

Towards an Efficient Single System Image Cluster Operating System

Christine Morin, Pascal Gallard, Renaud Lottiaux

*IRISA/INRIA, PARIS project-team, Campus de Beaulieu 35042 Rennes Cedex,
France*

Geoffroy Vallée

*EDF, Industry Branch, Research and development division, 1 avenue du général
De Gaulle, BP 408, 92141 Clamart cedex, France*

Abstract

The lack of a single system image Operating System (OS) for clusters restricts their use for parallel processing. We propose an approach for building an efficient single system image cluster operating system. The proposed system implements distributed services performing global and dynamic resource management to offer high performance, high availability and ease of use and programming. The programming API of the OS running on each cluster node is kept unmodified but the high level OS services can take benefit of all cluster resources. Our approach has been validated by a prototype based on Linux. Our prototype comprises of a limited kernel patch and a set of modules extending the kernel to implement the cluster distributed services. Existing applications running on symmetric multiprocessors (SMP) on top of Linux can be executed on top of our cluster OS without modification.

Key words: Cluster, operating system, distributed system, single system image

1 Introduction

Clusters are now attractive for executing high performance applications. However, cluster programming is difficult as clusters suffer from a lack of dedicated Operating System (OS) providing a Single System Image (SSI). An SSI provides the illusion of a single powerful and highly available computer to users and programmers of a cluster.

In this paper, we present our work on the design and implementation of Kerrighed¹, a cluster OS providing an SSI. This system is designed to support the execution of high performance sequential and parallel applications. Both the shared memory and the message passing parallel programming models are supported. Multithreaded applications based on Posix threads and OpenMP applications as well as MPI or socket based applications can be executed without modification on top of a Kerrighed cluster.

Several application domains may take benefit of a cluster executing Kerrighed. For example, our system can be used for executing numerical simulation applications, for supporting Web servers or as a departmental computing server. Our main goal is to combine ease of programming and use with high performance and high availability. Our system achieves these goals by implementing a set of distributed services providing global and dynamic management of all physical resources (memory, disk, processor and network interfaces). Global resource management allows to make transparent the distribution of resources in the cluster nodes and to take benefit of the whole cluster resources for demanding applications. Dynamic resource management allows to make transparent cluster reconfigurations (node addition or eviction) for the applications and to ensure the system high availability in presence of node failures. In addition, a checkpointing mechanism is provided to prevent applications from restarting their execution from the beginning in the event of a node failure.

Our system can be implemented as an extension to a standard single node OS. We have developed a prototype based on Linux kernel. A limited patch has been applied to Linux kernel which is otherwise extended by the standard module mechanism.

The remainder of this paper is organized as follows. Section 2 provides background on single system image mechanisms and systems. Section 3 gives an overview of the main features of Kerrighed and describes in more details the concept of container for global memory management, the process management mechanisms used to simply deploy a parallel application on the cluster and to migrate a process or thread. It also presents the dynamic resource management service dealing with system reconfigurations. In Section 4 preliminary results of a performance evaluation performed with our prototype are presented. Section 5 concludes.

¹ Kerrighed was previously named Gobelins. Kerrighed has been filed as a community trademark.

2 Background

To provide a single system image on top of a cluster global management of memory, processor and disk resources should be performed. A lot of work has already been done in global resource management. However, most of existing works focus on the management of one kind of resource. In this section, we first review previous work on global management of memory and processor resources. Less work has been done on systems that globally manage different kinds of resource as shown in Section 2.3.

2.1 Global Memory Management

Two kinds of mechanisms have been studied: remote memory paging and Distributed Shared Memory (DSM) systems.

Remote memory paging is an alternative to disk paging. The idea is to exploit the main memory of remote nodes instead of disk for paging [6,9,8]. In [6], a remote memory model is proposed in which a cluster contains dedicated remote memory servers that can be used by nodes having heavy paging activity. This idea is generalized in [9] which proposes to use the memory of idle nodes as a paging device. In [7], reliable remote paging is studied but few work deals with fault tolerance issues related to remote paging. The work described in [8] goes a step further by managing memory globally at the lowest level of the operating system. Thus virtual memory paging, mapped files and file system buffering are naturally integrated by using the low level global memory management algorithm. As only clean pages can be sent in global memory, node failures are tolerated.

Software DSM systems [15] are another way to manage memory in a cluster. It provides the illusion of a global shared memory on top of distributed memories. Node local memories are used as local caches of the shared data space. Shared data is migrated or replicated in the node memories. Many coherence protocols have been proposed to manage the coherence of multiple copies of the same data. Some research has been carried out to improve the reliability of DSM systems. However, few recoverable DSM described in the literature have been implemented [17].

2.2 Global Processor Management

Preemptive process migration [5] allows global management of the processor resource in a cluster. This mechanism may be supervised by load distribut-

ing algorithms to move a process from one node to another, in order to take advantage of available resources. However, few operating systems have implemented it [21,23,5,18]. In most of these systems, each workstation is individually owned thus limiting global processor management. Migration mechanisms implemented in message-passing distributed systems are often limited to individual processes which do not communicate with others and are thus not suitable to parallel applications.

Some form of global and integrated processor and memory management is provided in Mosix through the memory ushering algorithm [4]. This algorithm is activated when a node's free memory falls below a threshold value and attempts to migrate processes to other nodes which have sufficient free memory. Thus, process migration is decided not only based on processor load criterion but also taking into account memory usage.

2.3 Towards Single System Image

Since the late 80's, we assist to the emergence of systems combining the management of several resources. Sprite [19] is probably one of the first systems which has integrated within the same OS the management of several resources, namely disks and processors. Sprite is an operating system written from scratch, integrating a distributed file system ensuring a full transparency of file localization and a process migration mechanism. The advantage of combining these two mechanisms in a unique system is to offer a better transparency of the resource distribution and to limit the constraints met at the frontier between a resource managed globally and a resource managed locally. However, Sprite does not offer any global memory management mechanism, does not allow the migration of threads.

