An IPC Software Layer for Building a Distributed Virtualization System

Pablo Pessolani[1], Toni Cortés[2], Fernando G. Tinetti[3,4] and Silvio Gonnet[1,5],

[1]FRSF, Universidad Tecnológica Nacional, Santa Fe, Argentina
[2]Barcelona Supercomputing Center & UPC, Barcelona, España
[3]III-LIDI Facultad de Informática – UNLP, La Plata,
[4]Comisión de Investigaciones Científicas, Buenos Aires, Argentina
[5]INGAR - CONICET, Santa Fe, Argentina

Abstract. Hardware virtualization technologies were principally designed for server consolidation, allowing multiple Operating Systems instances to be co-located on a single physical computer. But, IaaS providers always need higher levels of performance, scalability and availability for their virtualization services. These requirements could be met by a distributed virtualization technology, which extends the boundaries of a virtualization abstraction beyond a host. As a distributed system, it depends on the communications between its components scattered in several nodes of a virtualization cluster. This work contributes M3-IPC, an IPC software layer designed to facilitate the development of an OS-based distributed virtualization.

Keywords: Distributed Systems, Virtualization, IPC.

1 Introduction

Hardware virtualization is the most common technology that provides Virtual Machines (VM) as abstractions. Operating System (OS) level virtualization is another technology that is located at a higher level, encapsulating user-space applications within Containers or Jails or Virtual Operating Systems (VOS) [1]. On the other hand, in the 1980s, several Distributed Operating Systems (DOS) [2, 3, 4, 5] were developed as a consequence of the limited CPU performance of a single host and the need for scalability and higher computing power.

Thinking of a distributed virtualization technology seems to make sense to achieve higher performance and increase service availability. OS-based virtualization and DOS technologies lead authors to think about their convergence to achieve these goals, extending the boundaries of the virtualized abstraction to multiple hosts and thereby running multiple isolated instances of DOSs sharing the same cluster.

A distributed OS-based virtualization approach will explore aggregation with partitioning. In such systems, a set of server processes constitutes a distributed OS running within an execution context which authors call "*Distributed Container*" (DC). Processes belonging to a DC may be scattered on several nodes of a cluster (aggregation); and processes of different DCs could share the same host (partitioning).

The development of such distributed systems would be facilitated with an IPC infrastructure that provides uniform semantics without considering process location. But, as it is also a virtualization system, this IPC infrastructure should support confinement to isolate communications among processes of different DCs. In addition to these features, other characteristics are required which are related to generic distributed systems. They refer to keeping IPC operational even during process migration and supporting fault-tolerant applications which use the *Primary-Backup* approach [6]. Therefore, there is also a need to redirect communications addressed to the failed *Primary* towards one of its *Backup* processes that becomes the new primary. All these features should be complemented by a suitable performance to make Distributed Virtualization a feasible approach.

This paper proposes M3-IPC, a general purpose IPC software layer which is the foundation of a "*Distributed Virtualization System*" (DVS) [7], a new model of OS-level virtualization for Cloud Computing (IaaS). Distributed Virtualization must not be confused with Clustered Virtualization [8, 9] in which an application could run in a distributed way across a group of Containers located on several nodes of a cluster. On such systems, the boundaries of each container are limited by the node where they run, and applications must be developed using special middleware to extend APIs, which avoid the direct migration of legacy applications.

M3-IPC was developed to communicate distributed components of a VOS running within a DC, but it can be used as a powerful infrastructure for designing generic distributed applications. Although the IPC mechanism could be embedded within the DVS, the authors considered useful to build it as an independent software module. Before M3-IPC design, several IPC mechanisms were evaluated, but none of them meet the aforementioned requirements.

The next sections present some works related to IPC mechanisms, followed by a sketch of DVS components, M3-IPC concepts, design goals and implementation issues. Then, results of performance tests are discussed in Evaluation Section. Finally, Conclusions and Future Works Sections summarize the main features of the resulting software, including future improvements.

