SockMi: a solution for migrating TCP/IP connections

Massimo Bernaschi
IAC-CNR,
V.le del Policlinico, 137
00161 Rome, Italy

Francesco Casadei
Quadrics Ltd
Via Veneto 183,
00187 Rome, Italy

Paolo Tassotti
Università "La Sapienza",
Via Salaria, 113,
00198 Rome, Italy

Abstract

SockMi *is a solution for the migration of TCP/IP connections between Linux systems. Only the migrating peer of the connection needs to reside on a Linux system. The migration is completely transparent to the other peer that can reside on a system running any operating system.*
Our solution does not require changes to existing Linux kernel data structures and algorithms and can be activated in any phase of the connection. Both 2.4 and 2.6 versions of the Linux kernel are supported.

1. Introduction

Migration of a TCP/IP connection means to replace (at least) one of the peers with another process that can be on the same or on a different system. There are a number of situations in which the migration of TCP/IP connections can be useful. For instance, when there are requirements of load balancing, quality of service, fault tolerance, security. Hereafter, we present a solution for migrating TCP/IP connections between Linux systems, that we named *SockMi*, that can be easily adopted for a wide range of applications. With respect to other solutions, *SockMi i)* is able to migrate both ends of a connection; *ii)* does not require cooperation on both ends; *iii)* may be used by a *stream* socket in any state; *iv)* does not require *proxy* systems; *v)* may be activated via the standard `sysctl` command.

2. Related work

In this section we present a quick overview of other solutions that have been proposed to support the migration of TCP connections.
Migratory TCP [1] (M-TCP) is a reliable connection-oriented transport layer protocol that supports connection migration for building highly available Internet services. M-TCP can transparently migrate the *server* endpoint of a live connection and assist server applications in resuming service on migrated connections. M-TCP provides a generic solution to the problem of service continuity and availability in case of connectivity failures. However, it does not support the migration of the client side of a connection (whereas in SockMi both sides of a connection can migrate). Moreover, the addresses of cooperating servers must be known at connection time.
MSOCKS [2] is an architecture for transport layer mobility that allows mobile nodes not only to change their point of attachment to the Internet, but also to control which network interfaces are used for different kinds of data leaving from and arriving at the mobile node. The transport layer mobility scheme is implemented by using a *split-connection proxy* architecture and a technique called *TCP Splice* that gives split-connection proxy systems the same end-to-end semantics as normal TCP connections. In MSOCKS the TCP sequence and ack numbers are modified when they pass through the proxy. This entails that the TCP checksum must be updated. Authors claim that the overhead of this operation is limited but it is not clear whether the mechanism scales well for a large number of connections.
Reliable sockets [3] (ROCKS) and reliable packets (RACKS) are two systems that provide transparent network connection mobility and protect sockets-based applications from network failures. ROCKS does not require kernel modifications and work at *user-level*. Each system can detect failure within seconds of its occurrence, preserve the endpoint of a failed connection in a suspended state for an arbitrary period of time, and automatically reconnect, even when one end of the connection changes IP address, without loss of *in-flight* data. ROCKS are transparent to applications but they must be present at both ends of the connection.
TCP Migrate option [4] allows a mobile host to restart a previously established TCP connection from a new address by sending a special Migrate SYN packet that contains a *token* identifying the previous connection with the mobile host at the new end point. The token is negotiated during the initial