Millipede [10] is a user level system implemented on top of Windows NT. It combines global memory and processor management. Just like in Sprite, this approach allows a good transparency of the resource distribution and limits the constraints met at the frontier between the management of processors and memories. Indeed, in the Millipede system it is possible to migrate threads, thanks to the use of a DSM. Nevertheless, the user level implementation of Millipede introduces many programming constraints and limits its performance. For instance, data shared through the DSM must be explicitly handled by the programmer thanks to a set of specific functions. Moreover, the use of system calls is strongly limited. Lastly, the Millipede system cannot modify the file cache, page replacement or disk management mechanisms of the underlying Windows NT system.

Genesis system [11] is currently the most advanced SSI cluster operating sys-

tem. Based on the micro-kernel technology, it offers to programmers the choice of the parallel programming model: message passing or shared memory. Genesis implements a number of parallelism management and SSI services, in particular a resource discovery service, a global scheduler, currently implemented by a centralized server, providing both static processor allocation and dynamic load balancing and relying on efficient group process management operations, and a DSM providing the sequential and release consistency models.

Several other systems like Nomad [20], UnixWare NonStop Cluster [24] or SSIC [1] have been proposed. However, it appears that none of these systems ensures a global management of every cluster resource, allowing to really offer the view of a single machine on top of a cluster.

3 Overview of Kerrighed

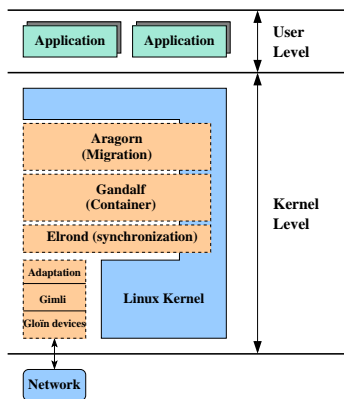


Figure 1. *Kerrighed architecture*

Figure 1 presents the architecture of Kerrighed cluster operating system. Kerrighed is composed of a set of distributed services providing global management of different logical resources in the cluster. Gandalf module is in charge of global memory management, Aragorn module is in charge of global process management, Elrond module provides a set of synchronization tools for parallel applications (atomic counters, locks, barriers, ...). An additional service called adaptation provides dynamic resource management: it deals with node eviction and addition to make such events transparent to other Kerrighed services. All these services are based on Gimli for their communications. Gimli is a service providing high performance communication inside the cluster. Gimli is itself based on Glon devices which constitute the hardware dependent communication system. This architecture provides a complete abstraction of the communication system to Kerrighed, makes it portable on various interconnection networks. A generic Glon device has been implemented to make the system completely independent from the network technology.

All Kerrighed services are implemented outside the Linux kernel using the standard module mechanism. In the remainder of this section, we describe in more detail the design of Gandalf, Aragorn and the adaptation service, which respectively offer mechanisms for global memory management, global processor management and dynamic resource management.

3.1 Global Memory Management Based on Containers

In a cluster, each node executes its own operating system kernel (called the *host operating system*), which can be coarsely divided into two parts: (1) *system services* and (2) *device managers*. For global memory management, we propose a generic service inserted between the system services and the device managers layers called *container* [22]. Containers are integrated in the core kernel thanks to *linkers*, which are software pieces inserted between existing device managers and system services and containers. The key idea is that a container gives the illusion to system services that the cluster physical memory is shared as in an SMP machine. We give in the remainder of this section a brief overview of container and linker mechanisms and how they can simply be used to provide a kernel level shared virtual memory system. A detailed description can be found in [16].

3.1.1 Container Definition

A container is a software object allowing to store and share data cluster wide as a COMA [12] architecture does. A container is a kernel level mechanism completely transparent to user level software. Data is stored in a container on host operating system demand and can be shared and accessed by the host kernel of other cluster nodes. Pages handled by a container are stored in page frames and can be used by the host kernel as any other page frame. Container pages can be mapped in a process address space or be used as a file cache entry.

By integrating this generic sharing mechanism within the host operating system, it is possible to give the illusion to the kernel that it relies on top of a physically shared memory. On top of this virtual physically shared memory, it is possible to extend to a cluster scale traditional services offered by a standard operating system (see Figure 2). This allows to keep the OS interface known by users and to take advantage of the existing low level local resource management.

The memory model offered by containers is sequential consistency implemented with a write invalidation protocol similar to [15]. This model is the one offered by a physically shared memory. Moreover, an injection mecha-

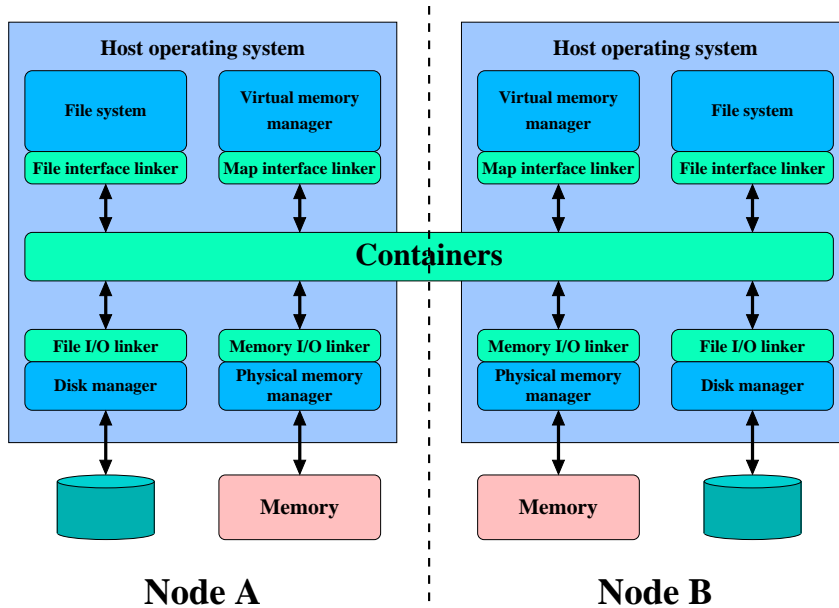


Figure 2. *Integration of containers and linkers within the host operating system*

nism similar to [8] is used to balance memory usage and avoid (or delay) disk swapping.

3.1.2 Linkers

Many mechanisms in a core kernel rely on the handling of physical pages. Linkers divert these mechanisms to ensure data sharing through containers. To each container is associated one or several high level linkers called *interface linkers* and a low level linker called *input/output linker*. The role of interface linkers is to divert device accesses of system services to containers while I/O linkers allow containers to access a device manager.

