Capabilities for per Process Tuning of Distributed Operating Systems

David Margery, Renaud Lottiaux, Christine Morin

To cite this version:

David Margery, Renaud Lottiaux, Christine Morin. Capabilities for per Process Tuning of Distributed Operating Systems. [Research Report] RR-5411, INRIA. 2004, pp.13. inria-00070595

HAL Id: inria-00070595
https://hal.inria.fr/inria-00070595
Submitted on 19 May 2006

Capabilities for per Process Tuning of Distributed Operating Systems

David Margery, Renaud Lottiaux, Christine Morin

N˚5411

December 2004

Systèmes communicants

ISSN 0249-6399 ISRN INRIA/RR--5411--FR+ENG

Capabilities for per Process Tuning of Distributed Operating Systems

David Margery, Renaud Lottiaux, Christine Morin

Systèmes communicants
Projets Paris

Rapport de recherche n˚5411 — December 2004 — 13 pages

Abstract: This paper introduces capabilities as a tool to tailor cluster mechanisms usage on a per process basis in a single system image operating system. Using capabilities, processes should only pay the cost in terms of OS resources for the operating system of the mechanisms they want to use. We relate capabilities to policies that are used to try to ensure optimal resource usage across all cluster nodes. Finally, we discuss implementation of these ideas in Kerrighed and describe its usage on a real industrial application.

Key-words: Cluster, SSI OS, Linux, Capabilities

(Résumé : tsvp)

Utilisation des capacités pour le paramétrage des processus d'un système d'exploitation distribué.

Résumé : Cet article présente l'utilisation de la notion de capacité pour le paramétrage des processus d'un système d'exploitation distribué, en particulier un système d'exploitation à image unique pour grappe. Grâce à l'utilisation de capacités, un processus ne subira que le coût des mécanimes distribués qu'il utilise. Nous faisons le lien entre les capacités et les politiques qui tentent d'assurer une utilisation optimale des ressources au niveau de la grappe. Pour finir, nous discutons de la mise en œuvre de ces idées au sein de Kerrighed et de leur utilisation sur une application industrielle.

Mots-clé : Cluster, système à image unique, Linux, Capacité.

1 Introduction

Designing a Single System Image (SSI) Operating System (OS) for a cluster implies giving the illusion of an SMP machine to the programs running on the cluster. This implies that the operating system should be able to perform a number of actions in a transparent manner. These actions include process and thread migration for load balancing, transparent access to shared memory for multi-threaded applications as well as the possibility of using inter-process communication channels such as sockets and pipes even thought the different processes are not located on the same node of the cluster and can migrate.

Therefore, a number of mechanisms need to be implemented and integrated in the operating system to implement the SSI illusion. These mechanisms should enable transparent access to the resources of the cluster by a process wherever the process is located. These resources are the processors, the memory, the network interfaces and the disks.

Furthermore, due to the multi-machine nature of a cluster, new functionality needs to be added to the operating system to enable the addition or retrieval of new machines in the cluster, where retrieval could be related to the failure of one node. We believe an SSI OS should take into account possible node failure and be designed in a way to limit its impact. Therefore fault tolerance mechanisms should be considered.

Nevertheless, once these mechanisms are working, two problems arise. The first is to decide when or how to use these mechanisms. This is the policy problem. The second is if a mechanism should be used for a given process. This is the capability problem.

Policy management deals with optimal resource management at the cluster level. The simplest example is that of the global scheduling policy. The basic mechanism involved is process migration, but being able to migrate a process is not sufficient to use the available processors efficiently. This is why a policy is needed. Deciding what optimal resource management is depends on the requirements of the cluster's owner, therefore the system should enable the implementation of different policies for different usages.

Capability management deals with fitting resource usage by a process with needed mechanisms. Most of the mechanisms provided to give the illusion of an SMP machine come at a cost. This cost can be in setup time, memory usage for the kernel data structures or time to execute system calls. Furthermore, one may want to limit the mechanisms that can be used by a given process for confinement or security reasons. Therefore, capability management offers a mean of enabling or disabling the mechanisms available to a process, so that resources it has access to and uses can be limited. This benefits the process because of reduced cost in resources, and it benefits the cluster because its administrator can limit the capabilities that a user or a process can use.