2 Motivation and Related Works

M3-IPC focuses on providing an IPC software layer to enable N-to-N communications between Client and Servers without any intermediary broker. It should facilitate the development of complex distributed systems such as an OS-level DVS. A great deal of effort had been already taken in providing IPC mechanisms for distributed systems, some of them being integrated as a component within classical DOS [10, 11] and others as added software through libraries, patches, modules, etc. [12, 13, 14, 15, 16]. Their features, semantics and performance, were previously evaluated and during M3-IPC design, implementation, and testing stages. For space limitation reasons, only a representative set of these suitable IPC software is considered here:

- Synchronous Inter-process Messaging Project for Linux (SIMPL) [12].
- Send/Receive/Reply (SSR) [13].
- Distributed IPC (DIPC) [14].

- Remote Procedure Call (RPC) [15].
- Telecommunications IPC (TIPC) [16].

All of the above presented IPC mechanisms were used for performance comparison against M3-IPC as it is shown in Evaluation Section. URPC [17] was not considered because, although it may reach a good performance, it is weak on issues related to security. Messages Queues, Pipes, FIFOs, and Unix Sockets do not have the ability to communicate with remote processes. DIPC presents interesting features, but it is no longer maintained.

One of the most important features needed to build a DVS is IPC isolation, which includes the following properties:

1. *Confinement*: A process running within a DC cannot communicate with any process within another DC.
2. *Private Addressing*: An endpoint number (an M3-IPC address which identifies a process or thread) allocated to a process running within a DC could be allocated to other processes running within other DCs.
3. *Virtualization Transparency*: A process does not need to know that it is a member of a DC or the node in which is running, or in which nodes other processes of the same DC are running. To allow DVS management, privileged processes could allocate endpoints and DCs for other non-related processes.

None of the evaluated IPC software meet all of these requirements considering that a DC could span several nodes of a cluster.

3 M3-IPC Design

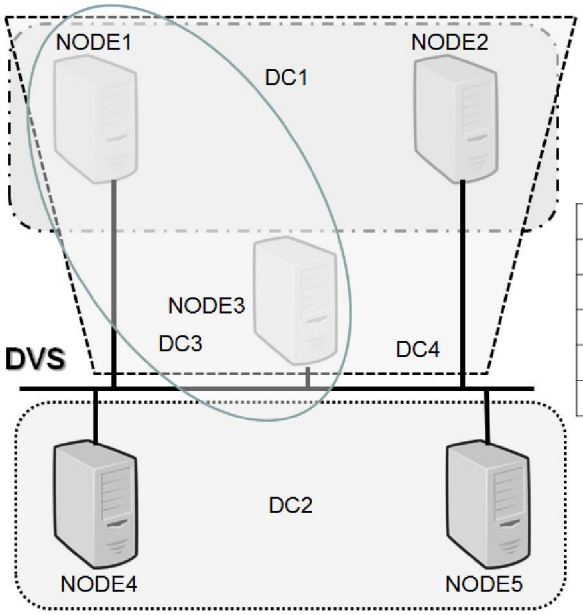
This section describes the design outline and rationale for M3-IPC to provide local and remote IPC for centralized and distributed systems. It presents the DVS topology model for which it was originally designed, its design goals, main concepts and implementation issues.

3.1 DVS Components

A DVS consists of the following components (Fig. 1):

- *DVS*: It is the top level layer that assembles all cluster nodes and it embraces all DCs.
- *Node*: It is a computer that belongs to the DVS where processes of several DCs are able to be run. All nodes are connected by a network infrastructure.
- *DC*: It is the group or set of related processes that might be scattered on several nodes. M3-IPC only allows communications among processes that belong to the same DC. The boundary of each DC can be based on administrative boundaries. A DC hides its internals from the outside world and hides network communication issues from its processes.
- *Endpoint*: An endpoint is an integer number which identifies a process/thread registered in a DC. Endpoints are unique and global within a DC, but could be repeated within other DCs.

M3-IPC does not impose a network/transport protocol to be used for inter-node communications. It allows programmers to choose the protocol that best fit their needs. Nodes communicate among them through *proxies*.



	DC1	DC2	DC3	DC4
NODE1	X			X
NODE2	X		X	X
NODE3			X	X
NODE4		X		
NODE5		X		

Fig. 1. DVS topology model.

