Migratable sockets in cluster computing

Kamran Malik, Osama Khan, Tahir Mobashir, Mansoor Sarwar *

Software Engineering Research Group (SERG), Department of Computer Science, Lahore University of Management Sciences (LUMS),
Opposite U Block, D.H.A., Lahore Cantt., Lahore 54792, Pakistan

Received 21 March 2003; received in revised form 19 February 2004; accepted 13 March 2004
Available online 12 May 2004

Abstract

Optimal utilization of cluster computing is partly dependent upon pre-emptive process migration. However, this migration involves a host of issues, one of them being the transfer of system-dependent resources. We focus on the overhead incurred by migrated processes using sockets. We then describe a solution that we devised and implemented to avoid this overhead through the use of ‘migratable sockets’. Our studies show that the use of ‘migratable sockets’ considerably improves the execution time of a process using sockets as compared to the execution time of a process that uses standard sockets and thus bears the communication overhead.

© 2004 Elsevier Inc. All rights reserved.

Keywords: Cluster computing; OpenMosix; Sockets; Inter-process communication; Load balancing; LINUX

1. Introduction

This paper concerns the problem associated with inter-process communication between migrated processes in a Linux cluster running OpenMosix, an open source version of the Mosix (Barak and La’adan, 1998) operating system. The current inter-process communication mechanism incurs a lot of overhead. To get around this problem, we have devised the idea of ‘migratable sockets’, which enables direct communication between the migrated process and its communication end point.

In Section 2, we look at the core problem of migrating processes which have already opened socket connections or which may open sockets after migration. Section 3 covers the background material related to the problems associated with process migration. In Section 4, we discuss the solution used by OpenMosix to solve this problem and the shortcomings of their solution. Section 5 describes our solution and the changes that were needed to incorporate our solution at both the user and kernel levels. Section 6 describes a performance study that compares our solution with the existing solution. Finally, Section 7 points out ways in which migratable sockets have made cluster usage better by emphasizing the change it has brought in overall computation efficiency. It then goes on to compare our solution with some known schemes and also discusses the areas in which future work could be fruitful by pointing out further enhancements in our existing solution.

2. Problem statement

Inter-process communication in cluster computing is done through sockets. A socket is uniquely identified by the <IP address, port number> pair. If a process with an open socket connection migrates, then that socket will be invalid at the new site because the IP address will be different at the new site. Thus, if a client wants to connect to a server that has migrated to a new site, it cannot do so because the original IP address and port number will be invalid. There is a need for a mechanism by which a migrated socket should be able to establish connections meant for the original site and initiate new connections. A mechanism is needed to insure that an existing socket connection is seamlessly transferred to the new site without the other side knowing anything about it.
3. Background

Pre-emptive process migration (Barak et al., 1995) is essential for optimal load balancing in cluster computing, which has plenty of real-world uses like faster code compilation (McClure and Wheeler, 2000), supercomputer emulation etc. Process migration in a cluster should allow the system to halt execution of a process, transfer it to another node in the cluster, and allow the process execution to continue from the point before it was interrupted for migration. As long as the execution of a process does not contain any system call invocation, it can be migrated without too much of a problem since all that has to be migrated is the memory area corresponding to the user space of that process and its CPU context. Since we assume that all the nodes in the cluster are homogeneous, therefore, transferring a process state and resuming process execution is not a big problem. However, real-world processes are much more complex than that. Most processes make numerous system calls during their execution. If a process is currently executing on a remote node (i.e. it has been migrated), then there is the issue of providing the migrated process the same kernel environment that it had at its Unique Home Node (UHN), the node on which the process was spawned.

4. The existing solution for process migration in OpenMosix

OpenMosix solves the process migration problem by categorizing system calls into two types, site-dependent and site-independent system calls (OpenMosix Operating System). It also introduces the concept of a deputy (Barak et al., 1999) that is similar to a kernel thread. The deputy keeps a record of the resources that the process is attached to and the kernel-stack used for the execution of system code on behalf of the process. In short, it is basically the process’ representative at the UHN. The user context, called the remote, contains the program code, stack, data, memory-maps, and registers of the process. The remote encapsulates the process when it is running in the user level. While the process can migrate many times between different nodes, the deputy is never migrated. The migration time has a fixed component, for establishing a new process frame in the new (remote) site, and a linear component, proportional to the number of memory pages to be transferred. To minimize the migration overhead, only the page tables and the process’ dirty pages are transferred. The interface between the user-context and the system context is well-defined. Therefore it is possible to intercept every interaction between these contexts, and forward this interaction across the network.

