almost any process may be migrated at any time to any available host. Each running process has a *Unique Home-Node* (UHN), which is the node where the process was created. After migration the process uses resources of the new host if possible but interaction with the environment requires communication with the UHN. Many system calls require data exchange between the user space and the kernel. For each remote call it is required to copy data between the migrated process and its part left on the UHN; the `copy_to_user`() and `copy_from_user`() kernel primitives transfer data through the network, and this operation is time consuming.

3. Requirements for socket migration

An essential requirement for the socket migration is that all modifications of communication libraries have to be transparent for the `TCP` and `UDP` protocols. To easily migrate a process with the socket interface in use, modifications to the communication library have to ensure the following:

1. establishing new connections must be allowed regardless of the number of process migrations; communication is possible between two processes with modified communication library
2. all connections that had been established before the migration took place have to be kept, and data must not be lost.

Another important requirement is that the programmer should not need to know in advance that a particular application will migrate at runtime. All modifications should be done in the user-space, and no changes are allowed to the kernel source code, nor are additional kernel modules. The new library should work with both `UDP` and `TCP` protocols.

The requirement not to change the Unix kernel forces us to modify the system calls and library calls. In the user code, communication through the `TCP` and `UDP` protocols uses sockets that are treated by the process as file descriptors. For this reason wrappers are used for each call that has file descriptors as arguments. In some cases this is a simple change that only translates the file descriptor number. The ability to create a wrapper to a function and a system call is a feature of the `Dynamite` dynamic loader [6]. `Dynamite` also takes care of process checkpointing and restoring.

4. Idea of migratable sockets

To allow uninterrupted redirection of packets or stream flow to a migrating process, it is possible to use one of the following approaches:

– all communication between processes goes through a daemon which forwards packets to the current process location (like DPVM when using indirect routing mode [5,7]),
– after migration, the process leaves a piece of code on the *old* machine and this code takes care of forwarding data to the new process location. In this paper, the term *mirror* is used for a process that redirects network packets. This concept was proposed in [1,10].
– the migrating process flushes all connections with other processes before migration and re-establishes them afterwards. This requires the immediate cooperation of all other processes involved. This solution is used in Hector [8] and for direct connections in `Dynamite`. It involves the use of special signal handlers and of daemons for signalling.

None of the solutions is perfect. The global (centralized) data distribution is not fault tolerant and could be too slow. In the second case, after process migration the machine is still in use, but in a different way. It means that the machine must not go down. This solution is not fault tolerant, either. The third solution requires special signal handling routines in all participating processes to ensure a timely response.

The *msocket* library makes use of all three concepts in a way which eliminates drawbacks. Our solution forwards some data but only after the migration. The *msocket* also uses the *mirror* but the life-time of the *mirror* is limited.

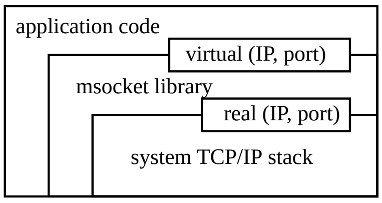


Fig. 1. Layers of the library.

After process migration, all connections have to be redirected to the new process location. The migrating process has to leave a mirror because some data could be on-the-fly (inside network stack buffers or inside active network equipment like a router or switch). The mirror should be removed as soon as possible. In this case, the mirror captures and redirects the packets which were on-the-fly during the process migration.

During normal work, processes use direct connections. The daemons are not used for passing user data between hosts. Daemons should exist on all machines, and their role is limited to redirecting connections only. When a process migrates, it has to inform peers about its new location. While restoring the process, daemons on all hosts which were involved in communication with the migrating process should be informed about its new location. Subsequently, the daemons force peer processes to redirect connections.

In the approach discussed in this paper, the data sent between processes always goes through an additional layer built of wrappers of the system calls. Inside the wrappers, some control information is changed but the communication is performed by the original `TCP/IP` stack (see Fig. 1).

This solution necessitates the use of address servers that allow to find the location of a socket. The communication with address servers is required only while establishing the connection; once the connection has been established, all data goes directly between processes. Our concept requires the use of daemons which just help to redirect the connection. As the `TCP` protocol is reliable, it should deliver all data to the process without losing or changing a single byte. To maintain the connection while the process is migrating, a mirror is kept; it receives data from peer processes and redirects it to the new process location.

5. Architecture of the *msocket* system

5.1. Mirror

The main aim of the mirror is to capture and redirect packets which were on-the-fly during the process migration. A mirror is started when the checkpointing of the process takes place. The mirror is a child of the process, so it inherits all sockets used by the process prior to migration. While the process is migrating *msocket* takes control – it happens after having accepted the checkpoint signal and before the checkpoint is saved. At that time *msocket* calls `fork`() and starts *mirror* as a procedure for child process (we don't use `exec`() because it is an unnecessary complication). The mirror works in a loop, it reads all data from the inherited sockets and sends these data to the new process location. After the migration, the process connects to the mirror and informs all the connection peers about its new location.