3.1.2.1 Connecting a Container to System Services System services are connected to containers thanks to interface linkers. An interface linker changes the interface of a container to make it compatible with the high level system services interface. This interface must give the illusion to these services that they communicate with traditional device managers. Thus, it is possible to "trick" the kernel and to divert device accesses to containers. It is possible to connect several system services to the same container. We have designed different interface linkers: one offering a *mapping* interface and one providing a *read/write* interface.

3.1.2.2 Data Input/Output in Containers During the creation of a new container, an input/output linker is associated to it. The container then

stops being a generic object and allows to share data coming from the device it is linked with. The container is said to have been instantiated. For each semantically different data to share, a new container is created. For instance, a new container is used for each file to share and a new container for each memory segment to share or to be visible cluster wide.

Just after the creation of a container, this one is completely empty; it does not contain any page and no page frame contains data from this container. Page frames are allocated on demand during the first access to a page. Similarly, data can be removed from a container when this one is destroyed or in order to release page frames when the physical memory of the cluster is saturated. These actions are performed by input/output linkers on containers demand. We have designed two I/O linkers: one dealing with memory data and one dealing with file data. A container linked to the first kind of I/O linker is called a *memory* container while one linked to the latter kind is called a *file* container.

3.1.3 Shared Virtual Memory

The virtual memory sharing service of an OS allows to share data between threads or between processes through a system V segment for instance. A shared virtual memory extends this service to a cluster scale.

A shared virtual memory allows several processes running on different nodes to share data through their address space. Providing this service requires to ensure three properties: (1) data sharing between nodes, (2) coherence of replicated data and (3) simple access to shared data thanks to processor read/write operations.

The container service ensures the two first properties. The third one is ensured by the *mapping* interface linker. Thus, mapping a memory container in the virtual address space of several processes via a mapping linker leads to a shared virtual memory.

When a process page fault occurs, the mapping interface linker diverts the fault to containers. The container mechanism places a copy of the page in local memory and ensures the coherence of data. Lastly, the mapping interface linker maps the local copy in the address space of the faulting process and changes the virtual page access right according to the rights of the page in the container.

3.2 Global Process Management

The process scheduling service of an OS allows to schedule processes on available processors of the architecture. The extension of this service to a cluster is a global scheduler. To place or move processes on available processors in the cluster, an efficient process migration mechanism is needed. This section is devoted to the description of Kerrighed process migration mechanism, which exploits containers.

In order to allow transparent process migration, processes first need to be identified in the cluster regardless of their location. Then, the process migration consists of three phases. The first one is the extraction and transfer of process information from the process source node to its destination node. The second one is the process state reception and the process restart on the destination node. The third one is the execution of the migrated process.

3.2.1 Global Management of Process Identifier

In a standard Linux kernel, processes are identified by a unique identifier: the *PID*. At cluster scale, this Linux *PID* cannot be used to identify a process independently from its location. Indeed, a *PID* used on a given node cannot be used anywhere in the cluster as it may correspond to a different process existing on another node. So, each process in Kerrighed is associated with a global identifier: the Kerrighed Process IDentifier (*KPID*). This *KPID* is composed of the identifier of the process creation node, of the thread number (zero for the master thread of a process), and of the master thread *PID* (process *PID* for mono-threaded applications). A thread manager is running on each node. This thread manager is able to communicate with all the threads running into the cluster. For example, when the operating system has to execute a function on each thread of a process, a request is broadcast to all the thread managers of the cluster which find the local threads associated with the *KPID* contained in the request, and then apply the function. This mechanism allows to manipulate threads (thread kill for example) and to globally manage the threads states.

3.2.2 Extraction of Process State for Migration

In Kerrighed operating system (like in the Linux system), a process state is a set of information: the process address space, the opened files list, the processor registers state and the process stack. Most of these informations are available from a kernel structure associated to each process (*task_struct*). The two most difficult issues for process migration are the migration of the process address space and the subsequent accesses to opened files. In this paper, we

focus on both the memory and the file issues and show how they can be solved using containers.

There are two kinds of memory management data: on one hand the page table which allows to solve paging requests, on the other hand some kernel structures to manage physical memory pages.

The kernel structure *mm_struct* manages the general memory information. The memory segments are managed by the *vm_area_struct* and all these segments (linear addresses) are associated with physical memory pages. In Kerrighed, some *vm_area_struct* are linked to a container (see Figure 3). These links allow to access remote memory pages.

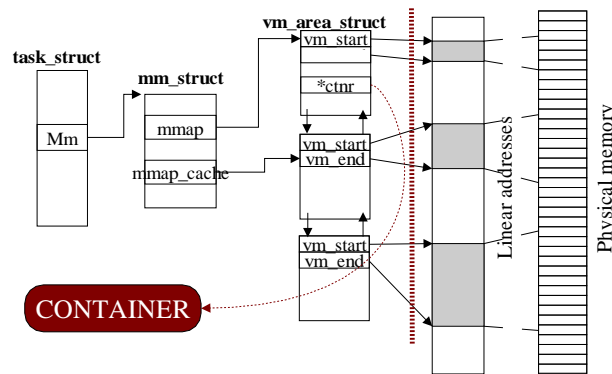


Figure 3. Kernel data structures of a process virtual memory with containers

Kernel file structures used for the management of opened files are more complex. There are three different file structures: *inode*, *file* and *dentry*. The *inode* structure allows to physically access a file on disk. But several processes can simultaneously access the same file. So, processes manipulate *file* structures instead of *inode* structures. The *dentry* structure is used to manage symbolic links. The kernel also manages two kinds of linked lists: the first one contains all files opened in the system while the second one contains files opened by each process. So, containers simplify the extraction of process informations for migration. For the memory information, we just need to extract *mm_struct*, *vm_area_struct* structures and container identifiers. We do not deal with page tables and physical page migration. For file memory information, we just need to extract *file* kernel structures and container identifiers. We do not have to deal with *dentry* and *inode* structures. Moreover, some memory segments are linked to a file. In Kerrighed, to provide access to remote files, a virtual inode and a virtual dentry linked to a container are created: every file data managed by a container is available all over the cluster (see Figure 4).

