Dynamic Streams for Efficient Communications between Migrating Processes in a Cluster

Pascal Gallard and Christine Morin

IRISA/INRIA – PARIS project-team
`Pascal.Gallard@irisa.fr`
`http://www.kerrighed.org`

Abstract. This paper presents a communication system designed to allow efficient process migration in a cluster. The proposed system is generic enough to allow the migration of any kind of stream: socket, pipe, char devices. Communicating processes using IP or Unix sockets are transparently migrated with our mechanisms and they can still efficiently communicate after migration. The designed communication system is implemented as part of Kerrighed, a single system image operating system for a cluster based on Linux. Preliminary performance results are presented.

1 Introduction

Clusters are now more and more widely used as an alternative to parallel computers as their low price and the performance of micro-processors make them really attractive for the execution of scientific applications or as data servers.

A parallel application is executed on a cluster as a set of processes which are spread among the cluster nodes. In such applications, processes may communicate and exchange data with each other. In a traditional Unix operating system, communication tools can be streams like `pipe` or `socket` for example. For load-balancing purpose, a process may be migrated from one node to another node. If this process communicates, special tools must be used in order to allow high performance communication after the migration.

This paper presents a new communication layer for efficient migration of communicating processes. The design of this communication layer assumes that processes migrate inside the cluster and do not communicate with processes running outside the cluster. In the Kerrighed operating system [5], depending on the load-balancing policy, processes may migrate at any time.

The remainder of this paper is organized as follows. Sect. 3 describes the dynamic stream service providing the dynamic stream abstraction and Kerrighed sockets. Sect. 4 shows how the dynamic stream service can be used to implement distributed Unix sockets. Sect. 5 presents performance results obtained with Kerrighed prototype. Conclusions and future works are presented in Sect. 6.

2 Background

The problem of migrating a communicating process is difficult and this explains why several systems, such as Condor [4], provide process migration only for non communicating processes.

The MOSIX[1] system uses *deputy* mechanisms in order to allow the migration of a communicating process. When a process migrates (from a *home*-node to a *host*-node), a link is created between this process and the *deputy*. Every communication from/to this process is transmitted to the *deputy* that acts as the process. In this way, migrated processes are not able to communicate directly with other processes and thus communication performance decreases after a migration. The Sprite Network operating system[3] uses similar mechanisms in order to forward kernel calls whose results are machine-dependent.

Several works like MPVM[2] or Cocheck[9] allow the migration of processes communicating by message passing. However, these middle-wares are not transparent for applications. Mobile-TCP[7] provides a migration mechanism in the TCP protocol layer using a *virtual port* linked to the real TCP socket. Mobility is one of the main features of IPv6[6] but communications can migrate only if the IP address migrates. In this case, one process must be attached to one IP address and each host must have several IP addresses (one for each running communicating process). Even in this case, only one kind of communication tools (inet sockets) can migrate. Another case of communication migration is detailed in M-TCP [11] where a running client, outside of the cluster, can communicate with a server through an access point. If processes on servers migrate, access points can hide the communication changes.

None of these proposals offer a generic and efficient mechanism for migrating streams in a cluster allowing a migrating process to use all standard communication tools of a Unix system.

We want to avoid message forwarding between cluster nodes when processes migrate. We propose a generic approach in order to provide standard local communication tools like `pipe`, `socket` and *char*-devices compliant with process migration (decided for load-balancing reasons or due to configuration changes in the cluster – addition/eviction of nodes). This approach has been implemented as part of Kerrighed project at the operating system level and in this way provides full migration transparency to communicating applications.

3 Dynamic Streams

Our work aims at providing standard communication interfaces such as Unix sockets or pipes to migrating processes in a cluster. Migrating a process should not alter the performance of its communications with other processes.

A communication comprises two distinct aspects: the binary stream between two nodes, and the set of meta-data describing the state of the stream and how to handle it. Our architecture is based on this idea.

We propose the concept of dynamic stream on which standard communication interfaces are built. We call the endpoints of these streams as "KerNet

Sockets" and these can be migrated inside the cluster. Dynamic streams and KerNet sockets are implemented on top of a portable high performance communication system providing a send/receive interface to transfer data between different nodes in a cluster.