5.2. Virtual address

In the standard `IP` addressing scheme, the address of a socket is associated with the machine where the process (the owner of the socket) is running. The *msocket* library cannot work in this way because the real address of the host changes with each migration. To become independent of the changes of real addresses, virtual addresses are used. The form of these addresses is the same as that of the addresses used for the `TCP` and `UDP` communication, and these addresses may migrate with a process. In the *msocket* library, the address server (`msmaster`) is a centralized part of the system. This server takes care of address translation and guarantees address uniqueness.

Each centralized part of the system can potentially cause the performance problem. It is possible to start a few address servers each of them for different part of network (domain), it speeds up address resolving and reduces the performance problem. While processes can communicate and migrate across this domains, communication within the same domain is more effective.

5.3. Daemon

The daemon participates in the redirection of connections. After migration, while restoring, a process has to inform its peers about its new location. The migrated process communicates with the daemon on its new host, which takes care of propagating this information.

The process after migration can not directly communicate with the peer process because it requires to break the remote process computation. It is possible to break computation by the signal handler which is sent by the daemon. Theoretically the migrated process can connect to the remote daemon without help of the local daemon but it is unclear and requires to integrate the communication protocol into the process.

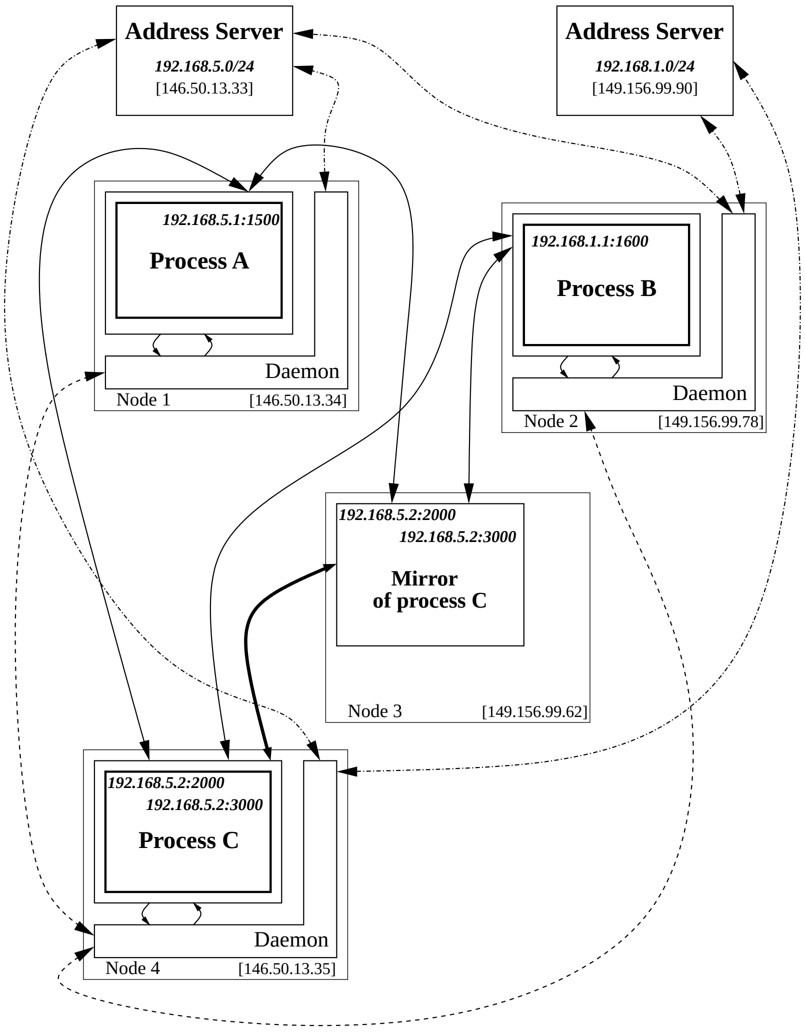


Fig. 2. Connections between address servers, daemons, processes and mirror.

5.4. *System overview*

In Fig. 2 a connection scheme between three processes and a mirror is presented. There are two address servers. One of them is responsible for the virtual network *192.168.5.0/24*, the other one for the network *192.168.1.0/24*. There are daemons running on nodes (except for `Node 3`). Process `C` is connected with `A` and `B`, and just migrates from the `Node 3` to the `Node 4`.

The real `IP` adresses of the hosts are written in square brackets. Daemons are running on the nodes 1, 2 and 4. On `Node 3` there is only the mirror of a process, the daemon is not necessary now, and may be assumed to have been terminated as the first step in bringing `Node 3` down. The daemon working on `Node 1` is connected only to one master address server, the one responsible for the *192.168.5.0* network. It is enough as the process `A` running on this node uses a socket with the virtual IP address *192.168.5.1* and this socket is connected with the process `C` and the mirror of the process `C`. In both cases the virtual address *192.168.5.2* with port number *2000* is used. The process `A` does not need information about sockets from the *192.168.1.0* network. The daemon on `Node 2` is connected to both servers because the process `B` uses the address from the network *192.168.1.0* and its socket is connected to the address *192.168.5.2*.