The values of the processor registers associated to each process are not available at any time in the system. In the Linux kernel, these values are available during an exception return, an interruption return, or after a system call return. So, we have created in the Linux kernel a new state to start Kerrighed

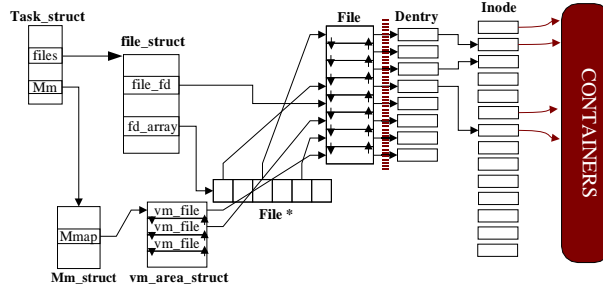


Figure 4. Kernel data structures of a process open file list with containers

mechanisms like process migration. This kernel modification is very light (just a few lines of code) and very similar to the kernel state for the signal treatment. If a process is marked as needing a Kerrighed mechanism (like marked for a signal treatment), the kernel is hijack to the Kerrighed process management module.

3.2.3 Process State Transfer

Kerrighed container mechanism allows the implementation of a kernel level shared virtual memory system over the cluster. Pages from a container do not need to be migrated during the process transfer. Only container identifiers need to be transferred in order to rebuild the link between memory segments (or files) and containers on the destination node. An interface linker is created and linked to a **vm_area_struct** with a container, which allows to use shared virtual memory. So, for migration of memory information, we just need to migrate **mm_struct** and **vm_area_struct** information. We do not have to migrate physical memory pages and to build the page table. All physical memory pages and virtual address space can be accessed through containers.

The container mechanism allows to share files accessed within the cluster. So, for process migration, we do not need to migrate all data structures representing files accessed by a process. Only high level open file data structures (**file** kernel structure) and container identifiers need to be migrated. With this data we can rebuild the link between high level kernel data structures and containers.

Only high level informations about memory and file accessed by a process need to be transfered. These informations are kernel structures which do not depend on the process memory size or on the number of accessed files. So, the size of information to be transfered is constant.

3.2.4 Execution of a Migrated Process

Once a process is migrated, it can resume his execution. Containers have two advantages for the execution of a migrated process. First, memory pages available in containers and accessed by the migrated process are copied on destination node on demand. So, for a sequential process, only pages accessed by the process are migrated on the destination node. If it is a thread (or a process) of a parallel application, it can always access shared data through containers.

Second, when a process accesses a file, data is loaded in the system file cache. These memory pages are available through containers. It is thus possible to access them from any node in the cluster. If a process accesses these pages (the migrated process or another one), they are transferred to its execution node through containers. So, containers allow to take advantage of a cluster cooperative file cache. If the process accesses a file which is not yet in the cooperative cache (a file which has not yet been accessed for example) and which is not locally present on the node, containers allow to make a remote file access.

Another important feature for process execution is the management signals. How a signal can be sent to a migrated process, and how a migrated process receives a signal ? A distributed mechanism to manage signals for migrated processes is needed. Our approach is to create a distributed kernel service for signal management: all nodes execute a kernel thread, the signal manager. A signal manager has a local table in order to localize migrated processes which were previously running on the local node, and to identify migrated processes which are locally running. So, a signal manager has two goals: to send signals destined to a migrated process to its destination node, and to receive signals sent by remote nodes and send them to their local destination processes.

3.3 Dynamic Resource Management for Dealing with Cluster Reconfigurations

We present in this section Kerrighed adaptation service on top of which other Kerrighed distributed services are built. This service performs dynamic resource management to make configuration changes in the cluster transparent to the services responsible of global resource management and consequently to applications. We first introduce some assumptions considered for the design of this service and the model of Kerrighed distributed services before describing the various components of the adaptation service and related implementation issues.

3.3.1 Assumptions

We call configuration the set of active nodes in the cluster. A node is considered to be active if it has not been detected failed by other active nodes and is not currently being added to or evicted from the cluster. The system is said to be in *stable state* when no configuration change is being processed. For the sake of clarity, we assume that only one modification in the cluster configuration (a node addition, eviction or failure) may occur at any time. In this context, we assume that a configuration change only happens when the system is in *stable state*. Thus a node failure cannot happen when the cluster OS is processing a node addition or eviction. We assume that the network is never partitioned, as a network partition would lead to the failure of several nodes at the same time.

We assume that the communication system guarantees that there is no message lost or duplication and that messages are delivered in the order they are sent. Moreover, we assume that messages are not altered during transmission. In this section, we present a generic dynamic resource management service that makes changes in the cluster configuration nearly transparent to the OS distributed services.

Node addition and shutdown are made completely transparent to the applications as Kerrighed provides a process migration mechanism and an injection mechanism for containers. A node failure is also transparent for checkpointed applications. Checkpointing [3] is out of the scope of this paper.

3.3.2 Kerrighed Distributed Service Model

Each Kerrighed distributed service in charge of the global management of a resource implements an abstraction. For example, Gandalf module for global memory management, implements the abstraction of container, the container being a set of pages. Each distributed service manages a directory. A directory entry contains global information about an instance of the abstraction implemented by the considered service. In Gandalf example, a directory is managed with an entry per page containing the identity of the node which is the current owner of the page (the owner of a page is the last node to have written the page). Directory entries are distributed on cluster nodes. Each node is then said to be the *manager* of a subset of the directory. The manager of a directory entry is often solicited by other nodes during the functioning of the service. For example, a Gandalf manager is solicited each time a page fault occurs.

3.3.3 Overview of the Adaption Service

The adaptation service is in charge of managing configuration changes in the cluster and to trigger reconfiguration of distributed services when needed (for example, after detection of a node failure). In the proposed architecture, the OS distributed services rely on the adaptation service (Figure 1). The adaptation service is designed to make reconfiguration changes transparent to any kind of distributed service.

Distributed services are programmed in an usual way. A generic interface has been defined between distributed services and the adaptation service. A generic protocol is used to handle any kind of configuration change in a cluster. We do not want to have as many different mechanisms as particular changes in the cluster configuration.