3.2 M3-IPC Design Goals

The following design goals were established:

- *Easy-to-use*: Defining, implementing, and deploying applications using M3-IPC should be straightforward.
- *Location Transparency*: The application should not need to consider the location of the involved processes in order to facilitate programming.
- *Confinement*: IPC must be confined so as to achieve a proper degree of isolation.
- *Concurrency*: It should support threads to maximize throughput and efficiency.
- *Transparent Live Migration Support*: The application would use process migration [18] as a mechanism to achieve workload balancing and downtime avoidance.
- *Transparent Fault Tolerance Support*: IPC must remain operational even during process switching from Primary to Backup [6] without the awareness of the other peer processes.
- *Performance and Scalability*: Performance goals are considered satisfied when meeting minimal thresholds established for local and remote transfers. IPC throughput between processes located on different nodes must be limited by the network capability (latency and bandwidth) and the transport protocol being used.
- *Communication Protocol Agnostic*: Developers can choose the Network/Transport Protocol to be used by proxies that best satisfies their needs.

- *Modular and Customizable*: It should be able to be used by any kind of application with similar requirements.
- *Client/Server and RPC Oriented*: M3-IPC must cover typical communication patterns between multiple clients against multiple servers.
- *Security*: Basic security principles must be applied to its design and implementation.

M3-IPC is based on the assumption of a cluster of network nodes that are connected to a switched LAN where the packet loss rate is low and the available bandwidth is high.

3.3 M3-IPC Main Concepts

M3-IPC was designed to emulate the semantics of a multi-server OS within the Linux kernel providing user-space APIs. Furthermore, it expands its use to processes located on remote nodes of a cluster, and includes features that could be required by distributed applications such as application isolation, location transparency, and message redirection on process migration or process switching on replication.

As M3-IPC supports threads to maximize concurrency, the following references to processes are also valid for threads.

M3-IPC APIs are classified as:

- *Communication APIs*: Those APIs emulate Minix 3 [19] IPC primitives. They are related to message and data transfers among processes. They do not include any reference to the DC a process belongs to, or to the node where it resides.
- *Management APIs*: Those related to DVS, DCs, proxies, nodes, and process management, which allow the mapping of applications to nodes and DCs.

A process which will use M3-IPC must be previously registered at the kernel to bind a DC to it. The process can register by itself, or by another local process with management privileges. Every process bound to a DC will have a unique *endpoint* number with backup endpoints as exceptions.

Management APIs consider: *Local endpoints* are those allocated to processes running on the same node; and *Remote endpoints* are those allocated to processes running on other nodes.

Every remote server should be first registered in the client's node specifying in which node is running. Once a remote endpoint is registered by the kernel, all local registered processes (including other clients) within the same DC are able to refer to that server endpoint in communication APIs. Remote client binding is easier because M3-IPC can automatically bind remote unprivileged endpoints when their messages arrive to the local node.

M3-IPC supports message redirection on node switching when a Primary process (on node A) is replaced by a Backup process (on node B). The local M3-IPC kernel automatically unlinks the endpoint from the previous *Primary* process (on node A) and links it to the new *Primary* process (on node B). A local process can be registered as a *Backup* of a remote *Primary* process with the same endpoint in the same DC, but messages sent by other local processes addressed to that endpoint will be sent to the *Primary* process. The *Backup* process will not be able to communicate with any other process until it is promoted as the new *Primary* by a privileged process. Message replication among *Primary* and *Backup* processes does not concern M3-IPC.

M3-IPC can keep communications in operational state and can automatically redirect messages on live process migration. Before a process begins to migrate, a management application must call *mnx_migr_start()*. At that moment, all messages addressed to the

migrating endpoint will be queued up. Once the process has successfully migrated, the management application must call *mnx_migr_commit()*. All queued messages will be sent to the process on its new location. If the migration fails, the management application must call *mnx_migr_rollback();* then the process can resume its execution as if nothing had happened by receiving the queued messages.

3.4 M3-IPC Implementation Issues

This section describes the implementation details of the M3-IPC core within the Linux kernel.

M3-IPC APIs use a software interrupt vector or interrupt gate that differs from that used by Linux system calls. This issue allows both APIs to share the same kernel with minimal interference or crosstalk. It also facilitates future maintenance of M3-IPC in current and future Linux kernels.