OpenMosix intercepts all site-dependent system calls on a remote node, sends them to its deputy on the UHN, and facilitates their execution at the UHN. The results of system call execution are sent back to the remote node (where the process is currently executing). This means that all site-dependent system calls have a certain overhead associated with them. If a socket is created after the process has been migrated, then all socket system calls are intercepted by OpenMosix and sent to the UHN for execution. Even if a socket is created before a process migrates, all subsequent system calls pertaining to the socket are intercepted and sent to the UHN. For example, if a process issues a \texttt{socket()} system call after having been migrated, OpenMosix intercepts this call and routes it to the UHN where it is executed. The kernel at the UHN returns a file descriptor (FD) from the pool of file descriptors avail-
able for the process on the UHN, i.e. the file descriptor is only relevant in the kernel environment at the UHN. The file descriptor is returned back to the remote node. When the remote process needs to send data through the socket, the send() system call is intercepted and forwarded to the UHN along with the data. The UHN then uses the previously allocated file descriptor to send data using the appropriate socket. The same mechanism is used when a recv() system call is issued by the remote process. All this adds an extra hop for each packet exchanged between the two endpoints. As can be seen, this mechanism has a huge overhead for processes that use inter-process communication in a cluster. This overhead can be understood easily if the non-migration case shown in Fig. 1 is compared to the migration case shown in Fig. 2.

5. Our solution

Work has been done to handle migration related problems in other architectures, such as PVM (Casas et al., 1995). In order to solve the problem of socket migration and mitigate the overhead of process migration in cluster computing, we have made changes at two different levels. At the user space level, we have coded a C++ library that supports socket migration. This library provides all of the standard functionality associated with BSD sockets. Additionally, the library provides the facility of socket migration. Sockets created using this library automatically migrate when the process that created them migrates. This socket migration is completely invisible to the user process and takes place without any data loss. The basic idea behind the library is to maintain two distinct socket connections for each socket connection established by the user program. One of these socket connections, called the L-connection (also called the L-link), is used for the actual data communication, and is recreated every time the process migrates. The second socket connection called the M-connection (also called the M-Link) persists throughout the lifetime of the process, and is never moved away from the UHN. M-connection is used only at migration time, and then again only to pass socket migration information (MIGINFO) to the non-migrating end of the connection. The user process is only aware of the M-connection and it calls all of the sends and receives on its M-socket. The library maintains an internal mapping table and it uses this table to locate the L-socket corresponding to this M-socket. The actual send or receive is then executed on the L-socket corresponding to this M-socket. The algorithm for the send function as well as for the socket migration function is given in Fig. 3.

The process of socket migration under our scheme is illustrated in Fig. 4. Initially, processes P1 and P2 are spawned at nodes 1 and 2. P1 and P2 then open a socket

```c
L_socket (args)
{
    m_fd = socket(args);
    sys_socket(SYS_SOCKET, args);
    return m_fd;
}
L_send (m_fd, args)
{
    // returns the L-socket FD for a given M-socket FD.
    l_fd = findLSocketFD (m_fd);
    // if process migration is detected, migrate the socket
    if (detect_migration(l_fd))
        migrate_socket (l_fd);
    sys_socket (SYS_SEND, args);
}
L_recv (m_fd, args)
{
    // returns the L-socket FD for a given M-socket FD
    l_fd = findLSocketFD (m_fd);
    // if process migration is detected, migrate the socket
    if (detect_migration(l_fd))
        migrate_socket (l_fd);
    sys_socket (SYS_RECV, args);
}
migrate_socket (l_fd)
{
    // close the old L-socket at the old node.
    close(l_fd);
    socket_attributes = socket(l_fd);
    new_fd = socket(socket_attributes);
    // create new socket using old attribute values.
    if(socketType[l_fd] == SERVER)
    {
        bind (new_fd, socket_attributes);
        listen (new_fd, socket_attributes);
        accept (new_fd, socket_attributes);
    }
    else if(socketType[l_fd] == CLIENT)
    {
        connect (new_fd, socket_attributes);
    }
}
```

L-Socket FD  File descriptor for L-socket (see Fig. 2)
M-Socket FD  File descriptor for M-socket (see Fig. 3)
l_fd        Same as L-Socket FD
m_fd        Same as M-Socket FD

Fig. 3. Socket migration algorithm.

Fig. 4. Process for socket migration.
connection each for data communication. Note that even though P1 and P2 have opened a single socket connection each, and are only aware of the M-socket connection, the library has also opened an L-socket connection that is invisible to the communicating processes. All data communication between the two processes is carried out on the L-socket connection. At some point in time, OpenMosix migrates P1 from node 1 to node 3 in order to better balance the computational load across the cluster. At the time of migration, P1 leaves behind a deputy process at node 1. A socket created by P1 detects this migration in the first send/receive called immediately after the migration. At this time data communication on L-connection is temporarily suspended, and this connection is shut down. Migration information such as the new IP address of the migrating process is passed to P2 via M-connection. This is the only time that M-connection is ever used after P1 has migrated. After this, a new L-connection is set up between node 2 and node 3, and data communication resumes on this L-connection.