The process `C` has been moved from `Node 3` to `Node 4`, and now the process `C` is connected to the mirror. The process `C` also asks the daemon on `Node 4` to redirect connections. This request is propagated to all the daemons on the peer nodes, in that case to the daemons on `Node 1` and `Node 2` (dashed lines in Fig. 2). The processes `A` and `B` establish new real connections to the process `C` while the old connections still exist.

The process `A` has one virtual socket with the local address *192.168.5.1* port *1500*. This socket is connected to the virtual socket of the process `C` with the address *192.168.5.2* port *2000*. Before the migration of process `C`, there was only a connection between `Node 3` and `Node 1` while after the migration one virtual socket (a socket from the user code point of view) has two sockets in the real communication system. The socket of the process `A` is connected with the mirror and with the new process `C`.

5.5. *Redirecting* `TCP` *connections in* `ESTABLISHED` *State*

For the `TCP/IP` sockets, there is no synchronization between read and write requests on both sides of the connection. After migration the *msocket* library creates a new socket (while restoring), registers it and asks the mirror to unregister the old one. Then the system call `listen`() is invoked on the new socket. Next, the migrated process sends the *redirect request* to the local daemon. This request contains the *virtual address* of the socket and the peer, *real address* of the peer and the new *real address* of the socket.

The daemon on the node where the process migrated to receives this request and contacts the daemon at the connection peer host and sends the redirect request there. At the beginning the remote daemon opens a connection to the new process (the new *real address* is a part of the redirection request). Then the daemon tries to find the connection in its database. This entry is associated with the pid number of the local process or processes which is/are the owner(s) of the socket. The remote daemon sends the request to all of those processes and then passes the open file (new socket) to them. In this way, all peer processes share again the same socket. To force the processes to read the request and receive the file descriptor the remote daemon sends a signal to them (using `kill`()). As a result, the processes enter the signal handler which is responsible for reading the request and modifying the internal library state. As soon as the new socket is opened by the remote daemon, it is passed to all peer processes which have the connection with the migrating process, and the remote daemon sends a response to the daemon which is running on the same host as the process after the migration. From this moment on, the new connection is established, while the connection through the mirror is still present.

At the peer (`A`) of the migrated process (see Fig. 2) one virtual file descriptor `vfd`, which is used by the process code as a socket, is associated with two real sockets, `fd` and `newfd`. The former of these sockets was created before the process migration (`fd`), while the later (`newfd`) was created during the redirection. On the migrated process side only one real socket is used, plus the socket connected to the mirror, but the mirror connection is shared by all sockets.

When the migration is taking place, the peer process (`A`) could be inside the `read`() system call and could be still waiting for the data from the original `fd` which after the migration is connected to the mirror. This system call should be interrupted after the migration. This is why the migrated process sends a control request to the mirror. After having received this request, the mirror is to close half of the connection (one direction) between the mirror and the *peer* process. The mirror calls `shutdown`() on the `old_sock` file descriptor.

The *peer* process is an active part during the redirection. Each time the *msocket* library takes control[1] it checks if the connection is flushed. When the connection between `fd` and `old_sock` is flushed, the *peer* process closes the connection. Then `fd` can be set to the value of `newfd`. If the mirror detects the closing of the `old_sock` it sends a control message to the new process (`C`). The new process knows that this connection is completely redirected and all read and write requests will be done through the new socket (without the use of the mirror).

The process `A` may write more data than the process `C` can consume. The data coming from the mirror should be kept in a buffer. The process `C` stores the data inside the buffer associated with the appropriate socket.

5.6. *Architecture for* `UDP` *sockets*

Our library also uses *virtual addresss* for the `UDP` sockets. Address servers and daemons handle two separate address spaces, one for the `TCP` and the second for the `UDP`.

`UDP` is a connectionless protocol and each write request can use different destination address. This means that each write request requires address translation. On the other hand `UDP` is an unreliable protocol, so it is

[1]During all library calls and the signal handler.

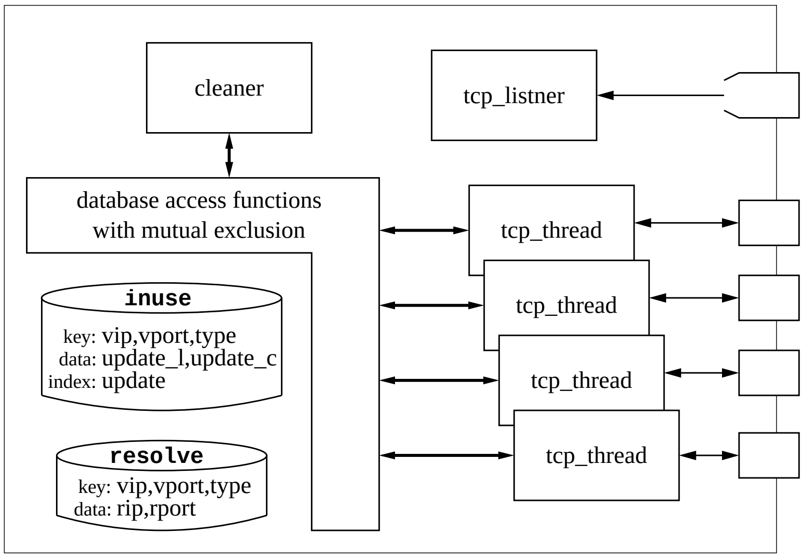


Fig. 3. The `msmaster`.

possible to create a cache for `UDP` addresses. An incoherent entry in the cache can cause packet loses, but this is not a problem because the application has to take care of retransmission of the lost packets anyway.