We have also found that capabilities are a very useful tool for cluster management. For example, on an SSI OS, the `ps` program that shows the running processes will give a global view of all processes running on the cluster because the */proc* virtual file-system will be globalized. If the cluster's administrator gives to the `ps` program the capability to see only the local */proc*, he will be able to see the programs running on the local node.

Finally, capabilities help in the debugging of Kerrighed features, because they enable control of the mechanisms used by a program, limiting possible interactions to the ones that are being debugged.

This paper is structured in the following way. Part 2 describes mechanisms used to implement the different services a SSI OS needs to offer. Part 3 explicits some of the policies that need to be considered for an SSI OS. Part 4 then defines the use of capabilities in such an OS. We then discuss related work in part 5. Part 6 discusses implementation of theses ideas in Kerrighed and gives examples of usage. We finally conclude.

2 Mechanisms

In this section we give a very brief overview of the mechanisms implemented in KERRIGHED,and of the functionality they enable.

2.1 Containers

Containers ([6]) are kernel level mechanisms that enable the sharing of objects (ie. pages or inodes) between any number of nodes of a cluster. Providing each access to the object is preceded by either `get_object` (read access) or `grab_object` (write access) and followed by `put_object`, containers provide sequential consistency between the different copies of the object.

Containers enable implementation of a distributed shared memory for a process (DSM), page injection as well as coherent access to the contents of a file, whether mapped or opened.

2.2 Ghosts

Processes in KERRIGHED are manipulated thanks to the ghost [9] mechanism. KERRIGHED can save a process to a ghost and create a process from a ghost. The ghost mechanism separates the save and creation logic from the storage media. Therefore, when different storage medias are used, different high level mechanism are implemented. Storage to a disk implements checkpointing, whereas storage and creation to and from the network card implements process migration or fork on a distant node.

2.3 KERNET streams

KERNET streams [5] are streams with mobile extremities. Sending messages on a KERNET stream only involves in most cases the sender node and the receiver node, whatever the number of times one of the extremities has moved. KERRIGHED implements standard unix interfaces on top of KERNET streams : unix sockets, inet sockets and pipes. Therefore, when using KERNET streams, programs implemented using processes communicating through streams can be deployed with each process on a different node.

3 Policies

Mechanism provide the building blocks to implement functionality. But most functionality must be implemented according to some policy. In this section we give a brief overview of policies that are applied in a SSI OS, but need to be tuned to best fit the needs of programs or cluster configurations.

3.1 Scheduling Policy

Implementing a global scheduler for a cluster that fits all possible workloads of a cluster is a challenging task. In contrast to other systems which implement a single policy (Mosix [2], OpenSSI [1]), we have chosen in KERRIGHED to provide a framework that enables configurable and adaptable dynamic scheduling ([10]). Therefore, KERRIGHED as an OS does not have to choose between a receiver initiated scheduling policy and a sender initiated policy, because it is possible to choose a specific scheduler at configuration time, as well as to change the scheduling dynamically while the cluster is running.

Moreover, in the case of programs running multiple processes, it should be possible for the user of these programs to give hints to the system on what kind of scheduling is best suited for this particular program.

3.2 Checkpointing Policy

In the same way, a single checkpointing policy [3] does not fit all programs. Checkpointing aware program might explicitly request checkpoints, whereas other programs may need different automatic checkpointing policies.

Here again, hints need to be given to the system to have it apply the best checkpointing policy for a given workload or a given program.

3.3 Memory Management Policy

Some kind of policy has to be implemented to manage memory at the cluster level. Indeed, with the page injection mechanism [4], it is possible to use memory from other nodes when memory from one node is saturated. But different strategies could be used to choose the node used for injection. It could be statically chosen (a few nodes in the cluster are dedicated as memory servers), dynamically chosen (the system chooses to dynamically dedicate a node for injection) or chosen using a round robin policy.