All communication on M-connection is preceded by an urgent signal to the receiver. In addition to using the first send or receive failure as a means of detecting migration, an urgent signal is also sent by the migrating process to all the endpoints with which it is communicating via sockets to inform them of its migration and new location. The urgent signal is used to accommodate migration of processes with multiple open connections. If the urgent signal were not used, a migrating process would have had to wait for all the other processes to do a send or receive and detect the disconnection before the migrated process could have come out of its post migration reconnection routine. By using the urgent signal a migrated process forces its receiver to initiate reconnection immediately instead of waiting until its first socket access fails.

Data loss as a result of an L-socket closing because of migration is handled by appropriately setting the SO_LINGER option of the socket. This option ensures that the data buffered in the sending queue of a socket, that is about to be migrated, is not lost because the kernel does not de-allocate the socket and its associated data structures until all the data in the buffer has been successfully sent to the receiver. However, this option alone cannot guarantee that unread data queued at the receiver is not lost if the receiving process is migrated. To handle this situation, we have added a buffer at the sender side. The receiver side sends an acknowledgement to the sender process, communicating the total number of bytes it has received on a connection as soon as:

(a) it detects a migration, or
(b) when the buffers at the sender side start to fill up and the sender side requests an acknowledgement so that it can clear up its buffers.

This entire process can be further clarified through the use of a message sequence diagram. In Fig. 5, data is transferred between nodes 1 and 2 until process 2 migrates to node 3. It notifies the non-migrating end of the connection of its migration by sending it some control information (MIGINFO). Process 1 then continues data transfer using a new connection established directly with node 3. All further communication takes place directly between the two nodes (1 and 3).

The library is implemented in a manner that makes it independent of the application that uses it. This library makes extensive use of the newly created system call sys_lsocket(), which is explained below.

At the kernel level we have implemented a new system call named sys_lsocket(). This system call allows processes to create sockets at nodes other than their UHN (Unique Home Node). Previously, a migrating process was not able to create a socket at its new node, and was forced to use the socket that it had created at its UHN.

Unlike the standard Linux kernel, OpenMosix has two separate system call tables—the standard syscall table identical to that of Linux and a remote syscall table. On invocation of a system call the kernel checks if the caller is executing remotely examining by a field in the current process’s task structure. If it finds that the process is executing remotely, it looks up the function in the remote syscall table. For every entry in the syscall table there is a corresponding function pointer in the remote syscall table that takes care of diverting the system call to the UHN. The functions pointed to by the remote syscall table are different from their counterparts in the standard syscall table in the following three aspects:

![Fig. 5. Message sequence diagram.](image-url)
1. They copy all the necessary user space data to the UHN via an M-link.
2. They have the UHN make a system call on the copied data and take other appropriate action.
3. They then copy the user space data back to the shell process’ user space.

OpenMosix maintains only the kernel environment for a migrated process at its UHN. Since all system calls are diverted to the UHN, all resource allocation is also done at the UHN. Even though the shell process, the process in which the user space of the migrated process is copied and then executed at the remote site, is a full fledged process in its own right, its per process tables like the FD tables are never used as the shell process never gets to reserve resources via system calls. Therefore, the kernel maintains only the bare minimum accounting information for a shell process. This is because OpenMosix does not allow remote processes to have their system calls executed at the remote node. Since we wanted to allocate and operate sockets at the remote node we needed a system call equivalent to the standard socket system call, which would execute at the remote node. At the same time we also wanted to retain the standard socket system call (referred to as the M-socket system call) that the library uses to exchange control information after a migration. Therefore, we decided to add our own system call `sys_lsocket()` and not modify the existing system call. For such a system call we needed to have the call bypass the OpenMosix system call interception mechanism. To accomplish this we made changes in the file Entry.S, that forced the kernel to look up the local syscall table whenever `sys_lsocket()` is called.

Process migration in OpenMosix is completely transparent to the user as well as to the programmer. Therefore, there were no means for a user program to find out if, when, and where it had been migrated. This information was important for our library since it had to initiate reconnections after migration. For instance, the `gethostname()` function was useless for this purpose because it would have returned the address of the UHN regardless of where the invoking process was currently executing. To notify the user library of migration we modified the kernel to issue a SIGUSR1 signal to any process using our `sys_lsocket()` system call. Issuing the migration signal only to processes compiled with our library was important because it would have resulted in unexpected termination of user processes that did not have a handler for SIGUSR1. We also modified the kernel to allow the library code running in the user space to find out the IP address of its current node so that it could be sent through the M-socket to the other end for reconnection.