Two important decisions made before developing M3-IPC were: its construction being based on software components that are available in the Linux kernel, and taking advantage of the parallelism in SMP and multi-core systems. Before starting the development, several Linux kernel synchronization mechanisms and mutual exclusion facilities were evaluated. *Kernel semaphores* and *RCU* (Read-Copy-Update) were too slow; and *spinlocks* were somewhat slower than Read/Write locks (*rwlocks*). Finally, *mutexes* were used for mutual exclusion, but since they have a similar performance to that of *rwlocks*. A compile option is available for system programmers to choose which synchronization mechanisms and mutual exclusion facilities want to be used.

Another design decision was related to the mechanism for data transfer from user-space to kernel-space and vice versa. *Netlink sockets* [20] have a quite complex start-up which do not meet DVS project needs. *Efficient Capability-Based Messaging* (ECBM) [21] has an impressive performance, but basic security issues were neglected. Finally, functions *copy_to_user()* and *copy_from_user()* provided by the Linux kernel were used. A custom function named *copy_usr2usr()* was built to copy data from the user-space buffer of a source process to the user-space buffer of a destination process.

The most outstanding characteristics of how M3-IPC was finally implemented are summarized in the following list:

- APIs support threads and were implemented using Linux kernel provided mutual exclusion facilities; Task Queues and Event Waiting were used for process synchronization; and Reference Counters were used to hold a count of processes from which a data structure is referenced.
- The granularity of internal critical sections was maximized at the process/thread level, allowing parallel message transfers among multiple pairs of processes.
- Higher performance is achieved because DCs do not share any data structure among one another during concurrent message transfers between pairs of processes that belong to different DCs.
- Data structures that are frequently used for registered processes (*struct proc*) are aligned with L1 cache lines to reduce access time.
- An Affinity Inheritance Protocol (similar to the well-known Priority Inheritance Protocol) was implemented to minimize performance impact of cache ping-pong. Thus, the facility provided on Linux to specify process affinity with a set of processors/cores was used. With this approach, there is a greater chance for the

message to remain in L1 cache when the destination process is scheduled, thus reducing access time.

- A CPU mask could be allocated by each DC to specify on which CPUs its local processes are able to run (only meaningful within the each node).
- The copy of data blocks and message transfers among co-located processes are made from the source process address space to the destination process address space without any intermediate copy through the kernel. The copy is made by the Linux kernel through the copy-on-write mechanism.
- Debugging information is sent to the Linux kernel ring buffer and can be shown by means of ***dmesg*** command.
- Data blocks which are page-aligned and whose lengths are greater than or equal to the page size are copied using the kernel provided *page_copy()* function, which is very efficient because it uses MMX instructions.
- Information about configuration, status, and statistics of M3-IPC abstractions is presented as directories and files within Linux ***/proc*** filesystem.

M3-IPC has been implemented in C programming language on Linux for Intel x86 32-bit and it is distributed as a kernel patch, a kernel module, and a set of libraries.

Proxies and IPC through the Network

M3-IPC uses application level proxies for communications between nodes, but also support kernel-level proxies too. Therefore, there is no restriction about the network/transport protocol proxies can use.

M3-IPC APIs provides a function to get messages from the kernel that need to be sent to remote nodes, and another function to insert messages into the local kernel coming from remote nodes to local processes as destinations.

The current M3-IPC distribution provides the use of one pair of proxies (sender-receiver) for each remote node. Proxies exchange *proxy messages* which consist of fixed length headers and, eventually, variable length payloads (data blocks).

The provided proxies send/receive messages and data through the network without taking account to which DC those messages or data belong to. Custom proxies may consider implementing encryption, compression, filtering, message logging, QoS, etc.

4 Evaluation

This section is devoted to M3-IPC performance evaluation against other IPC mechanisms. The compliance of the other design goals was verified during and after the development stage using several testing scenarios.

Two types of communication tests are presented in Fig. 2: 1) between co-located processes; 2) between processes located on different nodes. Furthermore, two types of micro-benchmarks were performed on each one: a) message transfer; b) data copy. The following common communication pattern was used: a server waits to receive a message from a client, and immediately replies. Once the client sends the request to the server, it waits for the reply. The reply may be a message (36 bytes) or a block of data (36 bytes to 64 Kbytes).

Although several performance metrics were evaluated as latency, CPU usage, network usage, by space limitations, only the throughput results are presented here.

The tests were made on a cluster of 8 (eight) quad-core Intel(R) i5 CPU 650@3.20GHz with a memory bandwidth of 20,841 Gbytes/s for 64 Kbytes blocks (reported by *bm_mem*), linked by a 1 Gbps dedicated LAN switch.