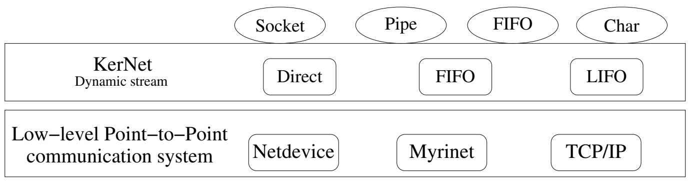


Fig. 1. Kerrighed network stack

The proposed architecture is depicted in Figure 1. Low-level Point-to-Point (PtP) communication service can be based on device drivers (such as myrinet), the generic network device in Linux kernel (netdevice) or a high-level communication protocol (such as TCP/IP). It is reliable and provides messages orders. On top of the low level point-to-point layer, we provide 3 kinds of dynamic streams: direct, FIFO and LIFO streams.

We use these dynamic streams, implemented by the *KerNet layer* to offer dynamic version of standard Unix stream interfaces (sockets, pipe...). It is a distributed service which provides global stream management cluster wide. In the remainder of this paper, we focus on the design and implementation of the KerNet layer and the Unix socket interface.

3.1 Dynamic Stream Service

We define a KerNet dynamic stream as an abstract stream with two or more defined KerNet sockets and with no node specified. When needed, a KerNet socket is temporarily attached to a node. For example, if two KerNet sockets are attached, send/receive operations can occur.

A KerNet dynamic stream is mainly defined by several parameters:

– **Type of stream**: it specifies how data is transfered using the dynamic stream. A stream can be:
  • DIRECT for one to one communication,
  • FIFO or LIFO for stream with several readers and writers.
– **Number of sockets**: number of existing sockets in the stream
– **Number of connected sockets**: it specifies the current number of attached sockets.
– **Data filter**: it allows modification of all data transmitted with the stream (in order to have cryptography, backup...).

Streams are managed by a set of stream managers, one executing on each cluster node. Kernel data structures related to dynamic streams are kept in a global directory which is distributed on cluster nodes.

3.2 KerNet Sockets

The KerNet service provides a simple interface to allow upper software layers implementing standard communication interface to manage KerNet sockets:

– `create`/`destroy` a stream,
– `attach`: to get an available KerNet socket (if possible),
– `suspend`: to unattach temporarily a socket (and to give an handle in order to be able to reclaim the KerNet socket later),
– `wakeup`: to attach a previously unattach KerNet socket,
– `unattach`: to release an attached KerNet socket,
– `wait`: to wait for the stream to be ready to be used (all required attachments completed).

KerNet provides two other functions (`send`, `recv`) for I/O operations.

The dynamic stream service is in charge of allocating KerNet sockets when it is needed, and of keeping track of these KerNet sockets. When the state of one KerNet socket changes, the stream's manager takes part in this change and updates the other KerNet sockets related to the stream. With this mechanism, each KerNet socket has got the address of each corresponding socket's node. In this way, two sockets can always communicate in the most efficient way.

At the end of a connection, a process is unattached from the stream. Depending on the stream type, the stream may be closed.

3.3 Example of Utilization of the KerNet API in the OS

Let us consider two kernel processes (*P1* and *P2*) communicating with each other using a dynamic stream. They execute the following program:

```
    Process P1                   Process P2
(1) stream = create(DIRECT, 2);
(2) socket1 = attach(stream);    socket2 = attach(stream);
(3) wait(stream);                wait(stream);
(4) ch = recv(socket1);          send(socket2, ch);
(5)                              sec2 = suspend(socket2);
                                 Process P3
(6)                              socket3 = wakeup(stream, sec2);
(7) send(socket1, ch);           ch = recv(socket3);
```

Initialization of a KerNet stream: **P1** creates the stream and requests a KerNet socket. Next **P1** wait for its stream to be ready, that is to say the two sockets to be attached. Assuming **P2** is running after the stream creation and has the stream identifier, it can get a KerNet socket, and then, wait for the correct state of the stream. The stream's manager sends the acknowledgement to all waiting KerNet sockets and provides the physical address of the other socket. With such information, KerNet sockets can communicate directly and *send/receive* communication can occur efficiently.

Migrating process using KerNet streams: If a process wants to migrate (or transfers its KerNet sockets to another process), it just uses the `suspend` function. When a process migration is needed, the KerNet socket is suspended on the departure node and re-attached on the arrival node. In our example, **P2** suspends the socket, which is latter re-attached when **P3** executes. The dynamic stream service is in charge of ensuring that no message (or message piece) is corrupted or lost between the suspend and re-attached time. The stream manager updates the other KerNet socket so that it stops its communication until it receives new information from the stream manager. When the suspended socket is activated again, its new location is sent to the other KerNet socket and direct communication between the two KerNet sockets is restarted.