4 Capabilities

4.1 POSIX Capabilities

The capability interface is described in a withdrawn POSIX draft (1003.1e [7]). It permits association of a capability to a process or a file, enabling and disabling of a capacity associated to a process and transmission of capabilities to the next program associated with a process. In particular, in most operating systems, some actions requested by a process are only performed if appropriate privileges are possessed by the process. This can be implemented by checking if the process has the capability needed for that action. The capability flags defined by the POSIX draft are permitted(P), effective(E) and inheritable(I). These flags apply to each capability separately. When they are associated to files, the permitted flag may be called forced(F) and the inheritable flag may be called allowed(A). For the purpose of this explanation, the effective flag of the file will be noted E_f

Transmission of capabilities is a simple copy for `fork` or `clone` system calls, and during `exec`, the following rules apply for the flags (P', E', I') of the new process (as described for Linux in [8]):

$$\begin{aligned} P' &= F \mid (\, A \,\&\, I \,) \\ E' &= P' \,\&\, E_f \\ I' &= I \end{aligned}$$

Because the Linux Virtual File System (VFS) does not support capabilities for files, the A and E_f flags are set to 1, and F is set to 0. We therefore have the following simplified rules:

$$\begin{aligned} P' &= I \\ E' &= P' \\ I' &= I \end{aligned}$$

4.2 Kerrighed Capabilities

4.2.1 Rational

Capabilities defined by the POSIX standard are mostly security related. In this article, we suggest that these capabilities could also be used by a cluster operating system to enable or disable cluster related mechanisms on a per-process basis.

Three use-cases motivate our work. Because KERRIGHED is a SSI OS, standard programs such as `top` and `ps` should reflect the state of all the nodes in the cluster. But because individual nodes may need administration, privileged users should be able to run, at the same time, the same programs (and thus the same binaries) to see the state of a particular node. Or, for debugging purposes, a user might want to see what is happening on a precise node using the same tools. Here, usage of the inheritable flag of the shell starting the tools could be enough to achieve our goal. Another similar example is to run the same program from a shell with the capability to create processes on any nodes of the cluster and from a shell that does not have this capability for performance comparison purposes.

As can be seen in this example, a capability is used to describe if a specific mechanism (here globalization of `/proc` or usage of all nodes for child processes) should be used for a process. Capabilities are therefore given an additional role on top of security: control. Nevertheless, capabilities do not specify the policy to be used with that mechanism. In the previous example, having the capability to use any node does not have any influence on the policy that will be used to choose the node on which child processes will be created. It only specifies that the process is allowed to create new processes on other nodes of the cluster.

The second use case motivating our work is the deployment of programs that create a hierarchy of processes. This would be the case for parallel invocations of make or for legacy programs written for clusters of nodes running independent OSs. As illustrated in figure 1, these programs can be described as a global coordinator coordinating local coordinators that themselves coordinate the working processes. Parallel `make` or MPI fit well is this pattern. The objective here is to create the local coordinators on different nodes by managing the capabilities of the global coordinator because from the user's perspective, the global coordinator is the program he is running.

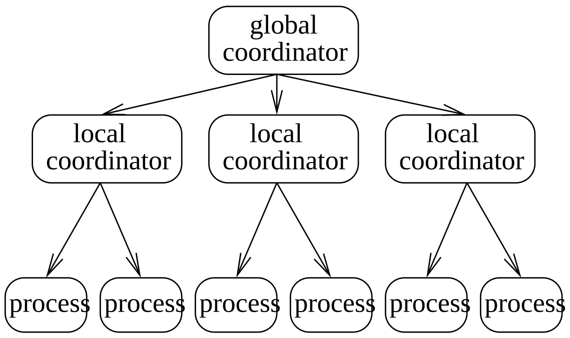

Figure 1: *Two stage fork pattern for program running on a cluster*

Using the standard Linux capability flags and transmission rules, it is not possible for a process to transmit capabilities to its son programs that should not be transmitted to their sons. Here, if the global coordinator is allowed to fork processes on any node, the local coordinators will also have the capability, leading to deployment nightmare.