The adaptation service is in charge of the following tasks: (1) implementing a protocol for a node to inform other nodes that it is added or shut down, (2) detecting node failures, (3) providing a consistent view of the cluster configuration, (4) coordinating the protocol to deal with a cluster configuration change, (5) computing a distribution function that keeps the directory entries management balanced between managers, (6) locating the manager of a particular entry of the directory (with a *locator*), (7) dealing with migration of directory entries when a configuration change happens.

The adaptation service must be as independent as possible from a specific service. However, directory entries management is obviously performed by the related distributed service. We need to have some generic operations like acknowledgment, and specific operations for each service. Moreover, when a change occurs in the configuration all services need to perform some reconfiguration. In this way, for each service similar operations is needed at the same time.

A distributed service can use the adaptation service after a registration step. The specific functions are declared during this registration. This is the main reason why we split the adaptation service into two parts. First, a global mechanism defines some generic operations inside the adaptation service. Tasks specific to a service are provided by a plug-in system.

3.3.4 Interface between the Adaptation Service and Other Kerrighed Distributed Services

For the sake of clarity, we choose to describe the specific functions that a service has to provide on the example of Gandalf service, which is in charge of global memory management.

In the considered service, a directory entry corresponds to a virtual page. Let us take the example of a cluster with three nodes. We assume that a copy of Page 42 is located on Node 3 in a page frame, and its directory information is managed by Node 2. Let us consider a page fault (read request) on Page 42 on Node 1. Node 1 (as client) calls: `adaptation_send (PageReadRq, 42)`. In this context, the client makes a request on a page and it does not try to know where the request goes.

3.3.5 Adaptation Service Components

At any time (*adaptation period* or *stable state*) several components of the adaptation service take part in the proper operation of the cluster. These different components are described in the remainder of this section.

3.3.5.1 Supervisor A supervisor process is executed on each node. It is responsible of the addition or the shutdown of the node on which it executes. This is the supervisor that prepares its own node and notices the cluster.

The set of supervisors in the cluster cooperate in order to maintain a consistent view of the cluster configuration. In this way, a node *supervisor* participates to the failure detection protocol. When a node failure happens (or is suspected) a consensus protocol, which is out of the scope of this paper, is executed.

The supervisor (see Figure 5) starts with the setting and the addition of its node in the cluster (stage 1, 2, 3). The *stable state* (4) corresponds to a node outside an addition, eviction or failure stage. At some time, the supervisor may take part to the addition of a new node (stage 5) or to a node eviction or to the failure detection protocol (stage 6). The supervisor is the link between the node and the global state of the cluster, defined by the consensus protocol.

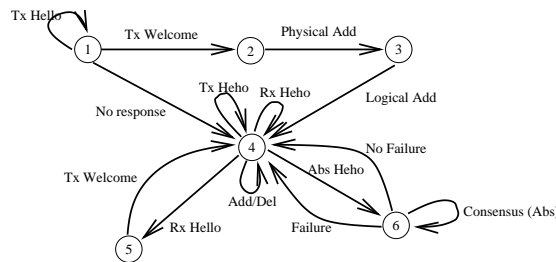


Figure 5. *Supervisor state*

When a configuration change happens in the cluster (see Figure 6), the supervisor makes an update of its communication layer (state 2) and, if needed, of the physical communication system. The next step (state 3) is the logical update, where the supervisor triggers directory entries migration. After mi-

gration of directory entries, the supervisors execute a consensus protocol and retrieve a *stable state*.

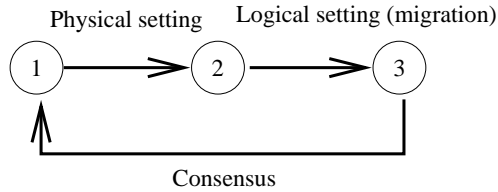


Figure 6. *Cluster state*

3.3.5.2 Locator The aim of the locator is to keep track of the node which is the manager of each directory entry. It must be quick to retrieve a directory entry.

A first approach is to assign directory entry managers to nodes with a static *distribution function* like *modulo*. Locating a particular directory entry manager is very simple and fast. However when a configuration change happens, the migration stage may affect all the nodes in the cluster and directory entries may be exchanged between two nodes that were already active before the configuration change[14]. When the cluster has got a large number of nodes, the directory entries migration step may become inefficient. This approach is not scalable.

In order to address the scalability issue, we choose to split up the whole set of nodes, in several subsets of limited size called cells. A tree of cells organizes the cells themselves. Each cell may contain other cells or nodes. The aim of a cell is to be a load-balancing unit: a bunch of directory entries is affected to a cell and when the composition of a cell is changed, the bunch of entries is re-distributed among the members of the cell. In this way, the distribution of directory entries affects only a few nodes in the cluster.

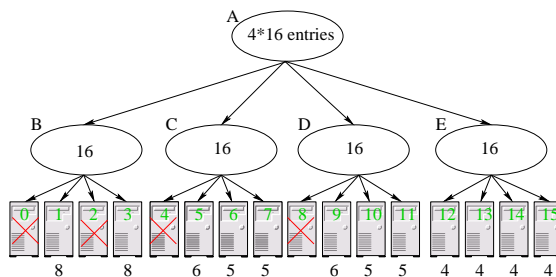


Figure 7. *Tree cell example*

In Figure 7, we have the example of a 2 level tree which corresponds to a cluster of 12 nodes. Each cell manages 16 directory entries. By assumption, Cell A manages 64 entries on 4 cells. There are 3 or 4 nodes affected to each of these cells. Despite that fact that these cells (B, C, D, E) have not the same size (2 to 4 nodes), they manage the same number of entries (16).

The same tree is used to distribute directory entries of all services existing in the cluster. In order to maximize the use of the different cells, the size of a service can be declared at its initialization and a particular service may be affected to a sub-tree instead of being affected to the whole tree. With this mechanism several small (in term of number of directory entries) services can be deployed on different nodes. Finally, the use of an array to store the deployment in a cell, makes the localization of the manager in a cell of a directory entry, constant in time.

The most interesting feature is that when a configuration change happens in the cluster, only one cell is affected by the operation. This way, the number of directory entries that migrate is limited. Moreover, in case of an addition, the most appropriate cell can be chosen in order to have the best load-balancing policy.