4 Implementation of Standard Communication Interface Using Dynamic Streams

Obviously, standard distributed applications do not use KerNet sockets. In order to create a standard environment based on dynamic streams and KerNet sockets, an interface layer is implemented at kernel level (see Figure 2). Each module of the interface layer implements a standard communication interface relying the interface of the KerNet service. The main goal of each interface module is to manage the standard communication interface protocol (if needed).

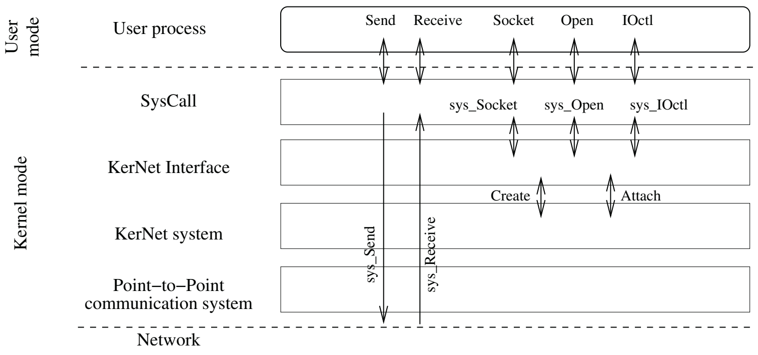


Fig. 2. Standard environment based on KerNet sockets

KerNet interfaces are the links between the standard Linux operating system and the Kerrighed dynamic communication service.

The Kerrighed operating system is based on top of a lightly-modified Linux kernel. All the different services, including the communication layer, are implemented as Linux kernel modules.

In Kerrighed operating system, the communication layer is made of two parts. A static high-performance communication system that provide a node to node service. On top of this system, the dynamic stream service manages the migration of streams's interfaces. Finally, the interface service replaces the standard functions for a given communication tool.

In the remainder of this section, we describe the Unix socket[10] interface on the KerNet sockets. We aim at providing a distributed Unix socket, transparent to the application.

In the standard Linux kernel, `Unix sockets` are as simple as unqueuing some packets from the sending socket and queuing them in the receiving socket. In this case the (physical) shared memory allows the operating system to access to the system structures of the two sockets. In the same way, the protocol management can be done easily. Obviously, in our architecture, the operating system of one node may not have access to the data structure of the other socket. Based on the KerNet services, the KerNet Unix sockets interface must manage the standard Unix sockets communication protocol.

When a new interface is defined, a corresponding class of stream is registered in the dynamic stream service. A class of stream is a set of streams that share the same properties. This registering step defines general properties of streams associated to this interface (stream type, number of sockets. . .) and returns it a stream class descriptor.

When a process makes an `accept` on a Unix socket, the Unix socket interface creates a new stream, attaches the first KerNet socket and waits for the stream allocate the other KerNet socket (as in **P1**). When another process (may be on another node) executes a `connect` on the Unix socket, an attach attempt is made (process **P2**). On success, the stream is completed and the two KerNet sockets (and by this way the two Unix sockets) can communicate directly.

The `accept`/`connect` example is a good representation of how we implement Unix sockets. With the same approach we have designed and implemented other standard socket functions like `poll` (in order to be able to use `select` syscall), `listen`, and so on. `Send` and `receive` functions are directly mapped on the `send` and `receive` KerNet socket functions.

Based on this interface, we may provide other standard interfaces such as `pipe`, `inet socket` and even some access to `char` device.

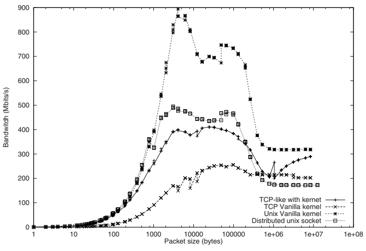
When a migration occurs (decided by the global scheduler or by the program itself), the migration service calls the `suspend` function and attaches the socket on the new node (such as **P3**).

5 Performance Evaluation

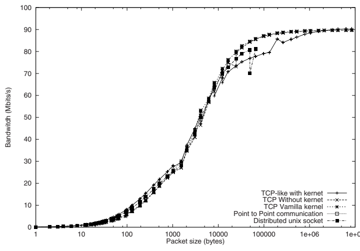
In the current implementation, KerNet provides standard Inet and Unix sockets interfaces. In order to have some performance evaluation of our communication system, we used the NetPipe[8] application. This benchmark is a ping-pong application with several packet's size. We use the *vanilla*-TCP version and a KerNet one in order to use `Unix socket`. Several physical networks are used inside the cluster: the loopback interface (Fig. 3(a)), FastEthernet (Fig. 3(b)), and Gigabits Ethernet (Fig. 3(c)).