6. Implementation

This section provides some implementation details of the parts of the system responsible for address translation for both `TCP` and `UDP` protocols.

6.1. Architecture of `msmaster`

The main function of the address server (`msmaster`) is to keep information about the sockets in use (the pairs of IP and port numbers) and about their states. Internally, the `msmaster` keeps two tables. One of them contains the *virtual address* and the time stamp of the last verification. This table is used when an application attempts to use a new address. The second table of `msmaster` keeps the pairs of addresses: the *virtual address* and the *real address*. This table contains only addresses needed when an application is attempting to establish the communication with this address.

The address server is decomposed into threads to simplify its structure (see Fig. 3). At the beginning two threads are started. One of them (*cleaner*) periodically checks the database contents and removes old entries. The second one (*tcp_listner*) takes care of communication, waiting for incoming connections from the daemons; a new thread (*tcp_thread*) is created for each incoming connection. Each *tcp_thread* serves a single peer (an `msdaemon`).

6.2. Architecture of `msdaemon`

To simplify and speed up the communication of an application process with address servers, the `msdaemon` is used on each node. On startup, this daemon reads the configuration file, which contains the addresses of the subnets with the address of the master server for each subnet. This structure simplifies the communication because it is centralized inside the daemon, and the *msocket* stubs do not need to parse the configuration file. The `msdaemon` has a cache for `UDP` addresses to speed up the communication. The `msdaemon` also takes part in the redirection of connections. The `msdaemon` has the multiple thread structure (Fig. 4).

6.3. Communication with an application process

The daemon uses message queues to communicate with processes running on the same node (Fig. 4). Each message has the field `mtype`[2] which is used as an address. The daemon reads all messages with `mtype` equal to 1. The processes read messages with `mtype` equal to their process id.

[2]`mtype` is a field of structure `msgbuf` defined in file `<sys/msg.h>`.

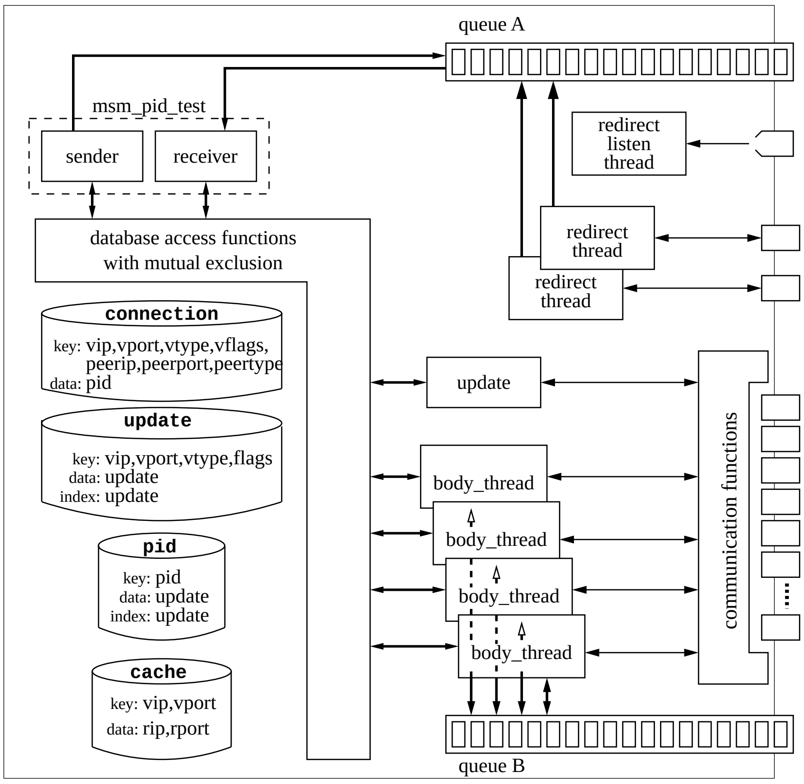


Fig. 4. The `msdaemon`.

The *msocket* daemon uses two separate message queues. One of them is used by an application process to ask the daemon to perform some operations (i.e. the address queries) while the second one is used by the daemon to force the process to e.g. redirect sockets, check if the process exists and so on.