connection establishment through the use of the *Migrate-Permitted* TCP option. The migrated connection maintains the same control block and state, including the sequence number space. The main drawback of this approach is that the peer must support the non-standard *Migrate* option.
Mobile IP is a standard proposed within the Internet Engineering Task Force [5] to solve the host mobility problem, that is to maintain existing transport-layer connections as a mobile node moves from place to place. To this purpose, it allows a mobile node to use two IP addresses. The *home address* is fixed and identifies the TCP connections whereas the *care-of address* changes at each new point of attachment. The home address makes it appear that the mobile node is continually able to receive data on its home network, where Mobile IP requires the existence of a network node known as the *home agent*. Whenever the mobile node is not attached to its home network (and is therefore attached to what is termed a foreign network), the home agent gets all the packets destined for the mobile node and arranges to deliver them to the mobile node's care-of-address. Mobile IP requires both the existence of a proxy (the home agent) and the manipulation of the packets to maintain transport-layer connections. However, with respect to M-TCP, this manipulation takes place at IP (that is, network) level. Mobile IP solves the problem of host mobility, whereas SockMi tries to address the problem of connections migration (between two hosts). Recently, tcpcp, a project similar to SockMi, was presented in [6]. tcpcp is a mechanism that allows applications to transport the kernel state of a TCP endpoint from one host to another, while the connection is established, and without requiring the peer to cooperate in any way. Tcpcp consists of a kernel patch for version 2.6.4 of the Linux kernel that implements the operations for dumping and restoring the TCP connection endpoint. However, tcpcp is more a building block that can be integrated into other systems rather than a complete solution for socket migration. For instance, it does not include a general mechanism for the redirection of network traffic and there is no way to force a process to migrate a socket (the process must invoke directly the migration primitive).

3. *SockMi*

The design of *SockMi* (shown in figure 1) aimed at achieving the following goals:

Transparency: The connection end-point that does not migrate should not be affected by the migration mechanism in any way with the exception of possible (but limited) delays due to the "triangulation" mechanism described in Section 3.4. This implies that no information should be exchanged between the peers to accomplish the migration.

Flexibility: It should be possible to migrate a socket descriptor regardless of its internal state, even a unconnected socket, as if it were a file descriptor passing procedure. Note that migrating a unconnected socket is interesting especially for server-side use (*e.g.*, migrating a socket in *listen* state).

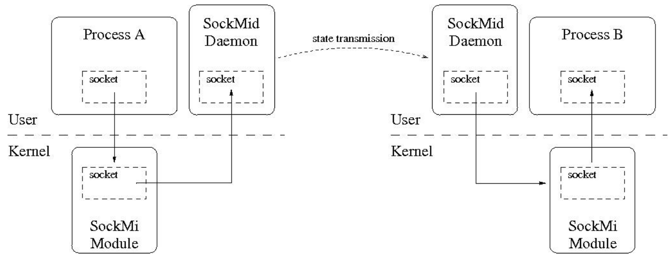

Figure 1. The design of the migration mechanism implemented in SockMi.

Portability: The migration mechanism should have minimum impact on the underlying operating system, meaning that: *i)* no *patches* to the kernel must be required; *ii)* no new system calls must be introduced. To fulfill these requirements, we implemented the migration mechanism as a loadable kernel module (**LKM**) and defined an Application Programming Interface (see Section 3.3) that hides all implementation details.

Symmetry: It should be possible to migrate both connection end-points.

Dynamicity: A connection end-point can be migrated more than once.

In what follows we are going to describe the main components of SockMi, namely the SockMi module, the SockMi daemon, the SockMi API and the IP redirection mechanism.

3.1. The SockMi module

The SockMi module is the core of the TCP/IP socket migration mechanism. In particular, the module is responsible for the following tasks: *i)* saving (restoring) the state of migrating sockets during the export (import) phase; *ii)* exchanging information about migrating sockets with the *SockMi* daemon; *iii)* providing low level primitives to activate and control the socket migration facility. The data structures involved in defining the state of a socket can be found out by inspecting the `task_struct` structure. Note that these data structures have cross references implemented as C pointers to memory locations. As a consequence, a simple approach based on data copy is not going to work, because pointers would make no sense in a different address space. Thus a primary requirement of the migration mechanism is to preserve the referential integrity among the data