OpenMosix Direct File System Access (DFSA) file system (Barak et al., 2000; Amar et al., 2003) does not allow processes to open non-DFSA files and allocate their file descriptors locally at the remote node. Since it also treats sockets as non-DFSA files we had to modify code in file.h and dfsa.h to make an exception to the above rule, in the case of socket file descriptors.

6. Performance study of our solution

We compared our implementation of migratable sockets with the existing solution mentioned at the beginning of the paper. To evaluate performance of our solution, we set up an echo server and a client that transferred large sized files back and forth between the client and server processes. One pair of the echo server and the client was compiled using our socket migration library and the other pair was compiled using the standard socket library available in Linux. The criterion for judging performance was total transfer time. The results are shown in the graphs reproduced below in Figs. 6–8.

6.1. Performance for the non-migration case

For the case in which a process does not migrate, the performance of socket connections created using our library is the same as that of connections created without our library, as shown in Fig. 6. This is to be expected, because without the functionality of socket migration, our library is just the standard socket library.
A high degree of linear association between the two variables, \( x \) (amount of data transferred) and \( y \) (time differential), as shown by the high value of the correlation coefficient.

Fig. 8. Performance differential of our library with migration. \( x \): amount of data transferred in MB; \( y \): difference in the data transfer times for the standard OpenMosix library and our library; \( R^2 \): correlation coefficient.

6.2. Performance for the migration case

The performance measurement of the case where the client process migrates was carried out by using three OpenMosix machines: A, B and C. The echo server was run on machine A. The client process was started on machine B and then migrated to machine C. For this case in which a process with open socket connections migrates, there is a considerable difference in performance as measured by total file transfer time, between socket connections established using our library and connections created using the standard socket library, as shown in Fig. 7. This time difference is consistent with our expectations, because our solution avoids the overhead of always going through the UHN. Note that the larger the file size, the better the performance of our library. This is understandable if you keep in mind the fact that there is a certain processing overhead associated with socket migration. But as the file size becomes very large, this overhead becomes a progressively smaller fraction of the total transfer time. Fig. 8 shows that the relationship between the differential time and the data transfer is fairly linear. That means that there is a high degree of linear association between the two variables, \( x \) (amount of data transferred) and \( y \) (time differential), as shown by the high value of the correlation coefficient \( R^2 \). Simply put, the greater the amount of data to be transferred, the more efficient our library is in terms of transfer time (the migration time is negligible compared to data transfer time).

7. Conclusions and related work

There is a strong theoretical case for the need of socket migration to fully utilize the gains of process migration in cluster computing. Our solution to socket migration for OpenMosix mitigates the post migration inter-process communication time incurred between two connected processes by providing a user level library and making appropriate changes in the kernel code to eliminate the intermediary code, known as the “deputy” in the current implementation of OpenMosix. The performance of our implementation of migratable sockets shows a considerable reduction in the total data transfer and execution times of communicating processes in comparison with the current implementation of migrated processes in OpenMosix.

There has been some work done in the past on transferring TCP connections to other machines. Pai et al. (1998), in their work on request distribution in cluster-based networks, propose a TCP handoff protocol which does handle this problem. However, this protocol does not apply to connections initiated from inside the cluster, while our solution does not impose any such restriction, i.e. it handles connection handoff between two processes inside the cluster as well as connection handoff between one end-point inside the cluster and the other outside it. If the second end-point is outside the cluster, our only requirement would be that the external end-point should also have our library installed on it. In Pai et al. (1998), they propose that all connections should be directed at a load balancer and that balancer will then decide which back-end node should handle that particular connection. All further data on that connection will be routed through the load balancer’s kernel to the appropriate node. Since the decision to divert data is taken fairly early in the network stack, the mechanism is quite efficient in terms of latency overhead. However, the load balancer is still a node through which data must pass to reach its destination, and this does represent an overhead, however minimal it may be. Our scheme improves upon it by removing the need for any intermediate entity altogether. After migration, all communication takes place directly between the two nodes.

In future we intend to explore the viability of moving all of the functionality of the socket migration library inside the kernel. This would make it possible for existing applications to benefit from socket migration without having to be recompiled with our library. Furthermore, the transfer of socket migration functionality inside the kernel would increase the processing speed and reduce the need for maintaining separate buffers in the user space.

Acknowledgement

We thank Dr. Shafay Shamail for giving feedback on the first draft of the paper.

References


