Portable Library of Migratable Sockets

Marian Bubak[1,2], Dariusz Żbik[1], Dick van Albada[3], Kamil Iskra[3], Peter Sloot[3]

[1]Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland
[2]Academic Computer Centre – CYFRONET, Nawojki 11, 30-950 Kraków, Poland
[3]Informatics Institute, Universiteit van Amsterdam, The Netherlands
{bubak,zbik}@uci.agh.edu.pl, {dick,kamil,sloot}@science.uva.nl

Abstract

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

1 Introduction

In order to use distributed computing power efficiently it is essential to enable process migration from one, heavily loaded host to another, idle host, in a way that also 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, and it seems that the communication libraries have to be rewritten.

One of the systems developed to support a dynamic load balancing is Dynamite [5, 7] which attempts to keep optimal task mapping in dynamically changing environment. Dynamite balances the system by performing the individual process migrations. Dynamite is built of monitoring, scheduling, and migration subsystems [7]. 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 which are available through the same pathname before and after the migration [5].

A lot of parallel computing environments use TCP or UDP through the socket interface as a communication mechanism. This paper presents the concept and first implementation of the library which, besides the same functionality

as system socket library for the TCP/IP protocols family, allows migration of the process during communication with other processes. The new library, called *msocket*, should be a substitution of the standard socket library so no changes in the program will be required. The changes will be done at the library level so no changes in the kernel will be required, either.

2 Environments Enabling Open Socket Migration

One of the environments for migrating processes is Hijacking [8]. This system does not require any changes in the process code; changes are done dynamically after a process starts. The Hijacking system uses the DynInst [3], an architecture independent API for changing the running program. The *mutator* process attaches to the process (application) which is to be migrated. The process is stopped, and then the 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. The important disadvantage of the *shadow* is that it is using the host where the process was initiated.

Mosix [1] is a software to support the cluster computing and it is implemented on the operating system level. Each Unix version requires different implementation of Mosix, and recent 7^{th} 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 from the new host if it is possible but the interaction with the environment requires the 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; `copy_to_user()` and `copy_from_user()` kernel primitive send data through the network, and this operation is time consuming.

3 Requirements for Socket Migration

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

1. establishing new connections must be allowed regardless of the number of process migrations,
2. that all connections that had been established before the migration took place have to be kept, and data must not be lost.

The first point requires to discuss the following cases:

- migrating processes should be able to connect and keep the connection between themselves,

- migrating processes should establish and keep connections with processes which use the original socket library,
- a client using the original socket library should be able to connect to a migrating process; the location of the migrating process should have no influence on communication establishment.

The last two cases require to build a kind of a proxy server.

An essential requirement is that a parallel application with processes intended to migrate during runtime should not need to be developed in any special manner, and a 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 in the kernel source code, neither are additional kernel modules. The new library should work with both `UDP` and `TCP` protocols.

Requirement not to change the Unix kernel forces us to modify the system calls and library calls. In the user code the communication through the `TCP` and `UDP` protocols uses sockets which are treated by the process as file descriptors. For this reason wrappers are used for each call with file descriptors as arguments. In some cases this is a simple change which only translates the file descriptor number. The possibility 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 Daemons or Mirrors?

To enable 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 [4, 6]),
- 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 term *mirror* is used for a process which is used to redirect network packets. This concept was suggested in [1, 8].

Neither solution is perfect. 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 can not go down. This solution is not fault tolerant, too.

The ***msocket*** library is based on both concepts. After process migration, all connections have to be redirected to a new process location. The 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, mirror captures and redirects the packets which were on-the-fly during process migration.

During normal work, processes use the direct connection. Daemon is not used for passing user data between hosts. Daemons should exist on each machine, and

their role is only to redirect connections. 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 processes to redirect connections.

5 Idea of Migratable Sockets

In the approach proposed 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)

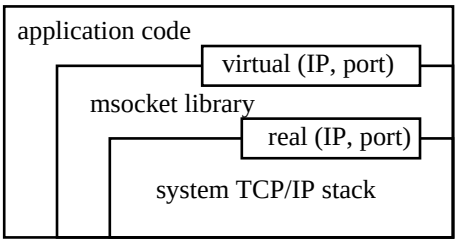

Fig. 1: Layers of the library.