7. Limitations

Generally, it is impossible to build a completely transparent wrapper of the socket library. The *msocket* library does not support nonblocking I/O operations. This kind of access to the socket is very often connected with the user defined signal handler, and to support this access, it is required to create a wrapper for the user signal handler. Urgent data is also unsupported; this feature of the socket stream also requires dealing with the signal handlers. Applications cannot use the socket options calls like `setsockopt`(), `ioctl`(), `fcntl`(). In the future, this limitation may be partially removed. To do this, it is necessary to retain more information about the socket state and restore it after the migration. Unfortunately, some instances of these system calls are system dependent.

The socket and file descriptors are equivalent in the Unix systems, therefore sockets can be shared between parent and child(ren) in the same way as files. This situation is dangerous for the *msocket* library because it is impossible to migrate a process which shares a socket with another process.

8. Tests of functionality and overhead

In order to verify the system, three groups of tests have been developed. The first group is designed to check if the planned functionality of the *msocket* is realized, whereas the second group focuses on measuring the level of overhead induced by the *msocket*. Finaly, the last group is developed to test the stability of the socket connections during the migration.

To verify the functionality, the following tests have been performed: creation of the virtual socket, establishing connection before the migration, establishing connection after migration(s), establishing connection during migration, migration with connection in use.

The overhead tests measure the time spent on executing the code of this library. The generic applications used in tests are typical producer – consumer and ping-pong programs.

8.1. Establishing a connection through the mirror

Two different TCP consumers were developed. Both of them require the port number to bind to as an argument. Both TCP consumers read all incoming data with the read() system call and write output data to the standard output. The first one, a single process (ms_tcpcons) accepts only one incoming connection and then closes the socket in the LISTEN state and starts reading.

The second one (ms_tcpconsfork) forks after having accepted a new incoming connection. The child process uses the new connection (in the ESTABLISHED state), while the parent tries to accept the next connection. The parent process closes the new socket after the fork whereas the child closes the socket in the LISTEN state.

When the process is checkpointed, the mirror takes care of incoming connections by accepting them. Socket states are restored by the process which starts working on a new node. The created socket is registered and then the process asks the mirror to redirect to the new location the connections accepted during migration time.

To test this functionality of the mirror, a test was proposed in which the process (consumer) is started and checkpointed, then the producer(s) is/are started. The consumer process is restored from the checkpoint file after the producer has been started.

After having restored the consumer process from the file all connections are redirected by the mirror to the new process location. All producers establish new connections with the new process (see Fig. 5).

8.2. Load swap

The producer – consumer application sends one message per second, which is enough to test the creation and the migration of the virtual sockets. It can also demonstrate that the mirror exits after the process migration. The previous tests are not suitable to show what happens with the data in the buffers and the data on-the-fly, so a test which sends as much data as possible needed to be created.

The producer sends data all the time and it is possible to adjust the size of the buffer and the number of buffers to send (number of iterations). To check the contents of the buffer at the consumer side, each buffer is filled in with a random pattern.

It is possible to set the *seed* for the random generator used by both the producer and the consumer. If the random generator is initialized with the same *seed* value it produces the same values at the producer and consumer sides, so it allows to check the contents of the data incoming from the network at the client side. The use of the values generated by the random() library call allows to check all bytes in the stream. If one of the bytes differs from the original one it is detected immediately by the consumer. If the library loses one or more bytes, all data would be corrupted, and the random generators would become unsynchronized.

The consumer (ms_rndcons) requires three arguments: the port number to bind, the size of the buffer to check (each buffer is checked separately) and the seed for the random generator. The result of each buffer checking is printed to the standard output by the consumer. Next, the index of the first different byte in the buffer is printed. At the end, the consumer prints information about the bytes read.

The producer (ms_rndprod) requires five arguments: the IP and port numbers of the consumer (or pipe, see Section 8.3), the size of the buffer, the number of iterations and the random seed. Information about each sent block of data is written to the standard output by the producer. At the end of its work, it prints how many bytes were written.

It is not necessary to use the same size of the buffer in the consumer and the producer. All data is treated as a single stream and it has no influence on errors detected at the end of the stream; the consumer checks only the fully filled buffers.

The consumer (in this case this is ms_rndcons) and the producer (ms_rndprod) are started on two different nodes **A** and **B** (see Fig. 6). Then, the process from the node **A** (the consumer) is migrated to the node **B**. Then two processes work on the same node, and at this time they are connected through a loop-back interface. Once all data buffered in the connection is flushed, the producer is migrated from the node **B** to the node **A**.

This test shows that it is possible to migrate the consumer as well as the producer. During this test a lot of data was sent (about 50 MB during each test). It has been checked that all data arrive at the consumer and not one byte is changed. The delay in data transfer resulting from the migration is about 20–30 seconds.