structures that define the state of a socket.
Besides the state of the socket, SockMi migrates also the corresponding *"in-flight data"*. These data are found in the **receive queue** (packets received by the system but not read by the application) and in the **transmit queue** (packets to be sent, or packets already sent but not yet ACKed) of the socket. Both queues are a linked list of `sk_buff` structures that store packet data, a bunch of pointers to the header of each networking layer and some additional information about the packet itself (such as length, checksum, *etc.*).
The SockMi module holds sockets ready to be imported in three different import lists corresponding to the TCP hash tables managed by the Kernel: *i)* the *bound* sockets list; *ii)* the *listening* sockets list; *iii)* the *connected* sockets list. These lists change their length dynamically when a new socket is received from the SockMi daemon or an application imports a socket. However, to avoid potential memory problems, we set a maximum length for each list. In case a list reaches its maximum dimension, no more sockets can be queued and the import fails with an error.
The module is SMP-safe and supports the effects of the *preemption* available in the 2.6 kernel.

3.2. The SockMi daemon

The SockMi daemon (*SockMid*) works in combination with the SockMi module to support the socket migration mechanism. The daemon carries out different tasks depending on the situation. During the **export phase**, it reads the state of exporting sockets from the SockMi module internal buffers; during the **negotiation phase**, it communicates with other SockMi daemons running on other hosts in order to choose where to migrate the socket; finally, during the **import phase**, it writes the state of importing socket to the SockMi module internal buffers. During the import and the export phase, the module and the daemon components of SockMi need to exchange information about the state of migrating sockets. Since the module lives in the kernel address space whereas the daemon is a normal user process, it is not possible to resort to standard Inter Process Communication (IPC) mechanisms to pass data between them.
To overcome this difficulty we implemented a buffer sharing system via the *mmap()* primitive.
The SockMi module is seen by the daemon as a **character device** that, through its *mmap()* file operation makes its internal buffers available (*i.e.*, it acts as a *memory device*). In this way kernel buffers can be read and written by the daemon as if they were in user space.

The module starts the export procedure by sending a signal `USR1` to the daemon when a socket is ready to be exported. On signal delivery, the daemon's signal handler executes the following steps:

1. open the SockMi character device in read-only mode;
2. map, in read-only mode, the file descriptor obtained in step 1;
3. copy information about the migrating socket in a user space buffer;
4. unmap the file descriptor;
5. close the SockMi device;
6. notify the module that the buffer containing the state of the socket to be exported can be released.

A complete socket migration entails the search for a host willing to "import" the socket. To be eligible to the import of a socket, a host must run an instance of SockMid.
SockMi defines and supports a communication protocol among the SockMi daemons that run on different hosts. This *negotiation protocol*, follows a plain request-response-confirm scheme that can be summarized as follows:

- when host *A* exports a socket, it sends a request in multicast to the SockMi daemons that run on other hosts;
- when a request arrives, a host replies provided that either the socket is explicitly exported to that host or no specific target host is defined in the request;
- if host *A* does not receive a valid response within a predefined timeout period, the migration fails;
- the first valid response triggers a confirmation mechanism by which host *A* notifies all SockMi daemons that the socket has been successfully migrated.

Other responses arriving after the first one are simply discarded. This is not a problem since each daemon assumes that it has not been selected unless it receives an explicit confirmation message. Choosing the first valid response is a very natural yet simple policy. Other more sophisticated policies based on rules or heuristics could be used. For example, it could be useful to maintain statistics on previous migrations and select the target host in such a way as to achieve a *load balancing* among the hosts. Collective communication among the SockMi daemons relies on *multicast*. This means that all the instances of SockMid have to join the same multicast group and *bind()* to the same UDP port. Note that it is still possible to deploy more than one daemon group by using different multicast addresses (corresponding to different multicast groups) and/or different ports. Finally, the daemon starts the import procedure and executes the following steps:

1. open the SockMi character device in read/write mode;
2. map, in write mode, the file descriptor obtained in step 1;
3. copy information about the socket state to the kernel buffer made available by the SockMi kernel module;
4. unmap the file descriptor;

5. close the device;
6. signal that the buffer has been successfully written and that the socket is ready to be imported (as in the export procedure, an *ioctl()* primitive has been used to this purpose).