In the FastEthernet and Gigabits Ethernet networks each node is a *Pentium III* (500MHz, 512KB cache) with 512MB of memory. The Kerrighed system used is an enhanced 2.2.13 Linux kernel.

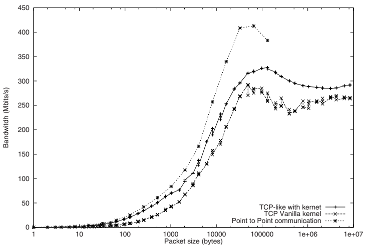
In addition, we provide in all cases, the performance of our point to point communication layer (without any dynamic functionality). We must notice that, these measures represent PtP communications from KerNet to kernel with buffers physically allocated in memory. In this case, buffers are in contiguous memory area and their size are lower than 128KB. Thus, the PtP low-level performance is a maxima for our KerNet stream.



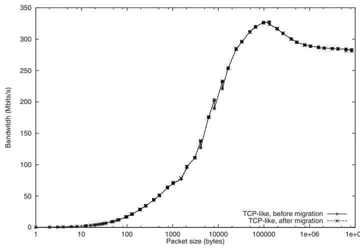
(a) Loopback



(b) FastEthernet



(c) GigaBit Ethernet



(d) GigaBit Ethernet w. migration

Fig. 3. Throughput of several communication systems

First, we notice that the interfaces have a low impact on the dynamic stream: Unix socket and TCP sockets have nearly the same results. With a *FastEthernet* network, the KerNet dynamic stream bandwidths are nearly the same than PtP low-level one. On *GigaBit ethernet* network, transfers between user-space and kernel-space are more perceptible.

When two communicating processes are on the same node, dynamic streams outperform the standard TCP sockets. This is mainly due to the small network stack in KerNet: IP stack provides some network services which are useless in a cluster or already performed by our low-level communication layer. However we do not reach (on a single node) the performance of a Unix socket. This is mainly due to the design of the low-level communication layer which as been designed for inter-node communications without any optimization for local communications.

When two communicating processes are not on the same node, KerNet outperformed TCP socket again. The reason are the same as above.

Other experiments have been performed to evaluate the impact of a migration on the dynamic stream performance. The NetPIPE application for TCP (NPtcp) has been modified to trigger a KerNet socket migration. Figure 3(d) shows that there is no overhead after a migration of a communicating process.

6 Conclusion

In this paper we have described the design and implementation of a distributed service allowing efficient execution of communicating processes after migration. We have introduced the concept of dynamic stream and mobile sockets. We have shown on the example of Unix sockets how standard communication interfaces can take advantage of these concepts.

The proposed scheme has been implemented in the Kerrighed operating system. We currently study communications in the context of the migration of standard MPI processes without any modification of the application and of the MPI library.

In future works on dynamic streams we plan to provide other stream standard communication interfaces like `pipes` and access to `char` devices. We also plan to study fault-tolerance issues in the framework of the design and implementation of checkpoint/restart mechanisms for parallel applications in Kerrighed cluster operating system.

References

1. A. Barak, S. Guday, and R. G. Wheeler. *The MOSIX Distributed Operating System*, volume 672 of *Lecture Notes in Computer Science*. Springer, 1993.
2. J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A migration transparent version of PVM. Technical Report CSE-95-002, 1, 1995.
3. F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software–Practice & Experience*, 21(8), August 1991.
4. M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison Computer Sciences, April 1997.
5. C. Morin, P. Gallard, R. Lottiaux, and G. Vallée. Towards an efficient single single system image cluster operating system. In *ICA3PP*, 2002.
6. C. E. Perkins and D. B. Johnson. Mobility support in IPv6. In *Mobile Computing and Networking*, pages 27–37, 1996.
7. Xun Qu, J. Xu Yu, and R. P. Brent. A mobile TCP socket. Technical Report TR-CS-97-08, Canberra 0200 ACT, Australia, 1997.
8. Q. Snell, A. Mikler, and J. Gustafson. Netpipe: A network protocol independent performace evaluator, 1996.
9. G. Stellner. Cocheck: Checkpointing and process migration for mpi. In *International Parallel Processing Symposium*, 1996.
10. R. Stevens. *Unix Network Programming*, volume 1-2. Prentice Hall, 1990.
11. F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Highly available internet services using connection migration. In *Proceedings of The 22nd International Conference on Distributed Computing Systems (ICDCS)*, July 2002.