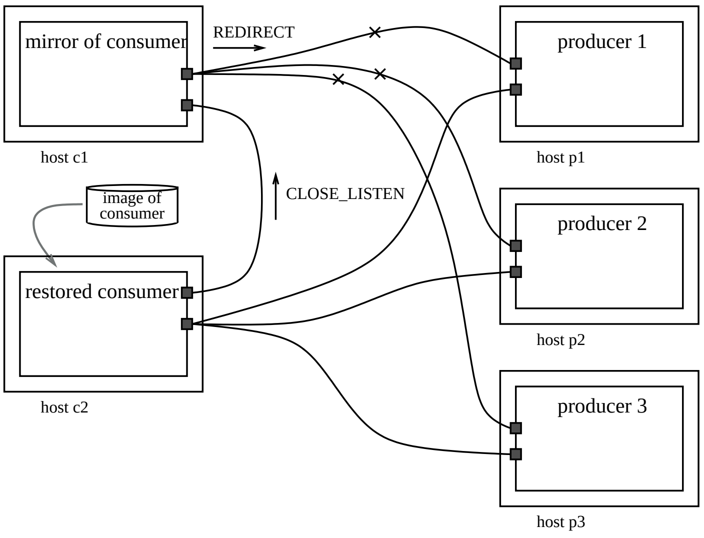


Fig. 5. Establishing a connection through the mirror.

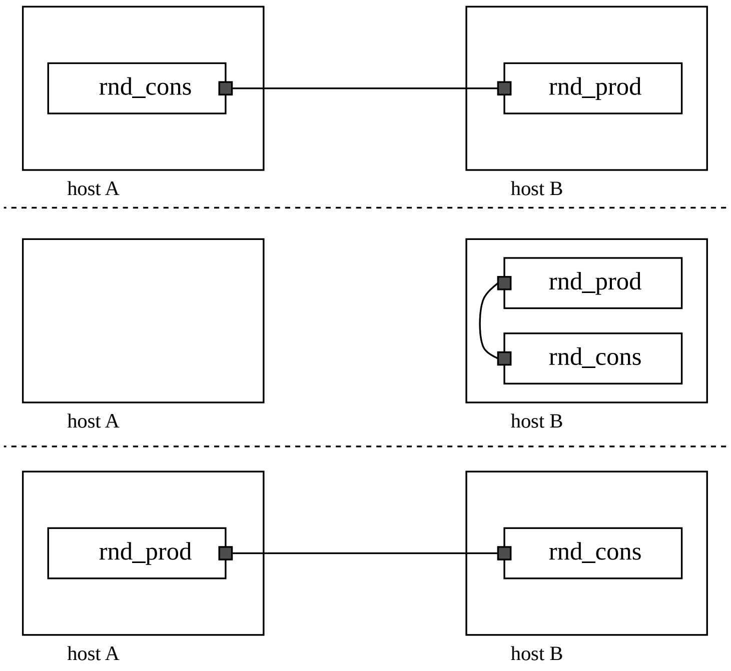


Fig. 6. The *load swap* test.

8.3. Communication through a walking pipe

For this test, the producer and the consumer are the same as in the test described in Section 8.2 above (`ms_rndprod` and `ms_rndcons`). To extend the simple producer – consumer example, the `ms_rndpipe` program was developed. It reads the data from one socket and sends it to the second one, without changing the stream of data.

The `ms_rndpipe` program expects four arguments: the `IP` and port numbers to connect to, the port number to bind to, and the size of the buffer. It informs about each block of data read and written, and at the end displays the numbers of read and written bytes.

The purpose of this test is to show that it is also pos-

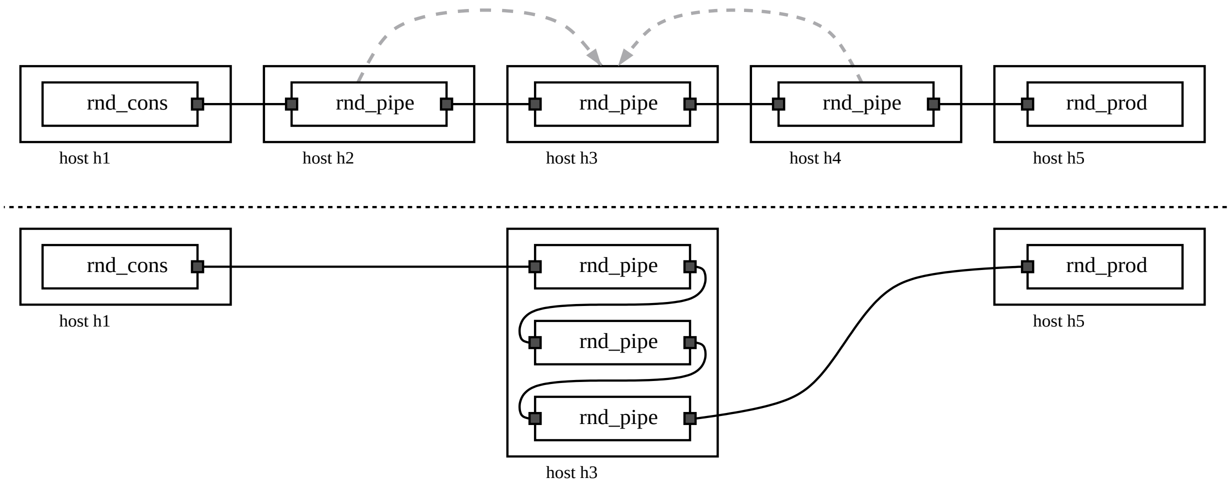


Fig. 7. The *walking pipe* test.

sible to migrate a process which reads and writes data. The producer and the consumer used in the previous test are applied. The only difference is that the producer and the consumer are connected through other processes (`ms_rndpipe`). During the test, one of the pipe processes is migrated from the node `h2` to the node `h3` and so on (see Fig. 7).