3.3.6 Cluster Reconfiguration

In this section, we present the adaptation service functions for the different cases of cluster configuration changes. We distinguish two kinds of events: foreseeable ones, namely node addition or eviction, and node failures. Protocols implemented in the adaptation service are described from two points of view: that of the node added to or evicted from the cluster, which we call requesting node, and that of any other active node in the cluster, which we call an observing node.

3.3.6.1 Foreseeable events In the event of a node addition or eviction, directory entries need to be migrated, in order to balance the number of directory entries managed by each node. When a configuration change is announced in the cluster, several communications may occur between nodes. The destinations of some of these communications may have changed due to the eviction of the destination node or the new directory entries distribution. These ongoing communications are delayed and not aborted. In this way, running applications do not notice the configuration change.

Requesting node In the foreseeable case, the node notifies every active node in the cluster about its situation. In all cases, it is able to prepare itself and starts the directory entries migration process. An added node only receives data from other nodes, while an evicted node only transmits data to other nodes.

Observing node The situation for an observing node is quite similar. When a notification of configuration is received, the directory entries migration phase is started. This migration happens in the *stable state* of the node.

Migration step Details of the protocol used for migrating directory entries

are represented in Figure 8. In the *stable state*, a node may receive (states 1, 2, 3) some directory entries from other nodes. This is the case of a node receiving directory entries while it has not yet received notification of the beginning of the configuration step. When a node receives the configuration change notification (state 4), it may receive other directory entries but also starts to transmit its own directory entries (states 5, 6). At that time the adaptation service makes transmissions, in a sequential way, for all registered services with the provided functions. Destination is one or several other nodes depending on the new directory entries distribution. At the end of the sending out step, the adaptation service sends to all nodes (including itself) an acknowledgement. The next step is to wait for directory entries (step 7) until all the *Ack* have been received. After receiving all acknowledgements, a node can finish the integration of the new data and recover its *stable state*.

From the point of view of the cluster, the configuration change ends when all the nodes finish their own part of the migration protocol.

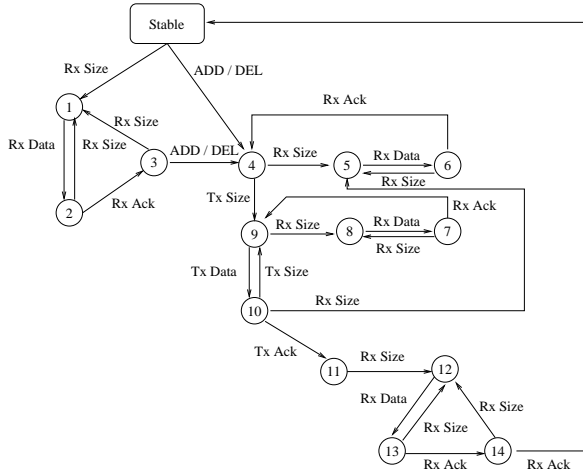


Figure 8. *Treatment of a node addition or eviction*

3.3.6.2 Node failure In the cluster, each node stores some data (for instance page copies) and directory information. In a first step, we only focus on the recovery of directory information, recovering data is to be handled by each particular service.

There are mainly two ways to recover directory information: using redundant directory information or rebuilding directory information at recovery. Managing redundant directory entries implies overheads during normal functioning. With the rebuilding method an overhead is incurred only when there is a node failure in the cluster. In fact, node failures are generally far less frequent than updates of directory information.

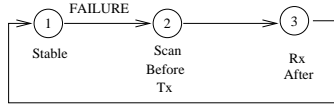


Figure 9. *Treatment of a node failure*

When all supervisors of active nodes agree on a failure, each node must scan its local directory information. This operation is performed by a “scan” function registered by each distributed service and automatically called by the adaptation service. When orphan data (for example a memory page previously managed by the failed node) is found on a node, this node temporarily takes the management of this data and rebuilds the corresponding directory entry. The next operations (see Figure 9) are the same as for a foreseeable change: the tree is updated and migration of directory information then takes place.

4 Performance Evaluation

The experimentation platform is made up of 6 PCs interconnected by a Gigabit Ethernet network. Each PC consists of a Pentium III 500 MHz processor, 512 MB of main memory. Each node runs Kerrighed operating system based on Linux 2.2.13.

4.1 Shared Memory based on Containers

First, we have evaluated the performance of containers when they are used to implement a shared virtual memory. We used a set of parallel algorithms programmed using POSIX threads according to a shared memory model. In this paper, we only present results obtained with the *MGS* algorithm which produces an orthonormal basis from an independent set of vectors, using the modified Gram-Schmidt algorithm. For each iteration, a new vector of the basis is computed by a processor and used by all other processors to correct the remaining vectors to standardize. A cyclic distribution of vectors is used.

Results presented in Section 4.1.1 were obtained without using the adaptation service. A *modulo* function is used to locate the manager node of a page. Results presented in Section 4.1.2 were obtained with a second version of Kerrighed where global resource management services are based on the adaptation service.

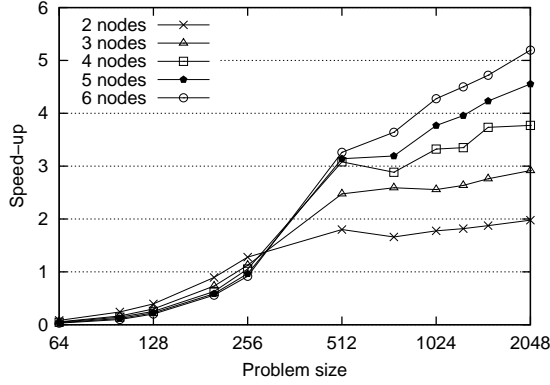


Figure 10. *Speed-up with Modified Gram-Schmidt Algorithm*

4.1.1 Static Resource Management

Figure 10 shows the speed-up obtained with the MGS algorithm. We varied the data set size from 64*64 to 2048*2048 double precision floating point elements and varied the number of node from 2 to 6. The speed-up increases slowly according to the size of the problem. Moreover, for a given problem size, the speed-up moves away quickly from the optimal speed-up when the number of nodes increases. Nevertheless, the speed-up reaches 5,2 on 6 nodes with a problem size of 2048*2048.