This solution requires address servers which allow to find the location of a socket. The communication with address servers is required only while establishing the connection, and when the connection is established, all data goes straight between processes. Our concept requires the daemons which help just 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 protect the connection while the process migrates, a mirror is kept; it receives data from peer processes and redirects it to the new process location.

6 Architecture of the *msocket* System

6.1 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 while checkpointing of the process takes place. In our implementationt, it is started from the `ms_usersave()` function called by the Dynamite loader. The mirror is a child of the process, so it inherits all sockets used by the process before the migration. The mirror works in a loop, it reads all data from inherited sockets and sends

this 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.

6.2 Virtual address

Address of a socket is associated with the machine where the process (the owner of the socket) is running. The *msocket* library can not work in this way because the real address of the host is changing 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 addresses used for the TCP and UDP communication, and these addresses may migrate with a process. In the *msocket* library address server (msmaster) is a centralized part of the system. This server takes care of address translation and guarantees uniqueness.

6.3 Daemon

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

6.4 System Overview

In Fig. 2 a connection scheme between three processes and a mirror is presented. Two address servers are shown. 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* works on the host with number `[149.156.99.90]`.

The real numbers of the hosts are written in the rectangle brackets. Daemons are running on the nodes 1, 2 and 4. On the `Node 3` there is only the mirror of process, the daemon is not necessary now. 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 number *192.168.5.1* and this socket is connected with process `C` and the mirror of process `C`. In both cases the virtual address *192.168.5.2* with port number *2000* is used. Process `A` does not need information about sockets from the *192.168.1.0* network. The daemon on the `Node 2` is connected to both servers because process `B` uses the address from the network *192.168.1.0* and its socket is connected to the address *192.168.5.2*.

Process `C` has been moved from `Node 3` to `Node 4`, and now process `C` is connected to the mirror. 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 connections to 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 process `C` with number