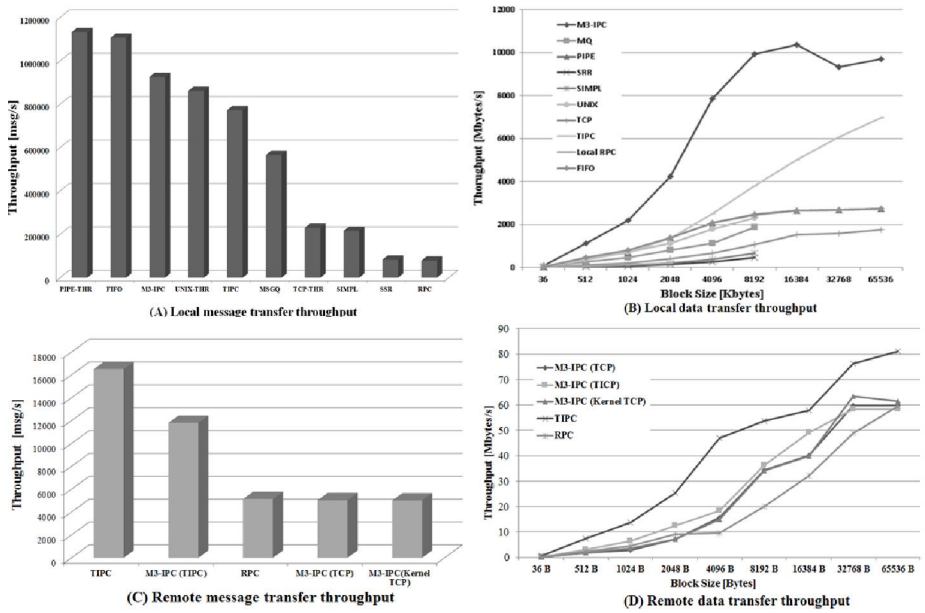


Fig. 2: Results of performance tests.

4.1 Tests between Co-located Processes

Tests between co-located processes allow the comparison of M3-IPC performance versus other IPC mechanisms available on Linux.

One of the design goals states that the expected performance should be as good as the fastest IPC mechanisms available on Linux. The following IPC mechanisms were tested using custom and [22, 23] provided micro-benchmarks: *Message Queues, RPC, TIPC, FIFOs, pipes, Unix Sockets, TCP Sockets, SRR, SIMPL.*

4.1.1 Message Transfer Performance

The presented results (Fig. 2-A) summarize the best throughput achieved by the IPC mechanisms running a single pair of client/server processes. Linux IPC mechanisms with the highest performance were pipes and named pipes (or FIFOs) followed by M3-IPC (925,314 [msg/s]).

Another micro-benchmark of message transfers between multiple pairs of Client/Server processes was run to evaluate performance in concurrency. The highest average throughput was 1,753,206 [msg/s], which was reached with 4 pairs of Client/Server processes (4 cores).

4.1.2 Data Copy Performance

As it is shown in Fig. 2-B, M3-IPC performance surpasses other IPC mechanisms on Linux. The reasons of this behavior are: 1) M3-IPC performs a single copy of data between address spaces while the others perform at least two copies (Source to Kernel, Kernel to Destination); 2) it requires a lower number of context switches; 3) it uses the Linux kernel provided *page_copy()* function which uses MMX instructions.

4.2 Tests between Processes Located on Different Nodes.

This section presents performance results of M3-IPC against RPC and TIPC.

M3-IPC does not consider flow control, error control, or congestion control. Those issues are delegated to proxies and the protocol that the use. Reference implementations of M3-IPC proxies use TCP and TIPC as transport protocols.

4.2.1 Message Transfer Performance

As it can be seen in Fig. 2-C, TIPC has the highest throughput. M3-IPC using TCP on proxies has throughput similar to RPC.

The remarkable performance of TIPC suggested that it could be a good option to be used by M3-IPC proxies as transport protocol. M3-IPC versatility and flexibility in proxy programming allowed authors to modify the source code of proxies in a few minutes so as to use TIPC instead of TCP. These changes result in an improvement of performance, emphasizing the impact that the transport protocol has on its throughput.