8.4. Performance tests

The performance tests consisted of checking the influence of the *msocket* library on the transfer speed and the delay caused by the data transfer.

8.4.1. Throughput test

As the first test the producer – consumer example was applied. There are two programs, both of them are compiled with and without the *msocket* library. These tests use the `TCP` protocol.

Both programs measure: wall clock time, user time and system time. The wall clock time is used to calculate the transfer speed, while the **CPU** times (user and system) are used to calculate the transfer cost per 1 MB.

We have measured the transfer speed for different buffer sizes varying between 1 byte to 8 K bytes at the producer side. The consumer buffer was always the same – 10 K bytes. The result concerning the producer was taken, so it is the test of *writing* through *msocket* link.

The first test takes place between two hosts connected by the 10 Mbps Ethernet. For each test point (buffer size) the difference between the *msocket* library and the standard `TCP/IP` was calculated. Next we calculate the weighted average. The weight of each point is equal to the distance between the given and the previous point.

The difference in the speed of the program with and without the *msocket* library is only 0.33 % for the whole range of used buffer sizes. For small buffers (between 32 to 1024 bytes) this difference is greater and is equal to 1.10 %.

The tests were also run on the DAS [4] distributed supercomputer built out of four connected clusters. Clusters use Myrinet and FastEthernet as local networks. Our tests use the FastEthernet.

The speed difference at the DAS cluster is 2.18% for the whole range of used buffer sizes and 16.15% for small buffers.

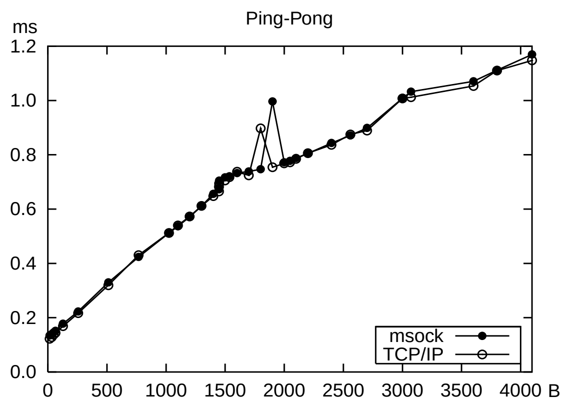
8.4.2. Delay Test – “ping-pong”

As the next performance test, the test measuring the delay of the data transfer was performed. There are two connected processes: the first one sends a message to the second one, which receives it and replies as soon as possible with a message of the same size. To increase the accuracy of the measurement, the time needed for multiple (10 000 or 1 000 000) send-receive actions was taken (see Figs 8 and 9).

While performing this test it was very important to set the TCP_NO_DELAY option on the socket. This option is supported by the *msocket* library.

The presented figures show the response time versus the size of message. The time is measured in ms and indicates the time of message transfer in both directions.

The results obtained for the program compiled with and without the *msocket* library proves that during nor-



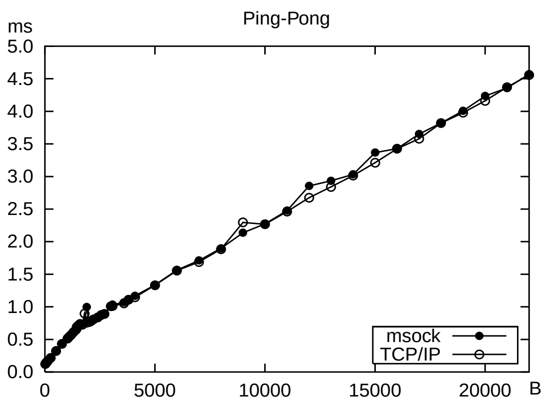
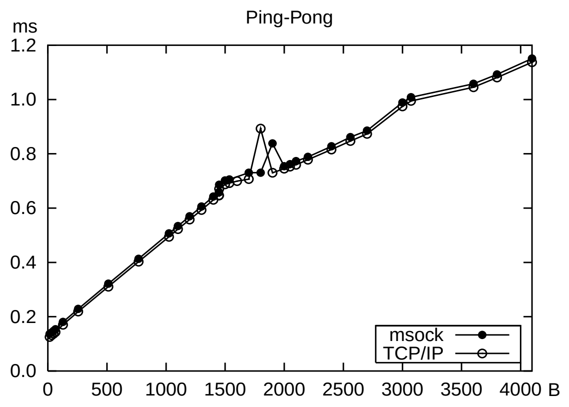


Fig. 8. Ping-pong tests, 10 000 messages.