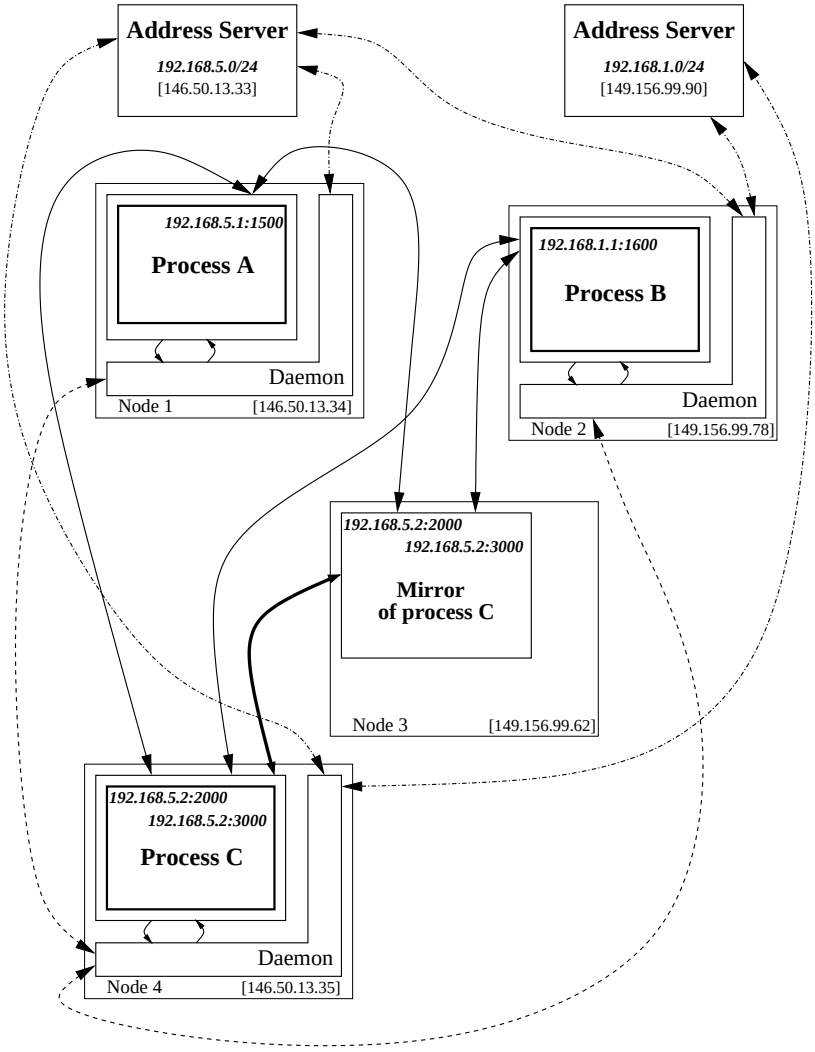

Fig. 2: Connections between address servers, daemons, processes and mirror.

192.168.5.2 port *2000*. Before the migration, a connection was only between hosts `[149.156.99.62]` and `[146.50.13.34]` and after the migration one virtual socket (socket from user code point of view) has two sockets in the real communication system. The socket of the process `A` is connected with mirror and the new process `C`.

7 Implementation

7.1 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 which are necesary when an application attempts 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*), checks the database contents from time to time and removes old entries. The second one (*tcp_listner*) takes care of communication, waits for the incoming connections from the daemons; for each incoming connection the thread (*tcp_thread*) is created. Each *tcp_thread* serves a single peer (an msdaemon).

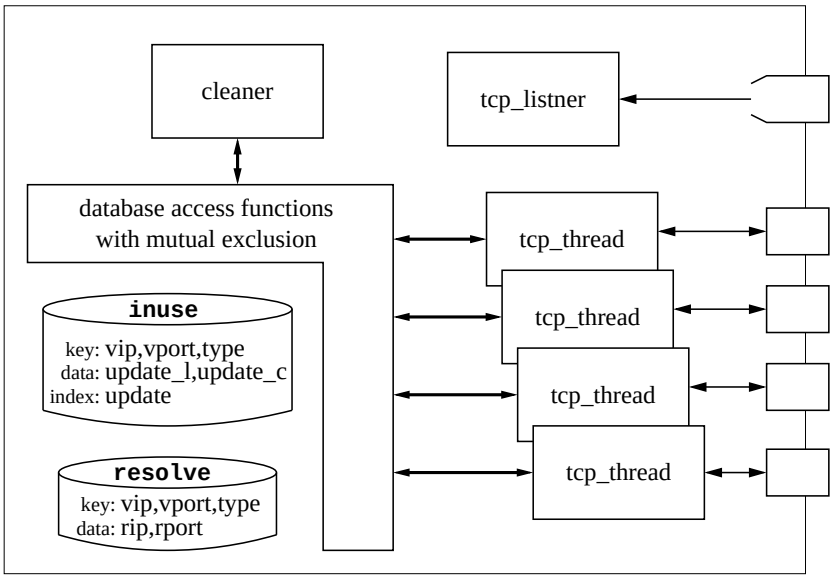

Fig. 3: The msmaster.

7.2 Architecture of msdaemon

To simplify and speed up the communication of an application process with master 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 the communication is centralized inside the daemon and the *msocket* stubs do not need to parse the configuration file. The msdaemon also takes part in the redirection of connections. The msdaemon has the multiple thread structure construction (Fig. 4).