Using the E_f flag for the file containing the program could solve the problem, but the first use case implies that the file stays the same. Moreover, using the E_f flag implies changing the VFS layer of Linux, which we want to avoid.

The third use case motivating our work is the case of multi-thread programs that where not designed to run over a software shared memory. Indeed, by default, the thread library of KERRIGHED will attempt to create each thread on a different node. This can lead to catastrophic performance in some cases. If this is known in advance, the process can be created without the capability to use distributed shared memory, thus preventing deployment on different nodes.

4.2.2 Definitions

We therefore introduce an additional capability flag, the inheritable effective (IE) flags, as well as the corresponding capability transmission rules, and for clarity reasons, I is renamed IP for inheritable permitted. :

$$\begin{aligned} P' &= F \mid (A \,\&\, IP) \\ E' &= IE \,\&\, E_f \,\&\, P' \\ IP' &= IP \\ IE' &= IE \end{aligned}$$

A process with capabilities in its set of flags might not use them because they are not needed during the life-cycle of the process. Therefore, a given process might use or not use some of the capabilities of the SSI OS that he is entitled to use. This usage or non-usage must be tracked in a way or another:

The *used* counter, which track capabilities that are used by the process. Such capabilities cannot be revoked, because it could lead to inconsistent states of the program with respect to its permitted flag, which could in turn lead to failure of the program because some operations could no longer be performed. For example, a process communicating with a distant process using the migratable pipe mechanism cannot loose the permission to use migratable pipes, without first closing all dynamic streams in use.

Therefore the system should check that the used counter is not positive before revoking a capability.

The *unavailable* counter, which track capabilities that cannot be used by the process. Such capabilities cannot by used, because using them will break the system. For example, a process having an open communication stream created without the capability to use migratable streams should not be able to migrate until that stream is closed. Therefore each time a non migratable stream is opened by a process, the counter associated to the migration capability should be incremented.

Therefore, the system should not only check that a given capability belongs to the effective capabilities of a process before using it, but also that the unavailable counter for that capability is not negative.

4.2.3 Usage

4.2.4 Available Capabilities

The following capabilities are defined in KERRIGHED. Not all of those capabilities are completely implemented or respected by the underlying OS services. Nevertheless, some of them are.

CAP_CHANGE_KERRIGHED_CAP Only processes with this capability can change their capabilities.

CAP_USE_CONTAINERS This capability will be used to check if containers should be used for that process, in the event of migrations. This capability is mandatory for processes using shared memory through system V IPC shared segments or through threads. For sequential processes it commands behavior of address space migration. With this capability set, pages are migrated as they are needed. Without it, all pages are migrated at the same time as the program.

CAP_CAN_MIGRATE This capability is used by the default scheduler to decide if it can migrate a program. For the time being, it is not used by the *migrate* program, nor when migration is requested by the program itself.

CAP_DISTANT_FORK This capability is used by the fork system call to decide if it should try and fork the new program on a distant node. Effective usage of a distant node is not guarantied, as this is a policy decision. Nevertheless, if deemed desirable by the scheduling policy fork will create the new process on a distant node.

CAP_CHECKPOINT This capability will be used to decide if a process is allowed to create checkpoints.

Checkpoint should be exportable to an other Kerrighed cluster

CAP_USE_INTRA_CLUSTER_STREAMS On creation request for streams (pipes, unix and inet sockets), this capability is used to decide if KERRIGHED streams should be used or if native Linux streams should be used. Usage of native streams will increment the *unavailable* counter for CAP_CAN_MIGRATE and CAP_DISTANT_FORK capabilities.

CAP_USE_INTER_CLUSTER_STREAMS This capability is reserved. It is used to decide if the created streams should be streams which can have an extremity on an other KERRIGHED cluster.

CAP_USE_EXTRA_CLUSTER_STREAMS This capability is reserved. It is used to decide if the created streams should be streams which can have an extremity on an unknown host.