4.1.2 Dynamic Resource Management

During normal execution, the function for locating directory entries is called many times, for example to locate a page memory manager. So the efficiency of this *locate* function impacts on the global efficiency of Kerrighed operating system. We make the same measures as previously on different clusters (2, 3 and 4 nodes). For comparison, two versions of Kerrighed were used: one based on *modulo* (STAT), one based on the adaptation service (DYN). The overhead shown in Figure 11 is calculated by: $overhead = \left(\frac{DYN}{STAT} - 1 \right) * 100$.

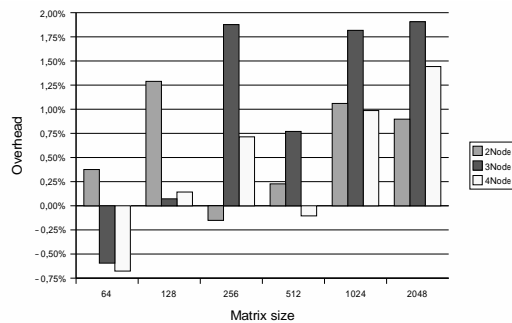


Figure 11. *Static / Dynamic overhead*

In all cases, the overhead is less than 2%. In four cases (64-3N, 64-4N, 256-2N

and 512-4N), the *dynamic version* is more efficient than the *static version*. As we indicate previously, the *static version* uses a distribution based on *modulo*. On another side, the *dynamic version* uses its own distribution that is different from *modulo*. In the particular case of Gram-Schmidt, the new distribution decreases the number of page requests across the network. Cluster with two nodes and four nodes are similar cases because in these configurations every node has exactly the same number of entries to manage. The worst case is a cluster with three nodes. In this case, the load of a virtual fourth node is spread on the three nodes. This repartition leads to a non-uniform load between the nodes.

4.2 Process Migration

We have compared the performance of Kerrighed process migration mechanism based on containers with the one of Mosix on the same platform. We have used a two node platform, each node being a Pentium II, 233 MHz with 128 MB of memory. The two nodes are interconnected by a 100Mb/s Ethernet network. Nodes communicate using the TCP protocol. For our experiments, we have considered the migration of a single mono-thread process executing a sequential version of the MGS algorithm. Different sizes of the MGS matrix have been used. Only one migration operation is activated (using the migration system call) during the process execution. Three different times have been chosen for triggering the migration: (i) one second after the beginning of the application, (ii) exactly in the middle of the application execution, (iii) three seconds before the end of the application execution. This allows us to quantify the impact of the application size on the migration performance.

First, we have evaluated the time needed to transfer the process state from the source node to the destination node. This time is measured as the time between the instant when the process is suspended on its source node and the time when it is ready to restart its execution on the destination node. For this set of experiments, the memory segments of the Kerrighed process are linked to containers when the process is created. In Kerrighed, the migration time is constant and equals 13.5 ms in average. Indeed, pages of the process address space are migrated on demand, once the process has resumed its execution on the destination node. In Mosix, the process transfer time increases with the process size as modified pages are transferred to the destination node during the process migration.

Secondly, we have measured the overhead due to migration on the process execution time. This overhead comprises the process transfer time and the overhead on the execution time of the migrated process on the destination node. Indeed, once the application is migrated, memory pages in containers

	150x150	300x300	500x500	750x750	1000x1000
Without migration	0,3149	3,329	16,3621	55,8434	133,17
Migration after 1 second of execution	0,3254	3,377	16,4636	56,0096	133,625
Migration exactly in the middle of the execution	0,334	3,4404	16,7976	57,0886	135,415
Migration 3 seconds before the end of execution	0,3341	3,4532	17,3988	58,0887	136,301

Table 1
Execution time of a process migrating once in Kerrighed

need to be migrated to the destination node on demand. Thus, the execution time of the process on its destination node is different from what it would have been on its source node if it had not been migrated. The overhead is calculated as the difference between the total application execution time obtained without migration and the total execution time obtained when the process migrates once during its execution. For this set of experiments, the Kerrighed process is created as a standard Linux process. The process address space is linked to containers when the process migrates for the first time. Thus, migration times provided in the following of this section for Kerrighed include the time needed to create containers for the application memory segments. Table 1 presents the migrating process execution time in Kerrighed and Figure 12 shows the migration overhead on the process execution time. The process execution time increases with the matrix size: the bigger the matrix is, the more pages need to be migrated. Even if the time needed to transfer pages increases with the matrix size, the migration overhead does not increase too. Figure 12 shows that the higher the computation time of the application is, the lower the migration overhead is². Moreover, it shows that the migration overhead in Kerrighed is between 0.5 and 7 percents of the execution time. The less the pages on the remote node are accessed, the less the overhead is.

Table 2 and Table 3 present comparable results for Mosix. Comparing with Kerrighed, it shows that Kerrighed process migration mechanism is more efficient than the Mosix one. In fact, in Mosix some pages transferred during the process migration are never referenced after migration. Other pages are migrated on demand as in Kerrighed. For the MGS application, the cost due to page transfers at migration time is not amortized in Mosix.

² A cache effect explains the result obtained for a 500x500 matrix and a migration 3 seconds before the end of the application.

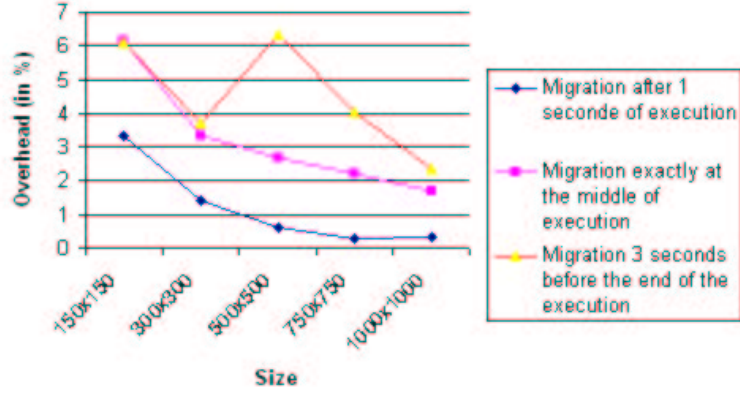


Figure 12. Overhead of a process migration on the application execution time in Kerrighed

	Migration after 1 second of execution	Migration exactly at the middle of the execution	Migration 3 seconds before the end of the execution
500	362.79	362.892	361.807
750	801.808	801.856	800.583
1000	1065.17	1065.01	1064.16
1250	1706.01	1767.8	1765.59
1500	1464.81	2119.09	2116.32

Table 2
Execution time of a process migrating once in Mosix

5 Conclusion

In this paper, we have presented the main features of Kerrighed cluster operating system which is unique in combining ease of use and programming by offering a Linux interface, high performance and high availability. We have presented encouraging results obtained from a preliminary performance evaluation of our prototype based on Linux kernel which has been lightly modified and extended with the standard module mechanism. This prototype is available as an open source software in the current version of the Kerrighed package (<http://www.kerrighed.org>).