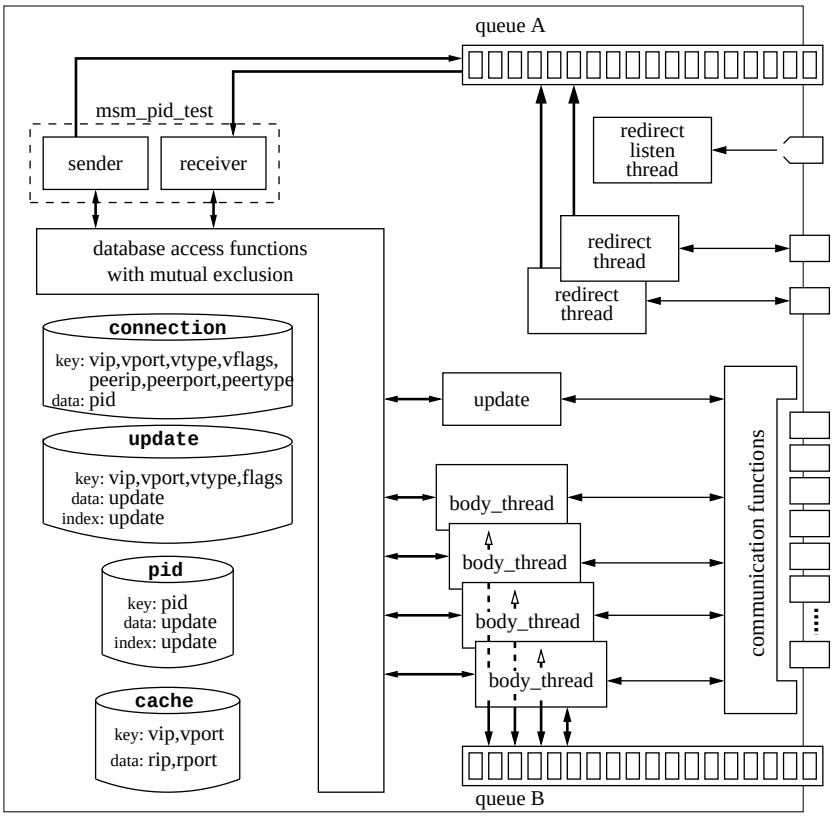

Fig. 4: The msdaemon.

7.3 Communication with an Application Process

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

The *msocket* daemon uses two separate message queues. One of them is used by an application process to ask the daemon to do some activities (i.e. the

address queries) while the second one is used by the daemon to force process to i.e. redirect sockets, check if the process exists and so on.

8 Limitations

It is impossible to build completely transparent wrapper of the socket library. The *msocket* library does not support nonblocking I/O operations. This kind of access to the socket very often is connected with the user defined signal handler, and to support this access it is required to create wrapper for the user signal handler. Urgent data is also unsupported; this feature of the socket stream also requires dealing with the signal handlers. Application can not 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 keep more information about socket state and restore it after the migration. Unfortunately, some cases of these system calls are system dependent.

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

9 Tests of Functionality and Overhead

In order to check the system development two groups of tests have been prepared. The first group was 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*.

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. Overhead tests measure the time spent in using this library. The generic application used in all tests is a typical producer–consumer program. The main features of this application are establishing the connection and communication, so it is sufficient to test the *msocket* library.

10 Concluding Remarks and Future Work

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

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

We are also working on a prototype implementation of MPI (*mpich*) above the `msocket` library. This research is aimed at the final validating whether the design presented here has a broader usage area, while 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 be also useful for load balancing under metacomputing environments [2].

Acknowledgements.
This research was done in the framework of the Polish-Dutch collaboration and it was supported partially by the KBN grant 8 T11C 006 15.

References

1. A. Barak, O. La'adan, and A. Shiloh. Scalable Cluster Computing with MOSIX for LINUX. In *Proceedings of Linux Expo 1999*, pages 95–100, May 1999. 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*, pages 409–418, April 2000. 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. 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*, pages 295–304, April 2000. ISBN 83-913639-0-2.
5. K.A. Iskra, F. van der Linden, Z.W. Hendriske, B.J. Ovreinder, G.D. van Albada, and P.M.A. Sloot. The implementation of Dynamite – an environment for migrating PVM tasks. *Operating Systems Review*, 34(3):40–55, July 2000.
6. 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*, volume 12, pages 101–115, May 1996.
7. G.D. van Albada, J. Clinckemaillie, A.H.L. Emmen, O. Heinz J. Gehring, F. van der Linden, B.J. Overeinder, A. Reinefeld, , and P.M.A. Sloot. Dynamite — blasting obstacles to parallel cluster computing. In Peter Sloot, Marian Bubak, Alfons Hoekstra, and Bob Hertzberger, editors, *Proceedings of High Performance Computing and Networking Europe*, volume 1593 of *Lecture Notes in Computer Science*, pages 300–310, Amsterdam, The Netherlands, April 1999. Springer-Verlag.
8. V.C. Zandy, B.P. Miller, and M. Livny. Process Hijacking. In *The Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC'99)*, pages 177–184, Redondo Beach, California, August 1999. http://www.cs.wisc.edu/paradyn/papers/.