REOPEN_TERM Used for debugging. This controls whether after migration or creation of an additional thread the program should reopen terminals it was using on the new local node or if it should maintain a connection to its original terminal. When debugging, this capability enables the users to see the output of the program on the node it is produced, which is useful is this output contains debugging data.

SEE_LOCAL_PROC_STAT Needed to see the local */proc*, because by default processes see a globalized */proc*.

DISABLE_DFS Needed to see the local disks, because by default processes see a globalized file system. (*i*e. `ps`)

4.3 Extended Kerrighed Capabilities

Kerrighed Capabilities are used to control what mechanism of the operating system can or should be used by processes running on the system. However, they do not control how these mechanisms are used.

To enable policy tuning on a per process basis, we associate two attribute vectors to all processes in the system. The first vector controls how KERRIGHED mechanisms behave for the current process, whereas the second is the vector transmitted to the son processes when the process forks.

Each mechanism has an associated attribute in the vectors, whose interpretation depends on the mechanism.

For example, for the CAN_CHECKPOINT capability, this attribute is used to decide if checkpointing should be automatic or asked for by the program. It can also be used to indicate the preferred storage method for the checkpoint (in memory, on stable storage, etc).

Extended capabilities are a standard method to attach information used by policies to a process rather than to the code describing the process. In the same way that ioctls are generic and standard way of exchanging information with the kernel using special files, extended capabilities are a standard way of exchanging information with KERRIGHED's policies using the capabilities of a process.

As policies are applicable on a per process way, extended capabilities therefore provide a standard interface to access additional context information that should be used by a given policy when dealing with a given process.

An extended capability attribute is a self defined type, with no references to other data structures, as it needs to be meaningful whatever the node the process is hosted on.

Here, Kerrighed should only provide the basic infrastructure to manage Extended capabilities. This infrastructure consists of a Kerrighed capability for each policy that specifies if the user of the system can change the extended capability related to that process. If that capability is granted to the user, Kerrighed provides an interface to write and to read the values present in extended capabilities. Additionally, the administrator of the system is granted privilege to change default Kerrighed capabilities given to processes on the cluster, and the default values associated to each extended capability.

5 Related work

We know of no other SSI OS that implements that kind of functionality. Capabilities have largely been used for finer control of programs for security reasons, but not as a mechanism to control of general OS functionality.

6 Implementation

6.1 Capability Manipulation

Capabilities are used at three levels:

1. at the system level. Here, functions that need to test if a given process can use a capability use the `can_use_krg_cap` function, with tests the unavailable counter as well as the effective flag for a given capability.

2. at the program level. The `krg_capset` and `krg_capget` functions enable a program to manipulate its capabilities.

3. at the command level. Here, the `krg_capset` program can change the capabilities of its father process. It will be extended to be able to change the capabilities of any program belonging to the same user as the user calling the program.

6.2 Example Usage

Legacy programs running on a cluster are often structured using a two stage forks: start a given program on every node (using `rsh` or `ssh`), and then let each copy start as many programs as necessary on its local node to perform their part of the task. This is the case for some implementations of `mpirun` for example. Using Kerrighed, starting a copy of a program on every node should be implemented using `fork`, with the CAP_DISTANT_FORK bit set in the effective flags, to take advantage of all the properties of the system. This can be done by using a fake `rsh` program (`krg_rsh`), that does the necessary capability management and then calls `fork` thus avoiding modifications to the code of the MPI middleware used.

Implementing `krg_rsh` comes down to the following steps:

1. add the CAP_DISTANT_FORK flag to the effective flags of the current process,
2. fork,
3. restore the old effective flags,
4. exec the specified program.

If the inheritable effective (IE) flags do not contain the CAP_DISTANT_FORK bit, the son process will fork processes without that flag set, leading to a local fork. Using inheritable effective flags therefore enables the two stage fork problem to be solved in an elegant fashion.