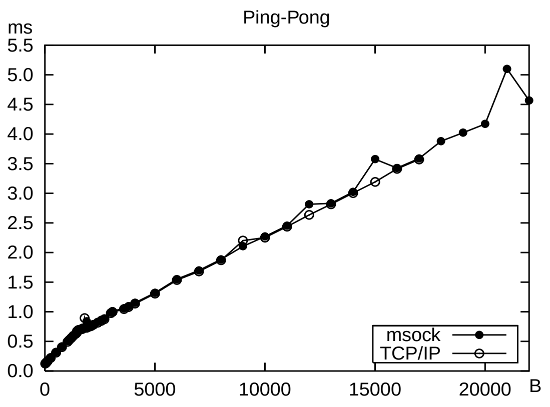


Fig. 9. Ping-pong tests, 1 000 000 messages.

mal work (without migration) the cost of using the library is insignificant.

8.5. *Stability of the migration*

Tests consisting of multiple migrations of processes were performed. The idea of this type of tests was that each process of producer-consumer was migrated fifty times. Then, the similar test was made for the ping-pong program. In this case only "pong" process could be migrated, and the migration took place 400 times. In both cases, the programs finished without a failure.

8.6. *Tests for the* UDP *protocol*

For the UDP protocol we performed only functionality tests. We tested sending packets during the process migration. We did not measure the overhead because UDP based communication is used relatively rarely.

9. Concluding remarks and future work

The *msocket* library is written in C and was tested on Linux systems. It is possible to port this environment to other system platforms, the number of system dependent parts is limited.

The current implementation of the `msmaster` and `msdaemon` can only use the TCP protocol to communicate among themselves. During the design of the *msocket* system the possibility to use UDP was considered but it was not implemented. At this time only the parsers for configuration files accept the `udp` flag as the specification of communication mechanism.

We are working on a prototype implementation of MPI (*mpich*) on top of the *msocket* library. This research is meant as the final test that the design presented here has a broader application area. A possible success will show under which conditions this is feasible. Our migrating socket library is low-level, being usable for any parallel and distributed application where communication is realized through sockets, so it may also be useful for load balancing in metacomputing environments [2].

Acknowledgements

This research is done in the framework of the Polish-Dutch collaboration and it is supported partially by the KBN grant 8 T11C 006 15. The Authors are grateful to Dr. W. Funika for his remarks.

References

[1] A. Barak, O. La'adan and A. Shiloh, Scalable Cluster Computing with MOSIX for LINUX, in: *Proceedings of Linux Expo 1999*, May 1999, pp. 95–100, E-mail: http://www.mosix.org/.

[2] M. Bubak, W. Funika, D. Żbik, G.D. van Albada, K.A. Iskra, P.M.A. Sloot, R. Wismüller and K. Sowa-Piekło, Performance Measurement, Debugging and Load Balancing for Metacomputing, in: *ISThmus 2000, Research and Development for the Information Society,* April 2000, pp. 409–418, ISBN 83-913639-0-2.

[3] J.K. Hollingsworth and B. Buck, DyninstAPI Programmer's Guide Release, Computer Science Department University of Maryland, http://www.cs.umd.edu/projects/dyninstAPI.

[4] The distributed ASCI supercomputer (DAS), http://www.cs.vu.nl/das/.

[5] K.A. Iskra, Z.W. Hendrikse, G.D. van Albada, B.J. Overeinder and P.M.A. Sloot, Experiments with Migration of PVM Tasks, in: *ISThmus 2000, Research and Development for the Information Society,* April 2000, pp. 295–304, ISBN 83-913639-0-2.

[6] K.A. Iskra, F. van der Linden, Z.W. Hendrikse, B.J. Overeinder, G.D. van Albada and P.M.A. Sloot, The implementation of Dynamite – an environment for migrating PVM tasks, *Operating Systems Review* **34**(3) (July 2000), pp. 40–55.

[7] B.J. Overeinder, P.M.A. Sloot, R.N. Heederik and L.O. Hertzberger, A dynamic load balancing system for parallel cluster computing, in: *Future Generation Computer Systems*, (Vol. 12), May 1996, pp. 101–115.

[8] J. Robinson, S.H. Russ, B. Flachs and B. Heckel, A task migration implementation of the Message Passing Interface, in: *Proceedings of the 5th IEEE international symposium on high performance distributed computing,* 1996, pp. 61–68.

[9] G.D. van Albada, J. Clinckemaillie, A.H.L. Emmen, J. Gehring, O. Heinz, F. van der Linden, B.J. Overeinder, A. Reinefeld and P.M.A. Sloot, Dynamite – blasting obstacles to parallel cluster computing, in: *Proceedings of High Performance Computing and Networking Europe,* P. Sloot, M. Bubak, A. Hoekstra and B. Hertzberger, eds, volume 1593 of Lecture Notes in Computer Science, Springer-Verlag, Amsterdam, The Netherlands, April 1999, pp. 300–310.

[10] V.C. Zandy, B.P. Miller and M. Livny, Process Hijacking, in: *The Eighth IEEE International Symposium on High Performance Distributed Computing* (*HPDC'99*), Redondo Beach, California, August 1999, pp. 177–184, http://www.cs.wisc.edu/paradyn/papers/.