3.3. The SockMi Application Programming Interface

SockMi provides a simple Application Programming Interface (API) in order to allow applications to activate the socket migration mechanism.
A socket can be exported either by the "owner" process or by another process having appropriate rights. The API consists of two functions: `import_socket()` and `export_socket()`. These functions hide the implementation details of the migration mechanism and provide applications with an easy-to-use method for importing and exporting sockets.
To import one or more sockets, an application calls the `import_socket()` library function. This function is designed to poll the availability of "exported" sockets matching the import criteria specified by the application. If one or more matching sockets are available, then the function imports them immediately, by replacing the local sockets referenced by the input descriptors, with the "exported" ones. Otherwise, if no matching socket is available, the function waits until either a timeout occurs or one or more "exported" sockets become available for import. The prototype of the `import_socket()` function is defined as follows:

```
int import_socket(struct import_req *irqs,
                  unsigned int nirqs, int timeout);
```

It is apparent the similarity with the well-known `poll()` primitive, from which `import_socket()` inherits the event polling behaviour. The main difference between the two functions is that `poll()` does not have any side effect on the polled sockets, whereas `import_socket()` replaces the input sockets, if successful. The information required to formulate an import request are the following: *i)* the descriptor of an unconnected local socket to be replaced with the imported socket; *ii)* the preferred state the imported socket should have (any combination of *bound*, *listening* and *connected* states can be specified); *iii)* the set of criteria a socket must match in order to be imported. The import criteria let the application define the "properties" of the socket to be imported. Such criteria are the set of allowed socket states (any combination of *bound*, *listening* and *connected* is valid), the local and remote IP addresses, and the local and remote TCP ports.
Convenient default values are defined, so it is not mandatory to provide all of the fields to complete an import request. Actually, only the socket descriptor and the socket state mask are mandatory. The `import_socket()` function tries to fulfil all requests according to a *best–effort* policy.

Exporting sockets is much simpler than importing, because there is neither need to specify criteria, nor a wait time until the desired event occurs (*i.e.,* the arrival of a matching socket). To export sockets an application calls the function:

```
int export_socket(int pid, int fd,
                  int af, const void *to);
```

The first argument is the ID of the process owning the socket to be exported (a negative value being an alias for the caller). If the socket is owned by the process that invokes the function, then the export is said to be *active*. Otherwise, the export is said to be *passive*. The second argument is the descriptor of the socket to be exported. The pair (pid, fd) is used by `export_socket()` to identify in a unique way the socket to be exported. It is worth to note here that nothing prevents the application from specifying the descriptor of a previously imported socket, thus allowing a *multi–hop* socket migration. The last two arguments allow to define the network address of the importing host. The `to` argument may be a null pointer if there is no need to specify a target system. In this case the function `export_socket()` let the migration mechanism automatically select a target system, according to the internal policy of the SockMid daemon.
Finally, a socket can be exported by means of the `sysctl` command. For instance, the command line

```
$ sysctl -w sockmi.export="pid=510 fd=5"
```

exports a socket, represented by the descriptor number 5, belonging to a process whose pid is 510.

3.4 IP Packet redirection

When a socket migrates to a different host it is necessary to redirect the packets coming from the peer towards the host that imports the socket. Moreover the packets sent to the peer must have the same IP source address of the original host (otherwise the peer replies with a **RST** packet).
Actually, two cases must be considered: *i)* the host that exports the socket can give up to its IP address in favor of the host that imports the socket. This case can be managed very easily, for instance by adding an "alias" IP address to the importing host; *ii*) the exporting and the importing host maintain their original IP addresses because both send and receive data on the network after the socket migration. To deal with this, much more complex, situation we resort to a special combination of Network Address Translation (NAT) operations. In particular, we employ a **Destination NAT (DNAT)** such that packets received by the exporting host for the migrated socket are redirected to the importing host.