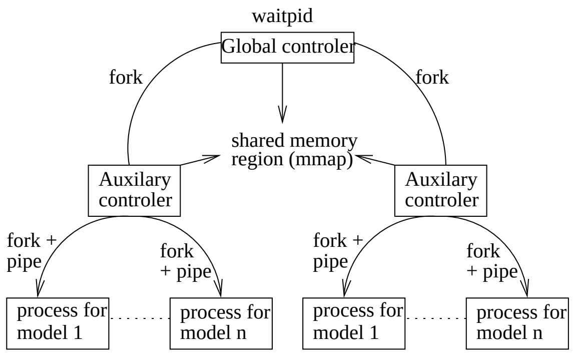

Figure 2: *Topology of processes in parallel* LIGASE

The second example is that of LIGASE, an legacy program studied in the framework of the COCA project. For the purpose of this article, LIGASE can be described as a program used to run parametric studies that uses the process topology described in fig 2. This topology is well suited for parallel execution on an SMP. By adding the CAP_DISTANT_FORK and CAP_USE_CONTAINERS to the effective flags of the global controller, this program was deployed very efficiently on a KERRIGHED cluster. The mmaped memory segment used for supervision is shared through containers, whereas locally, on each node, only standard unix mechanisms are used, to avoid any impact on performance (at the time of the test with this application, loopback of KERRIGHED streams was not always as efficient as loopback of standard pipes). Here, capabilities have effectively helped us achieve the desired results, by allowing control of the mechanisms used by this application.

The last example involves the use of SEE_LOCAL_PROC_STAT to enable monitoring of the cluster on a per-node base. Indeed, by default, programs like `xosview` that monitor resource usage by reading system informations available to user space will read information related to the cluster as a single machine. This is the case because an SSI OS should give the illusion of a single machine. Nevertheless, it can be useful to monitor the cluster node by node, for example to check that the scheduling policy effectively balances the workload evenly between all nodes. Here, the same program (ie `xosview`) can be used, provided the SEE_LOCAL_PROC_STAT is set in its effective flags. It must nevertheless be started on each node, and prevented from migrating (CAN_MIGRATE is not set in its effective flags). It is probable that such monitoring processes should not be checkpointed. Here again, usage of capabilities can ensure this.

7 Conclusion and Future Work

Capabilities as described here have been for the most part implemented in the current version of KERRIGHED, which is available under the GPL license at `www.kerrighed.org`. They are actively used on a day to day basis as a control and debugging tool by the KERRIGHED team.

Future work as far as capabilities are concerned involve implementation of extended capabilities and the associated tools and their usage by the different policies used in KERRIGHED.

8 Acknowledgments

The work presented here has been partly funded by the Direction Générale de l'Armement (DGA) for the COCA project, involving INRIA, Capgemini and ONERA.

References

[1] http://openssi.org/index.shtml.

[2] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System, Load Balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[3] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[4] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 201–212. ACM Press, 1995.

[5] Pascal Gallard and Christine Morin. Dynamic streams for efficient communications between migrating processes in a cluster. *Parallel Processing Letters*, 13(4), December 2003.

[6] Renaud Lottiaux and Christine Morin. Containers: A sound basis for a true Single System Image. In *Proceeding of IEEE International Symposium on Cluster Computing and the Grid (CCGrid '01)*, pages 66–73, Brisbane, Australia, May 2001.

[7] Ieee std 1003.1e. http://wt.xpilot.org/publications/posix.1e/, 1998.

[8] Boris Tobotras. Linux capabilities faq 0.2. http://ftp.kernel.org/pub/linux/libs/security/linux-privs/kernel-2.2/capfaq-0.2.txt, 1999.

[9] Geoffroy Vallée, Jean-Yves Berthou, Renaud Lottiaux, David Margery, and Christine Morin. Ghost process: a sound basis to implement new mechanisms for global process management in linux clusters. Research report, INRIA, IRISA, Rennes, France, December 2004.

[10] Geoffroy Vallée, Christine Morin, Jean-Yves Berthou, and Louis Rilling. A new approach to configurable dynamic scheduling in clusters based on Single System Image technologies. In *Proc. 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, page 91, Nice, April 2003. IEEE.

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399