The design of Kerrighed cluster operating system opens numerous research directions. We currently work on the optimization of global memory management using prediction mechanisms, on implementing efficient migratable sockets and on designing policies for global process scheduling. We also work on the implementation in the current prototype of checkpointing mechanisms

	500x500	750x750	1000x1000
Mosix	10.08	5.69	4.62
Kerrighed	0.62	0.3	0.34

Overhead with a migration after 1 second of execution

	500x500	750x750	1000x1000
Mosix	12.3	7.77	6.61
Kerrighed	2.66	2.23	1.69

Overhead exactly at the middle of the execution

	500x500	750x750	1000x1000
Mosix	11.21	8.3	7.27
Kerrighed	6.34	4.02	2.35

Overhead 3 seconds before the end of the execution

Table 3

Overhead of a process migration on the application execution time in Mosix

for parallel applications [2]. We have already studied checkpointing and recovery algorithms for applications based on the shared memory programming paradigm [13]. However, the implementation of these algorithms in a real multiprogrammed system is challenging. In our future work, we will also experiment our system with realistic OpenMP and MPI applications provided by industrial partners.

Acknowledgements

The authors would like to thank Viet Hoa Dinh who has implemented the Elrond module and Gimli generic network device and Louis Rilling who performed the Mosix measurements.

References

- [1] [<http://ssic-linux.sourceforge.net/>].
- [2] R. Badrinath and C. Morin. Common mechanisms for supporting fault tolerance in DSM and message passing systems. Rapport de recherche 4613, INRIA, Nov. 2002.
- [3] R. Badrinath, C. Morin, and G. Valle. Checkpointing and recovery of shared memory parallel applications in a cluster. In *Proc. Intl. Workshop on Distributed*

- Shared Memory on Clusters (DSM 2003)*, page ??, Tokyo, May 2003. Held in conjunction with CCGrid 2003. IEEE TFCC. To appear.
- [4] A. Barak and A. Braverman. Memory ushering in a scalable computing cluster. In *Proc. of IEEE third Int. Conf. on Algorithms and Architecture for Parallel Processing*. IEEE, Dec. 1997.
 - [5] A. Barak, S. Guday, and R. Wheeler. *The MOSIX Distributed Operating System*, volume 672 of *LNCS*. Springer Verlag, 1993.
 - [6] D. Comer and J. Griffioen. A new design for distributed systems: The remote memory model. In *Proc. of the USENIX Summer Conference*, pages 127–135, June 1990.
 - [7] E. M. G. Dramitinos. Adaptive and reliable paging to remote main memory. *Journal of Parallel and Distributed Computing*, 1999.
 - [8] M. Feeley, W. Morgan, F. Pighin, A. Karlin, and H. Levy. Implementing global memory management in a workstation cluster. In *Proc of the 15th ACM Symposium on Operating Systems Principles*, 1995.
 - [9] E. Felten and J. Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report 91-03-09, University of Washington, Mar. 1991.
 - [10] R. Friedman, M. Goldin, A. Itzkovitz, and A. Schuster. MILLIPEDE: Easy parallel programming in available distributed environments. *Software Practice and Experience*, 27(8):929–965, 1997.
 - [11] A. Goscinski, M. Hobbs, and J. Silcok. GENESIS: an efficient, transparent and easy to use cluster operating system. *Parallel Computing*, (28):557–606, 2002.
 - [12] E. Hagerstern, A. Landin, and S. Haridi. Ddm- a cache only memory architecture. *IEEE Computer*, 25(9):44–54, Sept. 1992.
 - [13] A.-M. Kermarrec and C. Morin. Smooth and efficient integration of high-availability in a parallel single level store system. In *Proceedings of Euro-Par 2001*, Aug. 2001.
 - [14] A.-M. Kermarrec, C. Morin, and M. Bantre. Design, implementation and evaluation of ICARE: an efficient recoverable DSM. *Software Practice and Experience*, 28(9):981–1010, July 1998.
 - [15] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *IEEE Transactions on Computers*, 7(4):321–359, Nov. 1989.
 - [16] R. Lottiaux, C. Morin, and G. Vallée. Containers: an architecture for an efficient cluster operating system. Publication interne 1442, IRISA, Feb. 2002.
 - [17] C. Morin and I. Puaut. A survey of recoverable distributed shared memory systems. *IEEE Transactions on parallel and Distributed Systems*, 8(9), Sept. 1997.
 - [18] F. D. J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software Practice and Experience*, 21(8):757–785, 1991.
 - [19] J. K. Ousterhout, A. R. Cherenon, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *Computer*, 21(2):23–36, Feb. 1988.
 - [20] E. Pinheiro and R. Bianchini. Nomad: A scalable operating system for clusters of uni and multiprocessors. In *Proceedings of the 1st IEEE International Workshop on Cluster Computing*, Dec. 1999.
 - [21] M. Powell and B. Miller. Process migration in DEMOS/MP. In *Proc. of the 9th Symposium on Operating Systems*, pages 110–119, 1983.

- [22] R.Lottiaux and C.Morin. Containers : A sound basis for a true single system image. In *Proceeding of IEEE International Symposium on Cluster Computing and the Grid*, pages 66–73, May 2001.
- [23] M. Theimer, K. Lantz, and D. Cheriton. Preemptable remote execution facilities for the V-System. In *proceedings of the 10th Symposium on Operating Systems Principles*, pages 2–12, Dec. 1985.
- [24] B. Walker and D. Steel. Implementing a full single system image unixware cluster: Middleware vs underware. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '99*, 1999.