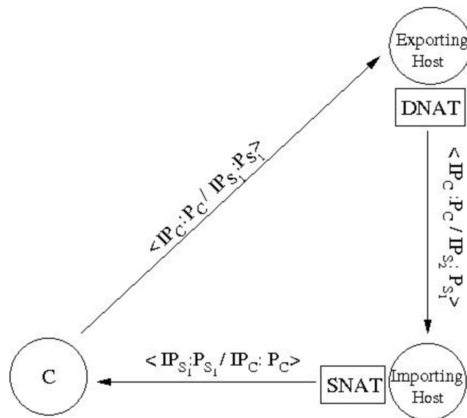

Figure 2. IP packets redirection

For this redirection the standard NAT capabilities offered by the `netfilter` module of the Linux kernel are adequate [7]. The DNAT is triggered by the SockMi daemon running on the exporting host. Besides the DNAT there is a **Source NAT (SNAT)** on the importing host such that the source address of packets sent by the imported socket is translated into the address of the exporting host.

The mechanism is represented in figure 2. Note how the mechanism is *asymmetric*: packets sent by the peer passes through the exporting host (*i.e.,* there is a "triangularization") whereas packets sent by the importing host go directly to the peer.

The SNAT required a modification to the standard NAT mechanism since the latter has a side-effect: the reply tuple is changed according to the applied address translation. The problem is that `netfilter` expects to receive packets having destination address equal to the translated source address whereas, in our case, the DNAT sets the destination address equal to the real address of the importing host.

To solve the problem we resorted to the NAT helper mechanism available in the `netfilter` architecture. Basically it allows to invoke a *custom* procedure we wrote that performs the address translation but does not alter the reply tuple.

Another issue has been the delay observed on the exporting host during the redirection of the first packet coming from the peer. This turned out to be an unpleasant *side-effect* of the fact that the DNAT rule is applied to an already established connection whereas netfilter calculates NAT bindings only for new connections. The solution to avoid this delay is to force the flush of exported connections from the netfilter table immediately after the migration. We used the `ip_ct_selective_cleanup()` netfilter function to this purpose.

The final problem we faced related to the IP packet redirection was to prevent any unexpected connection termination during and after the migration. There are two possible sources of troubles from this viewpoint: *i)* when the exporting host receives packets that it should redirect to the importing host, the TCP layer automatically sends RST packets because the connection is considered closed (note that the redirection happens at IP level); *ii)* when the process that exported the socket terminates, the TCP layer sends the FIN sequence that causes the shutdown of the connection.

The solution in this case has been to define a simple `iptables` filter that drops all FIN and RST packets sent by the exporting host to the peer. This filter is installed by the SockMi daemon on the exporting host.

4. Conclusions and future perspectives

SockMi is a mechanism, based on the cooperation of a kernel module and a daemon, that allows to migrate an end of a TCP/IP connection to another Linux system running the same software. SockMi is compatible with Linux version 2.4 and 2.6 and the source code is available from: `http://sockmi.sf.net`.

The *porting* of SockMi to other Unix-like operating systems depends mainly on the availability of the kernel source code. From this viewpoint, the porting looks possible, for instance, to the systems belonging to the BSD family.

At this time we have not analyzed yet if and how Windows systems can support the migration of TCP connections. This appears, in any case, a major effort since the implementation of sockets in Windows is significantly different with respect to Unix-like operating systems.

References

[1] F. Sultan, K. Srinivasan, D. Iyer, L. Iftode, Migratory TCP: highly available internet services using connection migration, in Proc. 22 ° ICDCS (2002).

[2] D. A. Maltz, P. Bhagwat, MSOCKS: an architecture for transport layer mobility, in Proc. IEEE Infocom (1998).

[3] V. C. Zandy, B. P. Miller, Reliable network connections, in Proc. ACM/IEEE Mobicom 2002.

[4] A. C. Snoeren, H. Balakrishnan: An end-to-end approach to host mobility, in Proc. ACM/IEEE Mobicom (2000).

[5] C. Perkins, "IPv4 Mobility support", RFC 2002, IETF (1996).

[6] W. Almesberger, TCP Connection Passing, Proceedings of the Linux Symposium, Ottawa (Canada), July 2004.

[7] http://www.netfilter.org