4.2.2 Data Copy Performance

As shown in Fig. 2-D, TIPC presents the highest throughput of 81[MB/s] that confirms results presented in [16]. The highest throughput achieved by RPC and M3-IPC was about 60[MB/s]. Fig.2-D also shows that there is no noticeable difference in performance when using TIPC instead of TCP as transport protocol on M3-IPC proxies to copy data blocks.

5 Conclusions and Future Works

IaaS providers always need higher levels of performance, scalability and availability for their virtualization services. These requirements can be met by a distributed virtualization technology. As a proof of concept, a DVS prototype using M3-IPC was developed.

M3-IPC addresses some issues about thread support, location transparency, message redirection on process migration, network-transport protocol agnostic, and IPC confinement for virtualization. The results show that M3-IPC achieves all its performance related design goals, with a high throughput for both intra-node and inter-node messages and data transfers.

To improve M3-IPC security and isolation issues, its integration to Linux Capabilities and *cgroups* [24] are being considered for further research and future works.

Until the development of M3-IPC, to the best of authors' knowledge, there was not a modular IPC software for Linux that fully satisfied the particular communication needs

of a DVS. This suggests that many questions remain about software for programming communications related to virtualization.

References

1. D. Hall, D. Scherrer, J. Sventek: A Virtual Operating System. Journal Communication of the ACM, 1980
2. David R. Cheriton; Willy Zwaenepoel: The Distributed V Kernel and its Performance for Diskless Workstations. Proceedings of the 9th ACM Symposium on Operating Systems Principles (1983)
3. Morin, C., at al.: Kerrighed and data parallelism: cluster computing on single system image operating systems. IEEE Computer Society (2004)
4. Barak A.; La'adan O; Shiloh A.: Scalable Cluster Computing with MOSIX for Linux. Proc. 5th Annual Linux Expo Raleigh (1999)
5. Pfister, Gregory F.: In Search of Clusters. Prentice Hall 1998, ISBN 0-13-899709-8.
6. Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg: The primary-backup approach. In Distributed systems (2nd Ed.), Sape Mullender (Ed.). ACM Press/Addison-Wesley Publishing Co., New York, NY, USA 199-216, 1993
7. Pablo Pessolani, Toni Cortes, Silvio Gonnet, Fernando G. Tinetti: Sistema de Virtualización con Recursos Distribuidos. (Spanish) WICC 2012. Argentina, 2012
8. J. Turnbull: The Docker Book. ISBN 978-0-9888202-0-3. 2014
9. Hindman, Benjamin, et al.: Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In: NSDI. 11: 22-22. Retrieved 12 January 2015.
10. Andrew S. Tanenbaum, Gregory J. Sharp and De Boelelaan A: The Amoeba Distributed Operating System. 1992
11. Barrera, J.: A Fast Mach Network IPC Implementation. In Proceedings of the USENIX Mach Symposium, November 1991.
12. John Collins and Robert Findlay: Programming the SIMPL Way. ISBN 0557012708, 2008.
13. SRR- QNX API compatible message passing for Linux. http://www.opcdatahub.com/Docs/booksr.html
14. M. Sharifi, K. Karimi: DIPC: A Heterogeneous Distributed Programming System. In Proceedings of the 3rd Annual Computer Conference of the Computer Society of Iran, 1997
15. Andrew D. Birrell and Bruce Jay Nelson: Implementing remote procedure calls. ACM Transactions on Computer Systems, 1984
16. Jon Paul Maloy: TIPC: Providing Communication for Linux Clusters. Proceedings of the Linux Symposium, 2004
17. Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, Henry M. Levy: User-Level Interprocess Communication for Shared Memory Multiprocessors. 1991
18. Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. "Process migration". ACM Computer Survey 32, 3, September 2000
19. Tanenbaum A., Woodhull A.: Operating Systems Design and Implementation. Third Edition Prentice-Hall. 2006
20. Ryan Slominski: Fast User/Kernel Data Transfer; Master Thesis. April 20. 2007
21. Luca Veraldi: Efficient Capability-Based Messaging (ECBM). 2003 (Italian)
22. Michael Kerrisk: The Linux Programming Interface; No Starch Press, ISBN 978-1-59327-220-3, 2010
23. ipc-bench: http://www.cl.cam.ac.uk/research/srg/netos/ipc-bench/.
24. CGROUPS. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt